Université de Montréal

Générateurs de nombres aléatoires modulo un grand entier, dont l'uniformité est assurée

par

Marc-Antoine Savard

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des arts et des sciences
en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Informatique

Orientation Recherche opérationnelle

janvier 2020

Université de Montréal

Faculté des arts et des sciences

Ce mémoire intitulé

Générateurs de nombres aléatoires modulo un grand entier, dont l'uniformité est assurée

présenté par

Marc-Antoine Savard

a été évalué par un jury composé des personnes suivantes :

Louis Salvail
(président-rapporteur)

Pierre L'Ecuyer
(directeur de recherche)

Fabian Bastin
(membre du jury)

Mémoire accepté le :
6 février 2020

Sommaire

Ce mémoire s'intéresse au générateurs de nombres aléatoires linéaires modulo un grand entier. Vérifier qu'un générateur possède de bonnes propriétés théoriques est essentiel pour la simulation Monte-Carlo. La famille de générateurs dont il est question produit des points possédant une structure de réseau bien connue pouvant être étudiée pour vérifier l'uniformité de ces générateurs. Le présent travail est motivé par la mise à jour du logiciel LatMRG qui permet d'étudier la structure de réseau de tels générateurs.

Ce mémoire présente d'abord les types de générateurs qui sont considérés par le logiciel. Il explique comment ils peuvent être utilisés pour produire des nombres et comment étudier la longueur de leur période. Il présente ensuite des équivalences entre certains membres de la famille dont l'utilisation permet de simplifier le travail dans LatMRG.

Il couvre ensuite la structure de réseau. En plus de décrire en quoi elle consiste, il explique comment la caractériser. On décrit une caractérisation de cette structure pour les générateur considérés. À partir de cela, on présente quelques algorithmes permettant d'extraire de l'information sur l'uniformité des générateurs.

Le mémoire fait ensuite la description du logiciel LatMRG. LatMRG contient un programme exécutable et une librairie. Ce mémoire présente leur raison d'être et décrit leurs fonctionnalités. Il décrit aussi diverses améliorations qui ont été faites au logiciel avec pour objectif principal de simplifier son utilisation. La description du logiciel s'accompagne de quelques exemples illustrant sa flexibilité et des voies de recherche intéressantes.

Mots-clés : nombres aléatoires, structure de réseau, simulation Monte-Carlo

Summary

This thesis is about linear random number generators with a large integer modulus. It is essential to be able to check that a generator has good properties in Monte-Carlo simulation. The generator family studied here produces points that possess a well known lattice structure that can be studied to assess the uniformity of these generators. The present work is motivated by the update of the LatMRG software which studies the lattice structure of the aforementioned generators.

This thesis first presents the different types of generators the software considers. It explains how they can be used to produce random numbers and how to study their period length. It then presents equivalences between some members of this family that are used to simplify LatMRG.

It then covers the lattice structure. The thesis describes what it is and how to characterize it. It describes the characterization of the lattice structure for the considered generators. From that, it presents a few algorithms that extract information on the uniformity of generators.

At last, the thesis describes the LatMRG software. LatMRG contains both an executable program and a library. The thesis presents both their purpose and their functionalities. It describes upgrades of the software that aim to simplify its usage. Along with the software description are a few examples that serve to illustrate the flexibility and future research avenues.

Keywords : random numbers, lattice structure, Monte-Carlo simulation, RNG

Table des matières

Liste des tableaux

Liste des figures

Liste des sigles

GNA: Générateur de nombres aléatoires

GLMm: Générateur linéaire modulo m

GCL: Générateur congruentiel linéaire

GCM: Générateur congruentiel multiple

GMR: Générateur multiplicatif avec reste

GCLM: Générateur congruentiel linéaire matriciel

GCMC: Générateurs congrutiels multiples combinés

LLL: Réduction de Lenstra, Lenstra et Lovàsz

BKZ: Réduction de Korkine-Zolotarev par blocs (Block Korkine-Zolotarev)

S&E: Séparation et évaluation

Remerciements

En premier lieu, je tiens à remercier Pierre L'Ecuyer, mon directeur de recherche, pour son soutien, ses conseils et son support financier. Il m'a proposé un projet intéressant, m'a fait découvrir le monde de la simulation et m'a permis de vivre des expériences uniques.

J'addresse aussi un merci tout particulier à Alexis qui a pris le temps de lire mon texte et qui m'a aidé à le rendre beaucoup plus « propre ».

Je remercie également ma copine Victoire. Si tu ne m'avais pas poussé un peu, peut-être que je serais encore en train d'écrire.

Merci à ma mère Caroline et à mon père Claude de ne pas m'avoir (trop) jugé quand j'ai entrepris le baccalauréat en mathématiques qui m'a mené ici.

Aux quelques autres, famille et amis, qui m'avez aidé lors de mes études, vous vous reconnaîtrez, merci!

Introduction

Générer des nombres aléatoires

De nombreux domaines, notamment la simulation Monte-Carlo, nécessitent la capacité de générer des nombres aléatoires. Il existe une large famille d'algorithmes dont l'objectif est de produire des valeurs qui imitent des variables aléatoires indépendantes et uniformes sur l'intervalle (0,1). On appelle ces algorithmes des générateurs de nombres pseudo-aléatoires ou, plus simplement par abus de langage, des générateurs de nombres aléatoires (GNA). Les différents GNA se distinguent les uns des autres quant à certaines propriétés, dont certaines sont plus importantes et désirables en simulation [**25, 27**].

(1) Propriétés théoriques : on veut que les points que l'on peut obtenir avec un GNA soient bien uniformes sur l'intervalle (0,1) et imitent correctement des variables indépendantes de la distribution uniforme.

(2) Propriétés statistiques : il est essentiel que les nombres obtenus par un générateur aient un comportement difficile à distinguer d'une véritable source aléatoire. Pour vérifier cela, on utilise des batteries de tests statistiques qui testent si la distribution des valeurs produites par le GNA est différente de celle désirée.

(3) Efficacité : on veut que la production des nombres aléatoire représente seulement une fraction de temps négligeable dans l'exécution d'une simulation. On veut également que le générateur utilise un espace mémoire raisonnable.

(4) Longue période : on verra plus loin que les générateurs déterministes sont périodiques. On veut que leur période soit plusieurs ordres de magnitude plus longue que la quantité de nombres désirée.

Les GNA doivent aussi idéalement posséder quelques propriétés dépendantes de leur implémentation :

- Répétabilité : il doit être facile de répéter une même séquence de nombres aléatoire. Cela est particulièrement pratique pour vérifier le bon déroulement d'un programme.
- Obtention de plusieurs sous-suites : les simulations à grande échelle nécessitent de simuler plusieurs processus en parallèle. Il faut plusieurs sous-suites aléatoires pour cela. Il existe aussi des techniques de réduction de variance qui dépendent de l'utilisation de sous-suites indépendantes.
- Facilité d'implémentation

Le présent mémoire s'intéresse principalement à présenter et étudier les propriétés théoriques des GNA linéaires modulo un grand entier m. C'est une grande et riche famille de générateurs décrite un peu plus loin. Il discute également de la plupart des autres propriétés pour cette famille.

Générateurs de nombres aléatoires : une définition

On définit un GNA $\mathcal{G}$ **[18]** par une structure mathématique $(\mathcal{S}, s_0, f, \mathcal{O}, g)$ où $\mathcal{S}$ est un espace d'états, $s_0 \in \mathcal{S}$ un état initial, $f : \mathcal{S} \to \mathcal{S}$ une fonction de transition, $\mathcal{O}$, un espace de sortie et $g : \mathcal{S} \to \mathcal{O}$ une fonction de sortie. On utilise ces éléments pour obtenir la séquence d'états du générateurs $\{s_n\}_{n\geq 0}$ avec $s_n = f(s_{n-1})$ et la séquence de sortie du générateur $\{u_n\}_{n>0}$ comme $u_n = g(s_n)$.

On définit les GNA par une relation de récurrence sur un espace d'états, qui est ensuite utilisée pour obtenir une *imitation de hasard* dans $\mathcal{O}$. En pratique, l'espace $\mathcal{S}$ est choisi fini. Autrement, il n'est pas possible de représenter tous ses éléments dans un espace mémoire fini, ce qui est problématique pour l'utilisation et l'implémentation. Cela fait aussi en sorte que f est périodique. Finalement, il est standard de choisir $\mathcal{O}$ comme l'intervalle (0,1) (ou [0,1] ou [0,1)). Cela vient du fait qu'il est possible d'obtenir une grande variété de variables aléatoires de distributions différentes à partir de variables aléatoires uniformes indépendantes **[46]**.

La suite $\{u_n\}$ peut être utilisée pour remplacer des variables aléatoires indépendantes identiquement distribuées $\mathcal{U}[0,1)$ ou des vecteurs $\mathbf{u}_n \sim \mathcal{U}(0,1)^t$ indépendants. Pour cela, il suffit de prendre $\mathbf{u}_n = (u_{nt}, \ldots, u_{nt+t-1})$. On définit l'ensemble $\Psi_t = \{\mathbf{u}_0 = (u_0, \ldots, u_{t-1}) : s_0 \in \mathcal{S}\}$ **[25]**. Il arrive parfois qu'utiliser toutes les valeurs de la suite $\{u_n\}$ pour obtenir des vecteurs puisse causer des problèmes sérieux. Il existe donc également un moyen d'obtenir des

structures similaires omettant certaines valeurs. Si $I = (i_1, \ldots, i_t)$ est un vecteur avec $i_k < i_l$ si $k < l$, alors on généralise $\mathbf{u}_n$ par $\mathbf{u}_n = (u_{ni_t+i_1}, \ldots, u_{ni_t+i_t})$ et Ψ_t par $\Psi_I = \{(u_{i_1}, \ldots, u_{i_t}) : s_0 \in \mathcal{S}\}$.

On note que Ψ_I correspond à l'ensemble de tous les points de dimension t pouvant être obtenus avec le générateur si $\mathbf{u}_n = (u_{ni_t+i_1}, \ldots, u_{ni_t+i_t})$. Puisque $\{u_n\}$ est périodique, $\{\mathbf{u}_n\}$ l'est aussi. Il suit que $\{\mathbf{u}_n\}$ est un échantillonnage uniforme (l'uniformité venant de la périodicité) de Ψ_I. Il est donc important que l'ensemble Ψ_I soit très uniforme sur $[0,1)^t$ pour que le générateur puisse également l'être.

Les générateurs linéaires modulo un grand entier

Dans ce mémoire, on s'intéresse à une famille de générateurs en particulier : celle des générateurs utilisant une congruence linéaire modulo un grand $m \in \mathbb{Z}$. Il est assez facile d'obtenir des nombres qui couvrent l'intervalle [0,1) à partir des entiers modulo m, $x \in \mathbb{Z}_m = \{0, \ldots, m-1\}$: on prend $u = x/m$. Compte tenu de cette propriété toute simple, on définit des GNA utilisant $\mathcal{S} = \mathbb{Z}_m^k$. On note les éléments s_n de $\mathcal{S}$ sous forme vectorielle $\mathbf{s}_n = (x_{n,0}, \ldots, x_{n,k-1})$, et on peut choisir g comme $g(\mathbf{s}_n) = x_{n,0}/m$.

Le plus complexe consiste à choisir f dans ce contexte. Dans ce mémoire, on impose la restriction que f soit linéaire. Bien que cela soit restrictif, il est possible d'obtenir des générateurs qui ont toutes les propriétés désirées. À titre d'exemple historique, les premiers générateurs de ce types ont été présentés par Lehmer en 1951 [**41**] en prenant $k = 1$, $a \in \mathbb{Z}_m$ et

$$f(s_n) = as_n \bmod m.$$

Il s'avère que la linéarité confère une structure particulière, que l'on nomme structure de réseau, aux vecteurs de Ψ_t. Notre connaissance des propritétés algébriques de $\mathbb{Z}_m$ permet de l'étudier.

LatMRG, un outil pour étudier cette famille de générateurs

Étymologiquement, le nom *LatMRG* vient de la contraction de *lattice*, le mot anglais désignant les réseaux, et *MRG*, le sigle anglais pour *multiple recursive generator* désignant un cas particulier (les générateurs congruentiels multiples) très important des générateurs

linéaires modulo un grand entier. LatMRG est en fait un outil logiciel et une librairie en C++ dont l'utilité est d'étudier les générateurs linéaires modulo un grand entier.

Le code originel de ce logiciel était écrit avec le langage Modula-2 et a été présenté par L'Ecuyer et Couture en 1997 [**32**]. Ce mémoire présente la nouvelle version traduite du logiciel. La traduction a donné l'opportunité d'implémenter toutes sortes d'améliorations au logiciel, qui sont la principale contribution de ce mémoire dont voici un court apperçu :

- L'efficacité des principaux algorithmes a été augmentée par l'utilisation de bornes plus serrées dans un problème d'optimisation.
- L'utilisation du logiciel a été simplifiée, ce qui le rend beaucoup plus général.
  - Il est maintenant facile d'utiliser le logiciel pour étudier Ψ_I pour I arbitraire.
  - Les exécutables et les fonctionalités principales de la librairie ont été réécrites pour alléger le code et simplifier sa prise en main.
  - Le format des fichiers de configuration est plus flexible et explicite.
- Les dépendances du logiciel ont changées : l'arithmétique, nécessitant parfois une précision arbitraire, utilise la librairie NTL [**52**] et les quelques fonctionnalités qui utilisaient la très complète, mais superflue, librairie Boost ont été réécrites.

Cependant, la nouveauté la plus importante à propos de LatMRG reste la publication du logiciel sur Github sous la licence Apache 2.0. LatMRG est disponible au public à l'URL https://github.com/savamarc/LatMRG.

Présentation du mémoire

Le chapitre 1 de ce mémoire s'occupe d'abord de recenser plus en détail les types de générateurs qui sont couverts dans LatMRG. On présente des résultats sur leur période, des équivalences entres différents choix de fonction de transition et le concept de sous-suites de GNA.

Le chapitre 2 rappelle la notion de réseau dans $\mathbb{R}^t$. Il présente également plusieurs concepts connexes tel que le réseau dual, les bases des réseaux et quelques algorithmes pour manipuler les réseaux. Ce chapitre présente également de quelle manière un générateur linéaire modulo un grand entier possède une structure de réseau.

Le chapitre 3 présente les fonctionnalités de LatMRG : quels types de générateurs peuvent être représentés et comment, les algorithmes importants du logiciel et les fonctions disponibles dans un exécutable. Cette description est accompagnée d'exemples présentant le logiciel. Ils couvrent la recherche de générateurs avec différents critères et l'utilisation de LatMRG pour tester ces générateurs.

Chapitre 1

Générateurs aléatoires linéaires modulo un grand entier

Dans ce premier chapitre, on présente les notions qui ont trait aux générateurs linéaires modulo m (GLMm) [**19, 28, 29**]. On présente d'abord les différents types de GLMm. On prend également le temps de décrire quelques équivalences existant entre eux.

On considère ensuite comment étudier la période de tels générateurs. Le plus gros du travail de cette section s'intéresse surtout à vérifier si les générateurs sont de période maximale, mais on explique également quelles sont les propriétés supplémentaires et désirables des GLMm de pleine période.

Après, on aborde le sujet de l'implémentation des GLMm. On présente une variété d'astuces employées lors de l'implémentation des générateur congruentiels. Il est question de l'efficacité des générateurs, mais aussi de leur précision. Une bonne partie de ces astuces permettent de s'assurer que l'arithmétique reste exacte, tout en utilisant tous les bits de précision que l'ordinateur met à la disposition de l'utilisateur.

Finalement, on discute de la génération de plusieurs suites de nombres aléatoires simultanément [**33**]. On établit quelles sont les techniques pouvant être utilisées en général, on détermine laquelle est la plus pertinente et on présente comment il est possible de l'appliquer avec les GLMm.

1.1. Différents types de générateurs linéaires modulo un grand entier

1.1.1. Générateur congruentiel linéaire

On présente d'abord le cas le plus simple de générateurs linéaire modulo un grand entier m : le générateur congruentiel linéaire (GCL). Il s'agit également de la première forme de GLMm présentée dans la littérature **[41]**.

Définition 1.1.1 (GCL)**.** *Soient $m \in \mathbb{Z}$, $\mathcal{S} = \mathbb{Z}_m$ et $a, x_0 \in \mathbb{Z}_m^*$. On définit un GCL en prenant $s_0 = x_0$ et*

$$s_n = f(s_{n-1}) = ax_{n-1} \bmod m. \tag{1.1.1}$$

On obtient des nombres dans [0,1) *avec la transformation $u_n = g(s_n) = x_n/m$.*

On choisit toujours $x_0 \neq 0$ et $1 < a < m$ pour éviter que la récurrence soit triviale. La période théorique maximale d'un tel générateur est de $m - 1$. Choisir m comme une puissance de 2 rend l'implémentation très rapide. Cependant, cela est problématique pour la longueur de la période : elle est maximale seulement si m est premier.

On peut aussi modifier la récurrence (1.1.1) pour pallier ce problème en ajoutant une constante $c \in \mathbb{Z}$:

$$s_n = f(s_{n-1}) = (ax_{n-1} + c) \bmod m. \tag{1.1.2}$$

Si $c \neq 0$, alors la période théorique maximale de cette récurrence est m. On préfèrera cette version puisqu'elle est plus générale. On verra plus loin que pour un a et un m donné les équations (1.1.1) et (1.1.2) ont essentiellement la même qualité en termes d'uniformité peu importe c.

La simplicité du générateur précédent cause cependant deux difficultés pratiques. D'abord, il est complexe d'implémenter efficacement un générateur de période suffisante. Dans la plupart des applications, on voudrait un générateur d'une période de longueur au moins 2^{100}. Pour ce faire, il faut prendre $m > 2^{100}$, ce qui veut dire qu'il faut manipuler des nombres plus grands que la précision standard des ordinateurs. Ensuite, la qualité des nombres obtenus n'est généralement pas très bonne **[5, 21]**.

C'est pour remédier à ces problèmes que les premiers générateurs congruentiels multiples (GCM) ont commencé à être utilisés.

Définition 1.1.2 (GCM). *Soient* $m \in \mathbb{Z} \geq 2$, $\mathcal{S} = \mathbb{Z}_m^k$ *et* $\mathbf{a}, \mathbf{s}_0 \in \mathbb{Z}_m^k$. *On note* $\mathbf{a} = (a_k, \ldots, a_1)$ *et* $\mathbf{s}_n = (x_n, \ldots, x_{n+k-1})$ *pour avoir*

$$x_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k}) \bmod m = \mathbf{a} \cdot \mathbf{s}_{n-k} \bmod m. \tag{1.1.3}$$

On appelle k *l'ordre de la récurrence. On obtient des nombres dans* [0,1) *avec la même transformation* $u_n = g(\mathbf{s}_n) = x_n/m$.

On choisit toujours $\mathbf{s}_0 \neq 0$ pour éviter que la récurrence soit triviale et $a_k \neq 0$ pour que l'ordre soit bel et bien k. La période théorique maximale de ces générateurs est de $m^k - 1$. Ce type de générateur est simplement une version plus générale du GCL : on prend une relation de récurrence d'ordre supérieur à 1.

On pourrait argumenter que ce type de générateurs est le plus important parmi ceux présenté dans ce mémoire. La plupart des autres générateurs étudiés ici peuvent être analysés comme un GCM. C'est pour cela que le logiciel dont on parle dans le chapitre 3 se nomme LatMRG, MRG étant le sigle anglais pour GCM.

1.1.2. Générateurs multiplicatifs avec reste

Les générateurs multiplicatifs avec reste (GMR) ont d'abord été introduits sous des formes particulières : les générateurs additifs avec reste et soustractifs avec emprunt [**45**]. Définissons $z_n = \pm x_{n-r} + x_{n-k} \pm c_{n-1}$. On peut définir ces générateurs par les récurrences suivantes :

$$x_n = z_n \bmod b, \quad c_n = \lfloor z_n/b \rfloor.$$

Ce type de récurrence peut être paramétré de manière à avoir une extrêmement longue période et s'exécuter très rapidement, mais a généralement une structure laissant à désirer [**5, 53**]. On préférera donc la forme générale qui ne restreint pas autant les paramètres [**6, 13**].

Définition 1.1.3 (Générateurs multiplicatifs avec reste (GMR)). *On prend $b \in \mathbb{Z}$ et $L \in \mathbb{Z}$ et on définit les récurrences*

$$x_n = d(c_{n-1} + \sum_{i=1}^{k} a_i x_{n-i}) \bmod b; \tag{1.1.4}$$

$$c_n = \left\lfloor \frac{1}{b} \left(c_{n-1} + \sum_{i=0}^{k} a_i x_{n-i} \right) \right\rfloor; \tag{1.1.5}$$

$$u_n = \sum_{\ell=1}^{L} x_{n+\ell-1} b^{-\ell}, \tag{1.1.6}$$

où $a_0, \ldots, a_k \in \mathbb{Z}$, $\text{pgcd}(a_0,b) = 1$ et $-a_0 d \equiv 1 \pmod b$. On nomme c_n le reste du générateur à l'étape n. La suite des nombres $\{u_n\}$ est dans $[0,1)$ et on l'utilise comme sortie du générateur.

En pratique, on prend b assez grand, par exemple 2^{64}, donc souvent $L = 1$. La période théorique maximale de ce type de générateur est $m - 1$ où $m = \sum_{i=0}^{k} a_i b^i$. Ces générateurs peuvent êtres vus comme une façon d'implémenter un GCL avec (potentiellement) un très grand modulo m.

Proposition 1.1.4 ([**8, 13, 54**]). *Un GMR est équivalent à un GCL avec $m = \sum_{i=0}^{k} a_i b^i$ et $a = b^{-1} \bmod m$. Si $\{y_n : y_n = a y_{n-1} \bmod m\}$ et $\{w_n : w_n = y_n/m\}$, la suite des nombres générés par ce GCL, alors :*

$$u_n = \lfloor b^L w_n \rfloor / b^L$$

si $0 < y_0 < m$ et

$$y_0 = b^k c_{k-1} + a_0 \sum_{i=0}^{k-1} x_i b^{k-i} - \sum_{j=1}^{k-1} b^j \sum_{i=1}^{j} a_i x_{j-i}.$$

Cela permet de déterminer la période du GMR avec les mêmes propriétés que pour un GCL. L'équivalence se fait au coût d'une perte de précision, mais en pratique cela n'a que peu d'impact. Si b et L sont bien choisis, on ne garderait pas plus de précision sur le nombre aléatoire du GCL qu'on ne le fait sur celui du GMR.

1.1.3. Générateurs matriciels

La récurrence (1.1.3) peut aussi être écrite sous la forme

$$\mathbf{s}_n = \begin{bmatrix} 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ a_k & a_{k-1} & \cdots & a_1 \end{bmatrix} \mathbf{s}_{n-1},\ n \geq k. \tag{1.1.7}$$

Si on change la matrice pour n'importe quelle matrice dans $\mathbb{Z}_m^{k\times k}$, alors $\{\mathbf{s}_n/m\}$ devient une suite de vecteurs pseudo-aléatoires. Cette méthode de génération est appelée générateur congruentiel linéaire matriciel (GCLM) [**1**].

Définition 1.1.5 (GCLM)**.** *Soient $m \in \mathbb{Z}$ et $\mathbf{0} \neq \mathbf{A} \in \mathbb{Z}_m^{k\times k}$. On prend $\mathbf{0} \neq \mathbf{s}_0 \in \mathbb{Z}_m^k$ et*

$$\mathbf{s}_n = \mathbf{A}\mathbf{s}_{n-1},\ n \geq k. \tag{1.1.8}$$

On obtient la suite de vecteurs pseudo-aléatoires $\{\mathbf{u}_n : \mathbf{u}_n = \mathbf{s}_n/m\}$.

En pratique, on peut aussi utiliser ce genre de générateur pour obtenir des vecteurs de dimensions quelconques. Notons

$$\mathbf{s}_n = (x_{nk}, \ldots, x_{nk+k-1})^t. \tag{1.1.9}$$

La suite $\{x_n\}_{n\geq 0}$ résultante permet de générer des nombres uniformes $\{u_n : u_n = x_n/m\}$ qui peuvent être utilisés de la façon standard pour générer des vecteurs aléatoires. On pourrait faire une correspondance différente entre les $\mathbf{s}_i$ et les x_n, mais deux points vont en faveur du choix fait ici.

(1) Avec cette notation, un GCLM est la généralisation d'un GCM puisque tout GCM est équivalent au GCLM où $\mathbf{A}$ est une puissance de matrice

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ a_k & a_{k-1} & \cdots & a_1 \end{bmatrix}^k.$$

Il sera possible d'étudier théoriquement un GCLM comme on le ferait avec un GCM.

(2) Pour obtenir un vecteur de t composantes, si $t = ak + b$ et $b < k$, on peut aussi prendre $\mathbf{w}_n = (\mathbf{u}_{(a+1)n}, \ldots, \mathbf{u}_{(a+1)n+a-1}, \lfloor \mathbf{u}_{(a+1)n+a} \rfloor_b)^t$ (le symbole $\lfloor \mathbf{u} \rfloor_b$ indiquant ici

le fait qu'on ne conserve que b composantes au vecteur $\mathbf{u}$). On concatène des vecteurs de la suite $\{\mathbf{u}_n\}$ en ne gardant que les b premières valeurs dans les $a+1$-ièmes vecteurs. Cette stratégie est cependant plus complexe à noter et ne change pas la théorie développée dans le prochain chapitre.

Les GCLM généralisent les GCM qui en sont un cas spécifique, mais aussi les GCL en les transformant en une forme matrice-vecteur. Les générateurs de cette forme ont été très étudiés depuis leur proposition par Grube en 1973 [**14**]. Il existe des implémentations modernes de ces types de générateurs, par exemple les générateurs MixMax [**49**].

1.1.4. Générateurs combinés

En général, les générateurs présentés plus haut ont des failles. Il peut être envisagé de combiner différents générateurs, avec différentes forces et faiblesses pour les compenser. On obtient alors un générateur que l'on appelle un GCM combiné (GCMC) [**16, 17, 37, 19**].

Définition 1.1.6 (GCMC)**.** *Soient $J \in \mathbb{N}$, $m_1, \ldots, m_J$ relativement premiers et*

$$x_{j,n} = (a_{j,1}x_{j,n-1} + \cdots + a_{j,k_j}x_{c,n-k_c}) \bmod m_j, \quad 1 \leq j \leq J.$$

Notons $\delta_1, \ldots, \delta_J$ des entiers avec $\text{pgcd}(m_j, \delta_j) = 1$*. On obtient des nombres dans* [0,1) *à partir des suites $\{x_{j,n}\}_n$ avec*

$$u_n = \frac{1}{m_1}\left(\sum_{j=1}^{J} \delta_j x_{j,n} \bmod m_1\right), \tag{1.1.10}$$

ou bien

$$w_n = \left(\sum_{j=1}^{J} \delta_j x_{j,n}/m_j\right) \bmod 1. \tag{1.1.11}$$

Notons $\rho_1, \ldots, \rho_J$ les périodes des différents GCM. On sait que la période de w_n divise $\text{ppcm}(\rho_1, \ldots, \rho_J)$.

Dans la définition précédente, on remarque que l'on utilise les suites $\{x_{j,n}\}_n$ telles que produites par un GCM. En pratique, il suffit d'avoir une suite $\{x_{j,n} : x_{j,n} \in \mathbb{Z}_{m_j}\}_n$ pour tout $1 \leq j \leq J$, pas nécessairement obtenue avec un GCM, pour calculer $\{u_n\}$ et $\{w_n\}$. En théorie par contre, la prochaine proposition justifie cette exigence.

Proposition 1.1.7 (GCM équivalent à un GCMC [**20**])**.** *Prenons la suite $\{w_n\}$ comme dans (1.1.11) à partir de J GCMs.*

On pose les définitions suivantes:

$$k = \max(k_1, \ldots, k_J), \tag{1.1.12}$$

$$m = \prod_{j=1}^{J} m_j, \tag{1.1.13}$$

$$n_j = (m/m_j)^{-1} \bmod m_j, \quad 1 \leq j \leq J, \tag{1.1.14}$$

$$a_i = \left(\sum_{j=1}^{J} \frac{a_{j,i} n_j m}{m_j} \right) \bmod m, \quad 1 \leq i \leq k. \tag{1.1.15}$$

Alors, $x_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k}) \bmod m$ *et* $\mu_n = x_n/m$ *définissent un GCM et si* $(w_0, \ldots, w_{k-1}) = (\mu_0, \ldots, \mu_{k-1})$*, alors* $w_n = \mu_n$ *pour tout* $n \geq 0$*.*

C'est-à-dire que si les nombres $x_{j,n}$ utilisés pour obtenir w_n sont les états de GCMs, alors w_n est également l'état d'un GCM. La proposition suivante illustre un résultat similaire sur u_n.

Proposition 1.1.8 ([**20**])**.** *On définit*

$$\Psi^+ = \{j : 2 \leq j \leq J \text{ et } \delta_j(m_j - m_1) > 0\},$$

$$\Psi^- = \{j : 2 \leq j \leq J \text{ et } \delta_j(m_j - m_1) < 0\}.$$

$$\Delta^+ = \sum_{j \in \Psi^+} \frac{\delta_j(m_j - m_1)(m_j - 1)}{m_1 m_j} + \sum_{j \in \Psi^-} \frac{\delta_j(m_j - m_1)}{m_1 m_j},$$

$$\Delta^- = \sum_{j \in \Psi^+} \frac{\delta_j(m_j - m_1)}{m_1 m_j} + \sum_{j \in \Psi^-} \frac{\delta_j(m_j - m_1)(m_j - 1)}{m_1 m_j},$$

Si $(w_0, \ldots, w_{k-1}) = (\mu_0, \ldots, \mu_{k-1})$*, alors* $u_n = (\mu_n + \epsilon_n) \bmod 1$*, où* $\Delta^- \leq \epsilon \leq \Delta^+$*.*

On remarque que les bornes Δ^+ et Δ^- sont plus serrées en prenant J ou $|m_j - m_1|$ plus petits. Ce résultat justifie le fait de prendre la suite u_n plutôt que la suite w_n dans les cas où son implémentation s'exécute plus rapidement.

1.2. Période des GLMm

Il a déjà été dit que pour qu'un générateur soit considéré bon, il doit avoir une période assez longue. La longueur de période nécessaire dépend bien sûr de l'application, mais pour qu'un séquence pseudo-aléatoire ait l'air aléatoire, elle doit être assez longue pour que seulement une infime partie de celle-ci soit utilisée par l'application qui s'en sert [**18**].

Prenons l'exemple d'un générateur dont 10^6 copies sont utilisées en parallèle pour produire 10^6 nombres par seconde pendant 10^6 heures. La quantité de nombres nécessitée par cette application est un peu plus petite que 2^{72}. D'une part, cela est bien supérieur à ce que la plupart des application nécessitent; d'une autre part, cela ne représente qu'une infime fraction d'une séquence qui contiendrait 2^{200} nombres. Le nombre 2^{200} sera parfois utilisé dans ce mémoire comme longueur de période intéressante pour cette raison.

Pour permettre aux générateurs d'atteindre des longueurs de période satisfaisantes, on étudie quelles sont les conditions nécessaires et suffisantes pour avoir un GLMm avec une pleine période. Cette section se penche sur deux cas particuliers.

(1) Puisque la période des GMR et des GCMC dépend directement de la période d'un GCM, il suffit d'étudier ce type de générateur pour obtenir des résultats qui couvrent tous ces générateurs.

(2) Les GCL nécessitent d'être étudiés séparément puisque leurs conditions de pleine période diffèrent légèrement.

L'équation (1.1.3) définit une relation de récurrence sur $\mathbb{Z}_m$. On définit le polynôme de cette récurrence comme $\varphi(x) = x^k - \sum_{i=1}^{k} a_i x^{k-i}$. On s'intéresse seulement aux cas où m est un nombre premier, puisque ce sera toujours le cas en pratique. Avec cette restriction, il est possible d'utiliser la théorie de Galois pour extraire de l'information décrivant la récurrence en considèrant l'extension de corps $\mathbb{F}_{m^k}/\mathbb{F}_m$.

Le cas le plus intéressant a lieu lorsque le polynôme $\varphi(x) \in \mathbb{F}_m[x]$ possède une racine primitive β dans $\mathbb{F}_{m^k}$. Alors

$$\mathbb{F}_m[x]/(\varphi(x)) \simeq \mathbb{F}_m(\beta) \simeq \mathbb{F}_{m^k}.$$

Dans ce cas, il s'avère que β est un élément d'ordre $m^k - 1$, c'est à dire que $\min\{k : \beta^k = 1\} = m^k - 1$. La récurrence du générateur se trouve alors à avoir une période de longueur égale à l'ordre de cet élément β. Les détails algébriques dépassent le cadre de ce mémoire, mais sont disponibles dans les manuels de référence de Dummit and Foote **[10]** et de Lidl et Niederreiter **[43]**. On retrouve dans la littérature la caractérisation suivante des polynômes avec une racine primitive.

Proposition 1.2.1 (**[15]**). *Si m est premier et que l'on pose $r = (m^k - 1)/(m - 1)$, alors les conditions suivantes garantissent que le polynôme caractéristique de la récurrence est primitif :*

(1) $((-1)^{k+1}a_k)^{(m-1)/q} \neq 1$ *pour tous les facteurs* q *de* $m-1$*;*

(2) $x^r \bmod \varphi(x) = (-1)^{k+1}a_k$*;*

(3) $x^{r/q} \bmod \varphi(x)$ *est de degré plus grand que 0 pour chaque* q *facteur de* r*.*

En pratique, ces conditions ne sont pas difficiles à calculer, mais factoriser $m-1$ et r peut être très coûteux. Cette proposition s'applique aussi dans le cas d'un GCLM. Il suffit en fait de vérifier les même conditions pour le polynôme caractéristique de la matrice pour s'assurer que le générateur est de pleine période [**46**].

Lorsqu'on s'intéresse à un GCL, le contexte est considérablement simplifié. Si $c = 0$, alors on veut simplement vérifier si a est primitif modulo m.

Proposition 1.2.2. *Le multiplicateur* a *est une racine primitive modulo* m *si et seulement si pour tout* q *divisant* $m-1$

$$a^{(m-1)/q} \bmod m \neq 1.$$

Si $a = 2^\omega$*,* $\omega \geq 1$*, alors* a *est primitif modulo* m *si et seulement si* 2 *est primitif modulo* m *et* $\text{pgcd}(\omega, m-1) = 1$*.*

Cette caractérisation s'applique aux GCL, mais aussi aux GMR [**13**]. Pour ceux-ci, il suffit de vérifier cette condition pour pour le GCL équivalent comme à la proposition 1.1.4 si m est premier. Si $c \neq 0$, cela modifie les conditions, on a une formulation plus générale pour m quelconque :

Proposition 1.2.3 ([**15**])**.** *Un GCL est de pleine période* m *si et seulement si*

(1) $\text{pgcd}(c,m) = 1$*;*

(2) tout q *premier qui divise* m *divise* $a-1$*;*

(3) si 4 *divise* m*, alors* 4 *divise* $a-1$*.*

Il est à noter que la proposition précédente exige indirectement que m ne soit pas premier puisque sinon $q = m$ divise m ne peut pas diviser a. Elle ne s'applique pas non plus au cas $c = 0$ à cause de la première condition.

Les GCMC quant à eux, dépendent seulement de la période de leurs différentes composantes.

Corollaire 1.2.4. *La période maximale pour un GCMC avec* J *composantes de périodes* $\rho_1, \ldots, \rho_j$ *est* $\rho = \text{ppcm}(\rho_1, \ldots, \rho_j)$*.*

Les conditions à respecter pour qu'un GCM ait une pleine période nécessitent que m soit premier. Il n'est donc pas possible d'avoir un GCMC avec la période $\rho_1 \cdots \rho_J$ si chaque

composant possède une pleine période. Dans ce cas, $m_j - 1$ est pair pour tout $1 \leq j \leq J$ et alors [**19**]

$$\text{ppcm}(\rho_1, \ldots, \rho_j) = \text{ppcm}(m_1^{k_1} - 1, \ldots, m_j^{k_j} - 1) \leq \frac{1}{2^{j-1}} \prod_{l=1}^{j} (m_l^{k_l} - 1) < m^{\max(k_1,\ldots,k_j)} - 1.$$

Pour que le générateur atteigne sa période maximale, on voudra que chacune de ses composantes soit elle-même de période maximale.

1.3. Implémenter les générateurs congruentiels

En pratique, l'efficacité d'un GNA est un des facteurs les plus important afin qu'il soit utilisé. Bien que ce ne soit pas la première préoccupation du présent document, elle reste nécessaire à considérer, particulièrement lorsque vient le temps d'implémenter les générateurs.

Pour obtenir une période de longueur intéressante avec un GLMm, on utilise généralement m près de la limite représentable par les ordinateurs. Puisque les états x_i du générateur peuvent prendre une valeur jusqu'à $m - 1$, effectuer des produits sur ceux-ci et conserver l'arithmétique exacte est l'autre principal défi à relever en implémentant les générateurs..

Cette section sert particulièrement à justifier les choix qui ont été fait pour les générateurs des exemples au chapitre 3 et leurs implémentations en annexe.

Dans cette section, les références aux GLMm, sauf en cas d'indication contraire, parlent spécifiquement des GCM. Les GCL sont aussi des GCM; on implémente un GCMC en implémentant chacune de ses composantes, des GCM, indépendamment; les GCLM ne possèdent pas de particularité qui modifie la discussion; il ne reste donc que les GMR dont certaines particularité peuvent être exploitées. Cela permet d'alléger le texte, sans compromettre l'information.

1.3.1. Éviter les débordments

Supposons que l'on ait un ordinateur qui fait l'arithmétique des entiers sur b bits. Il faut que les résultats des calculs intermédiaires des générateurs restent tous dans l'intervalle $[0,2^b - 1]$ ou alors tous dans l'intervalle $[-2^{b-1} + 1, 2^{b-1} - 1]$. Posons $m = 2^{b-e} - r$. Si e est petit, il est difficile d'éviter les débordements lors des multiplications.

La factorisation approximative [**31**] est un première astuce permettant d'éviter les débordements. Si $a^2 < m$, il est possible de calculer $ax \bmod m = (a(x \bmod q) - \lfloor x/q \rfloor p) \bmod m$

avec $q = \lfloor m/a \rfloor$ et $p = m \bmod a$. Il est à noter qu'on a $-m \leq a(x \bmod q) - \lfloor x/q \rfloor p \leq m$ et que cette stratégie impose la contrainte $a < \sqrt{m}$.

Algorithme 1.3.1 (Factorisation approximative)**.** *On connait $q = \lfloor m/a \rfloor$ et $p = m \bmod a$. L'algorithme suivant retourne le résultat de $ax \bmod m$.*

AF(x):

$k = \lfloor x/q \rfloor$

$r = a \cdot (x - q \cdot k) - kp + m$

retourne $r \bmod m$

Remarque 1.3.2. *Dans un langage comme le langage C, l'opérateur modulo* `%` *est plutôt un opérateur reste. C'est-à-dire que* `a%b == a-a/b*b`*. Si* `a < 0`*, cela fait en sorte que* `a%b < 0`*. C'est pour permettre d'avoir $r > 0$ que l'algorithme additionne m dans le calcul $r = a \cdot (x - q \cdot k) - kp + m$.*

Un autre option est de décomposer les coefficients comme des sommes de puissances de 2 [**38**]. Si $a_i = \sum_{n=1}^{N} 2^{q_n}$, on calcule $a_i x_i$ avec $\sum_{n=1}^{N} 2^{q_n} x_i$ où chaque produit est décomposé

$$2^q x \bmod m = (2^q x_0 + r x_1) \bmod m$$

avec $x_0 = x \bmod 2^{b-e-q}$ et $x_1 = \lfloor x/2^{b-e-q} \rfloor$. On obtient alors que $2^q x_0 < m$ et que $r x_1 < m$ si $r < 2^q$ et $r(2^q - (r+1)2^{q-b+e}) < m$.

Algorithme 1.3.3. *On connait $\mu = 2^{b-e-q} - 1$. On note $x >> n$ un décalage de n bits à droite de x, $x << n$ un décalage de n bits à gauche de x et $\wedge$ un ET bit à bit. L'algorithme suivant retourne le résultat de $2^q x \bmod m$.*

P2(x, m):

$k = (x \wedge \mu) << q + r(x >> (b - e - q))$

retourne $k \bmod m$

On mentionne en dernier lieu l'approche naïve permettant d'éviter les débordements : imposer $a(m-1) < 2^b - 1$. Cette option est difficile à mettre en place avec peu de précision; il est trop ardu d'obtenir des générateurs de période suffisante. Par contre avec 64 bits, ce n'est plus le cas. Pour produire une séquence de longueur 2^{200}, il n'est pas nécessaire de s'approcher de la limite $2^{64} - 1$. Tout GCM de pleine période avec $k = 4$ et $2^{50} < m$ le fait.

Il suffit de regarder les générateurs 32 bits pour s'en convaincre : MRG32k3a [**22**] possède une période relativement longue et ses produits ne débordent pas en 64 bits.

Avant de parler de l'efficacité de l'implémentation, il est nécessaire de glisser un mot sur les additions d'un générateur. À partir des bornes sur les résultats des produits ax que l'on obtient avec les techniques précédentes, on obtient des bornes sur les sommes $a_i x_i + a_j x_j$. En particulier, si $0 \leq a_i x_i < m$, alors $0 \leq a_i x_i + a_j x_j < 2m - 1$; cela impose de choisir $2m - 1 \leq 2^b - 1$. Il est aussi possible de jouer un peu avec les signes des multiplicateurs pour simplifier l'arithmétique. Puisque $ax \bmod m = (a - m)x \bmod m$, il est toujours possible de remplacer a par $a - m$. Alors, $-m < a_i x_i + (a_j - m)x_j < m$ peut être une contrainte préférable lors de l'implémentation. En augmentant le nombre d'additions, les bornes ne feront que grossir.

1.3.2. Améliorer l'efficacité

On peut subdiviser les techniques augmentant l'efficacité en deux catégories. Celles qui réduisent le nombre total d'opérations effectuées par le générateur et celles qui utilisent des opérations plus efficaces sur un ordinateur binaire.

Il est possible de réduire le nombre d'opérations du générateur en prenant certains de ces coefficients égaux à zéro. Cela diminue cependant la performance aux mesures présentées au prochain chapitre. C'est un des problèmes que les GCMC permettent de résoudre : leurs composantes peuvent être de mauvaise qualité, avec certains coefficients à 0, mais leur combinaison reste bonne [**20**] si elles sont choisies adéquatement. Ce phénomène est à l'œuvre dans l'exemple 3.3.2 au chapitre 3.

On peut aussi vouloir réduire le nombre d'additions effectuées par le générateur. C'est pourquoi on préfère généralement k petit. Pour une période donnée, il n'est possible de réduire k qu'en augmentant m. C'est pourquoi on choisit souvent m près de la limite représentable.

Les derniers types d'opérations dont on veut limiter l'usage sont les modulos et les comparaisons. On veut toujours ramener x_i dans l'intervalle $[0, m - 1]$, ces opérations sont donc inévitables, mais on souhaite n'avoir à les faire qu'une fois. Pour cela, il suffit d'effectuer tous les produits et les additions avant d'appliquer le modulo. Si k est grand, cela peut

donner des bornes assez larges sur le résultat avant d'appliquer le modulo. Dans l'exemple 3.3.2, c'est le principal facteur qui a influencé le choix du modulo.

Pour profiter des ordinateur binaires, on considère deux astuces : choisir m comme une puissance de 2 et ou utiliser des multiplicateurs qui sont des sommes de puissances de 2. Les multiplicateurs qui sont des sommes de puissances de 2 ont déjà été mentionnés dans la section 1.3.1. Les produits par de tels multiplicateurs sont effectués comme un décalage de bits qui peuvent donner un gain d'efficacité [**38**]. Choisir un modulo qui est une puissance de 2 permet d'effectuer l'opération modulo avec un ET bit à bit généralement plus rapide qu'un modulo. Cela est particulièrement utilisé avec les GCL si $c \neq 0$ ou les GMR, puisqu'il est possible d'avoir de bonnes conditions de période sur ces générateurs avec un modulo qui n'est pas premier.

1.4. Suites et sous-suites de GLMm

On termine ce chapitre par une brève section introduisant le concepts de sous-suites d'un générateur [**26, 31**]. Dans cette section, on utilise le terme *sous-suite*, qui possède un sens mathématique strict, assez libéralement. On utilise le terme *sous-suites de nombres aléatoires* pour parler de plusieurs suites de nombres aléatoires $S_1, \ldots, S_K$. Le *sous* vient du fait que, souvent, on obtienne ces suites à partir d'un seul GNA en changeant l'état initial. Si $c_k \in \mathbb{N}$ avec $1 \leq k \leq K$ sont des constantes avec $c_i \neq c_j$ si $i \neq j$, alors $S_k = \{s_{k,n} : s_{k,n} = u_{n+c_k},\ n \in \mathbb{N}\}$. On a que $S_k = \{u_n\}$ en tant qu'ensemble, mais que $s_{k,n} \neq u_n$. En pratique, on veut aussi qu'il existe un grand $N \in \mathbb{N}$ tel que les sous-segments $\{s_{i,n} : n < N\} \cap \{s_{j,n} : n < N\} = \varnothing$ si $i \neq j$.

On présente d'abord pourquoi et comment on veut utiliser plusieurs sous-suites d'un générateur. On s'intéresse ensuite aux différentes stratégies permettant d'obtenir plusieurs sous-suites. On expose finalement de quelle manière il est possible d'implémenter la génération de sous-suites pour les GLMm.

1.4.1. Utilisation de plusieurs suites

Il existe trois raisons de vouloir obtenir des sous-suites de nombres aléatoires.

(1) **Vérifier l'implémentation** : en implémentant une nouvelle simulation, on veut pouvoir répéter l'expérience avec exactement les même nombres pour vérifier certains

choix de paramètres et s'assurer que le code est exempt d'erreurs. Il est aussi pratique de pouvoir réutiliser les même nombres sur plusieurs ordinateurs différents. Il devient alors possible de partager le code afin qu'il soit examiné par quelqu'un d'autre.

(2) **Les calculs en parallèle** : lorsque l'on fait des calculs en parallèle, il est nécessaire d'avoir une suite de nombres aléatoires différente pour chaque processus de calcul. Des sous-suites bien construites auront ces propriétés. Ce genre d'approche peut aussi être utilisée pour faire des expériences en utilisant des processeurs graphiques par exemple [**33**].

(3) **Les techniques de réduction de variance** : l'utilisation de nombres aléatoires communs [**3, 24, 30**] permet de mieux comparer différents paramètres en simulant un système. La principale manière de faire cela est d'utiliser une sous-suite différente pour chaque source d'aléa et de réutiliser les mêmes sous-suites pour les mêmes aléas pour l'autre paramétrisation.

Peu importe la raison, il est coutumier de fournir un moyen d'obtenir plusieurs sous-suites dans l'implémentation d'un générateur. Dans un contexte de simulation, le choix de générateur est plutôt flexible. Il est souvent plus utile de choisir un générateur permettant d'avoir plusieurs suites qu'un générateur marginalement meilleur en termes statistiques.

1.4.2. Obtenir plusieurs suites

Trois approches différentes permettant d'obtenir des sous-suites de nombres aléatoires peuvent être envisagées [**33**]. La première est d'utiliser un générateur différent pour chaque sous-suite. C'est-à-dire que l'on choisit un type de générateur et on choisit différents paramètres pour chaque sous-suite. Même si cela est possible pour certains types de générateurs, cette approche reste plus coûteuse en termes de temps ou d'espace mémoire que les autres.

Avec les GLMm on pourrait, par exemple, obtenir chaque sous-suite comme un GCM avec $k = 7$ et $m = 2^{61} - 1$. Il suffit alors de changer les multiplicateurs $\mathbf{a}$ pour obtenir une sous-suite différente. Par contre, trouver un nouveau vecteur $\mathbf{a}$ avec de bonnes propriétés est coûteux en temps, en avoir déjà plusieurs bons nécessite un gros fichier de paramètres précalculés et le prendre au hasard peut donner de mauvaises sous-suites, ou des sous-suites dépendantes entre elles.

La deuxième option considérée est de séparer la période du générateur en plusieurs sous-sections de longueur égale. L'idée est simplement de choisir ν grand tel qu'il est possible d'obtenir $\mathbf{s}_{n\nu}$ pour tout $n \in \mathbb{N}$. Le seul prérequis pour pouvoir appliquer cette stratégie est qu'il faut être capable de calculer la fonction de transition du générateur pour autant d'étapes simultanées f^ν. La prochaine sous-section expliquera comment faire cela avec les GLMm. C'est cette technique qui sera préférée lorsque possible.

La dernière approche est de changer l'état initial du générateur aléatoirement. L'idée est que si la période du GNA est suffisamment grande, la probabilité que deux sous-suites obtenues à deux endroits dans la période du générateur aient une section en commun est négligeable.

Cette technique est surtout utile dans deux situations. D'abord, lorsqu'il n'est pas possible de d'avancer de plusieurs étapes simultanément. Ensuite, lorsqu'on simule des systèmes où l'attribution des sous-suites dépend des nombres obtenus dans certaines sous-suites. On veut alors un nombre aléatoire de sous-suites et on ne veut pas que celles-ci soient obtenues séquentiellement, autrement l'expérience ne sera pas reproductible.

1.4.3. Sous-suites de GLMm

On explique à présent comment il est possible de calculer la fonction de transition f^ν pour tout $\nu > 0$ pour un GLMm. On se rappelle qu'il est possible d'écrire l'équation d'un GCM sous une forme matricielle en utilisant la matrice $\mathbf{A}$ comme dans (1.1.7). On veut être capable de calculer $\mathbf{s}_{n\nu}$ pour $n \in \mathbb{N}$ et on sait que

$$\mathbf{s}_{n\nu} = \mathbf{A}^{n\nu}\mathbf{s}_0 \bmod m = (\mathbf{A}^{n\nu} \bmod m)\mathbf{s}_0 \bmod m.$$

Il est à noter que cette équation matricielle est aussi valable pour un GCL si $c = 0$ et un GCLM.

Pour être capable d'avancer de ν étapes, il suffit donc de connaître $\mathbf{A}^\nu \bmod m$ et ce calcul peut être implémenté efficacement. Il existe un algorithme diviser pour régner qui permet de faire l'exponentiation en $\mathcal{O}(\lg \nu)$ multiplications matricielles. Il suffit de décomposer les produit pour calculer $\mathbf{A}^a$, $a \in \mathbb{N}$ comme :

$$\mathbf{A}^a \bmod m = \begin{cases} (\mathbf{A}^{a/2} \bmod m)(\mathbf{A}^{a/2} \bmod m) \bmod m & a \text{ pair}, \\ \mathbf{A}(\mathbf{A}^{a-1} \bmod m) \bmod m & a \text{ impair}. \end{cases}$$

Pour un GCL avec $c \neq 0$, il faut un peu plus de travail. Il reste cependant possible d'obtenir une forme explicite de l'état x_a du générateur pour tout $a \in \mathbb{N}$ [**15**] :

$$x_\nu = \left(a^\nu x_0 + \frac{(a^\nu - 1)c}{a - 1} \right) \bmod m.$$

Lors de l'implémentation, quelques options sont à considérer. Il est possible de fixer ν. Dans ce cas, il faut fournir les valeurs de $\mathbf{A}^\nu$ (ou a^ν et $((a^\nu - 1)c)/(a - 1)$) avec le code pour calculer, au fur et à mesure que l'on en a besoin, les valeurs $\mathbf{s}_{(n+1)\nu} = \mathbf{A}^\nu \mathbf{s}_{n\nu} \bmod m$. Cela nécessite de toujours garder en mémoire la dernière valeur $\mathbf{s}_{n\nu}$ que l'on a générée. Il est aussi possible d'implémenter le calcul de $\mathbf{A}^\nu$ pour utiliser une valeur différente de ν à chaque fois que l'on génère une sous-suite différente. Les sous-suites ne sont plus nécessairement de longueur égale, et il est plus long d'en obtenir une nouvelle, mais cela permet plus de flexibilité [**2**].

Chapitre 2

Réseaux, structure et étude des générateurs linéaires

Ce chapitre présente les réseaux, une structure algébrique qui apparaît naturellement lorsque l'on étudie les points générés par un GLMm. Dans le cadre des GLMm on appelle cela la *structure de réseau*.

On présente d'abord en quoi consiste un réseau. Cela couvre la définitions, ainsi que les concepts de base et de dual. On parle également de comment on peut représenter et construire un réseau. Ensuite, on couvre la structure de réseau des GLMm. On explique de quelle manière elle survient et comment en construire une représentation. Finalement, on décrit quelles sont les propriétés d'un réseau qui peuvent être utilisées pour quantifier la qualité d'un ensemble de points comme distribution uniforme. Cela fait appel aux empilements compacts et à quelques notions d'algèbre linéaire.

2.1. Réseau et base

On commence par présenter le concept de réseau de la manière dont il est utilisé dans ce chapitre.

Définition 2.1.1 (Réseau sur les entiers)**.** *Soient $t \in \mathbb{N}$ et $\mathbf{v}_1, \ldots, \mathbf{v}_t$ des vecteurs de $\mathbb{R}^t$ linéairement indépendants sur $\mathbb{Z}$. Un réseau sur les entiers en dimension t, L_t, est l'ensemble de points dans $\mathbb{R}^t$ généré par les combinaisons linéaires sur $\mathbb{Z}$ des vecteurs $\mathbf{v}_1, \ldots, \mathbf{v}_t$. On écrit*

$$L_t := \left\{ \sum_{j=1}^{t} z_j \mathbf{v}_j : z_j \in \mathbb{Z},\ 1 \leq j \leq t \right\}. \tag{2.1.1}$$

On appelle $\mathbf{v}_1, \ldots, \mathbf{v}_t$ une base de ce réseau.

Le concept de réseau est l'analogue discret à celui d'espace vectoriel. La base d'un réseau n'est pas unique, mais un réseau est tout de même entièrement caractérisé par sa base. Ce n'est pas n'importe quel ensemble de vecteurs linéairement indépendants dans le réseau qui décrit une base. Si $\mathbf{v}_1, \ldots, \mathbf{v}_t$ est une base, $2\mathbf{v}_1, \ldots, \mathbf{v}_t$ sont linéairement indépendants, mais ne forment pas une base.

Définition 2.1.2 (Matrice de base)**.** *On note la matrice de base du réseau*

$$\mathbf{V} = \begin{bmatrix} \mathbf{v}_1 & \cdots & \mathbf{v}_t \end{bmatrix}^t.$$

Dans cette matrice, chaque ligne est un des vecteurs de la base du réseau.

Cette notation d'une base sera utile pour manipuler les réseaux. On présente maintenant le réseau dual, une autre notion centrale de ce chapitre.

Définition 2.1.3 (Réseau dual)**.** *On définit le réseau dual de $L_t \subset \mathbb{R}^t$ comme*

$$L_t^* = \{\mathbf{h} \in \mathbb{R}^t \mid \mathbf{h} \cdot \mathbf{v} \in \mathbb{Z}, \forall \mathbf{v} \in L_t\}. \tag{2.1.2}$$

L'ensemble L_t^* est un réseau et on note sa matrice de base $\mathbf{W}$.

On appelera simplement ce réseau le dual de L_t. L'importance du dual apparaîtra plus évidente dans la section 2.3.

Tel que spécifié plus haut, un réseau est entièrement caractérisé par sa base. De plus, la relation entre un réseau et son dual fait en sorte que ce dernier est également caractérisé par la base du réseau.

Définition 2.1.4. *Une matrice $\mathbf{M}$ est dite unimodulaire si $|\mathbf{M}| = 1$ et ses coefficients sont entiers. L'inverse d'une matrice unimodulaire est également unimodulaire.*

Proposition 2.1.5. *On dit que $\mathbf{V}_1 \sim \mathbf{V}_2$ si elles sont de mêmes dimensions et s'il existe une matrice unimodulaire $\mathbf{U}$ telle que $\mathbf{U}\mathbf{V}_1 = \mathbf{V}_2$. Ceci est une relation d'équivalence. Si $\mathbf{V}$ est une matrice de base pour L_t, alors $\widetilde{\mathbf{V}} \sim \mathbf{V}$ est également une matrice de base pour L_t.*

Démonstration. On démontre d'abord que $\sim$ est une relation d'équivalence.

(1) On a que $\mathbf{V}_1 \sim \mathbf{V}_1$ en prenant $\mathbf{U} = \mathbf{I}$.

(2) Soient $\mathbf{V}_1 \sim \mathbf{V}_2$ et $\mathbf{U}$ unimodulaire avec $\mathbf{U}\mathbf{V}_1 = \mathbf{V}_2$. Puisque $\mathbf{U}$ est unimodulaire, ses coefficients sont entiers et $\mathbf{U}^{-1}$ est également unimodulaire. Donc $\mathbf{V}_1 = \mathbf{U}^{-1}\mathbf{V}_2$ ce qui implique que $\mathbf{V}_2 \sim \mathbf{V}_1$.

(3) Supposons que l'on ait également $\mathbf{V}_3$ avec $\mathbf{V}_2 \sim \mathbf{V}_3$ et $\mathbf{W}$ unimodulaire telle que $\mathbf{W}\mathbf{V}_2 = \mathbf{V}_3$. Alors, $\mathbf{W}\mathbf{U}\mathbf{V}_1 = \mathbf{V}_3$ ce qui implique que $\mathbf{V}_1 \sim \mathbf{V}_3$.

Pour montrer que $\widetilde{\mathbf{V}} \sim \mathbf{V}$ est une base de L_t, il suffit de montrer que $\widetilde{\mathbf{V}}$ contient des vecteurs du réseau et qu'il est possible de générer les vecteurs de $\mathbf{V}$ à partir des vecteurs de $\widetilde{\mathbf{V}}$.

Si $\widetilde{\mathbf{V}} \sim \mathbf{V}$, alors $\mathbf{V} \sim \widetilde{\mathbf{V}}$. Soit $\mathbf{U}$ avec $\mathbf{U}\mathbf{V} = \widetilde{\mathbf{V}}$. Comme les coefficients de $\mathbf{U}$ sont entiers, il suit que les vecteurs de $\widetilde{\mathbf{V}}$ sont dans L_t. De plus, $\mathbf{U}^{-1}$ est aussi entière. Comme $\mathbf{U}^{-1}\widetilde{\mathbf{V}} = \mathbf{V}$, il suit qu'il est possible de générer $\mathbf{V}$ à partir de $\widetilde{\mathbf{V}}$. On a que $\widetilde{\mathbf{V}}$ est une base de L_t. □

On caractérise un réseau et un dual par leurs bases grâce aux propriétés suivantes.

Proposition 2.1.6. *Soient* $\mathbf{V}$ *et* $\mathbf{W}$ *les bases d'un réseau et de son dual. Alors,*

$$|\mathbf{V}\mathbf{W}^t| = \pm 1.$$

Il existe une matrice $\widetilde{\mathbf{W}}$ *telle que* $\mathbf{W} \sim \widetilde{\mathbf{W}}$ *et* $\mathbf{V}(\widetilde{\mathbf{W}})^t = \mathbf{I}$.

Démonstration. On montre d'abord que $(\mathbf{V}^{-1})^t = \widetilde{\mathbf{W}}$ est une base du dual. Puisque la matrice est inversible, il est clair que les vecteurs de $\widetilde{\mathbf{W}}$ sont linéairement indépendants. Il faut s'assurer qu'ils génèrent le dual. Un vecteur $\mathbf{h}$ du dual est tel que $\mathbf{h} \cdot \mathbf{v}_i = p_i \in \mathbb{Z}$. De plus $\mathbf{V}\mathbf{h} = \begin{bmatrix} p_1 & \cdots & p_t \end{bmatrix}^t$. On note les lignes de $\widetilde{\mathbf{W}}$ comme $\tilde{\mathbf{w}}_1, \ldots, \tilde{\mathbf{w}}_t$. On remarque que $\sum_{i=1}^{t} p_i \mathbf{w}_i = \mathbf{h}$ puisque

$$\mathbf{V}\left(\sum_{i=1}^{t} p_i \mathbf{w}_i\right) = \sum_{i=1}^{t} p_i \mathbf{V}\mathbf{w}_i = \begin{bmatrix} p_1 & \cdots & p_t \end{bmatrix}^t,$$

et $\mathbf{V}\mathbf{h} = \begin{bmatrix} p_1 & \cdots & p_t \end{bmatrix}^t$ admet une solution unique.

De ce qui vient d'être démontré, on déduit qu'il est possible d'écrire une matrice $\mathbf{U}$ telle que $\mathbf{U}\widetilde{\mathbf{W}} = \mathbf{W}$ pour toute matrice de base duale $\mathbf{W}$. Si les lignes de $\mathbf{W}$ sont $\mathbf{w}_1, \ldots, \mathbf{w}_t$, alors $\mathbf{V}\mathbf{w}_i = \begin{bmatrix} u_{1i} & \cdots & u_{ti} \end{bmatrix}^t$ pour tout $1 \leq i \leq t$ et on a

$$\mathbf{U} = \begin{bmatrix} u_{11} & \cdots & u_{t1} \\ \vdots & \ddots & \vdots \\ u_{1t} & \cdots & u_{tt} \end{bmatrix}.$$

Pour conclure la preuve, il suffit de vérifier que $\mathbf{U}$ est unimodulaire. Pour cela, il faut que $\mathbf{U}$ soit inversible et que son inverse soit entière. Puisque $\mathbf{W}$ est une base du réseau dual,

il suit que pour tout vecteur $\tilde{\mathbf{w}}_i$, il existe une combinaison linéaire entière de $\mathbf{w}_1, \ldots, \mathbf{w}_t$ avec $\sum_{j=1}^{t} z_j \mathbf{w}_j = \tilde{\mathbf{w}}_i$. On assemble ces coefficients pour construire $\mathbf{U}^{-1}$. □

2.2. Réseau des GLMm

Tel qu'indiqué précédemment, les GLMm possèdent ce que l'on appelle une structure de réseau [**15, 18, 32**]. Soit I un ensemble de t indices et Ψ_I les vecteurs obtenus avec un GLMm. Alors il existe un réseau L tel que

$$L \cap [0,1) = \Psi_I. \tag{2.2.1}$$

Spécifiquement, on a que la continuation périodique $\Psi_I + \mathbb{Z}^t = L$ est un réseau. Étudier l'uniformité de ce réseau donne donc de l'information sur l'uniformité du GLMm.

En pratique, on veut être capable de décrire la base d'un réseau et de son dual sans perdre de précision malgré la mémoire finie des ordinateurs. On veut représenter les réseaux en utilisant seulement des nombres entiers puisque tout nombre entier peut être décrit avec une précision finie. Cela veut dire qu'on ne veut pas directement manipuler $\Psi_I + \mathbb{Z}^t$ lorsque l'on étudie un GLMm.

Définition 2.2.1 (Changement d'échelle d'un réseau)**.** *Soit $L_t \subset \mathbb{R}^t$, une matrice de base $\mathbf{V}$, $L_t^* \subset \mathbb{R}^t$ et une matrice de base duale $\mathbf{W}$.*

On dit que nL_t est le changement d'échelle par $n \in \mathbb{N}^*$ du réseau. Un réseau qui a subi un tel changement d'échelle admet $n\mathbf{V}$ comme matrice de base.

Si on change d'échelle le dual de L_t par $n \in \mathbb{Z}$, on nomme le nouveau réseau nL_t^* le n-dual de L_t.

La structure de réseau des GLMm possède un changement d'échelle qui permet de s'assurer qu'elle est entière. On définit $L_t(I) = m(\Psi_I + \mathbb{Z}^t)$ et on note $L_t = m(\Psi_t + \mathbb{Z}^t)$ dans le cas particulier $I = \{0, \ldots, t-1\}$. Ce réseau est dans $\mathbb{Z}^t$, possède la même structure que dans l'équation (2.2.1), mais sa matrice de base $\mathbf{V}$ est entière. On s'intéresse à la construction d'une base pour ce réseau.

Il est possible de construire cette base explicitement. On commence par présenter le cas particulier d'un GCM (et d'un GCL) avec $I = \{0, \ldots, t-1\}$.

Proposition 2.2.2 (Base d'un GCM)**.** *Si $t \leq k$, alors $L_t = \mathbb{Z}^t$ puisque construire $m\Psi_t$ consiste à tronquer tous les vecteurs de $\mathbb{Z}^k$ à t composantes.*

Si $t > k$, on peut obtenir la base directement tel qu'expliqué dans la littérature **[25, 32]**. *Notons $e_d(i)$ le i-ème vecteur de la base canonique de $\mathbb{Z}^d$ et $x_{i,j}$ la valeur de x_j si $\mathbf{s}_0 = e_k(i)$. Alors les vecteurs*

$$\begin{aligned}
\mathbf{v}_1 &= (1, 0, \ldots, 0, x_{1,k}, \ldots, x_{1,t-1})^t \\
&\vdots \\
\mathbf{v}_k &= (0, 0, \ldots, 1, x_{k,k}, \ldots, x_{k,t-1})^t \\
\mathbf{v}_{k+1} &= (0, 0, \ldots, 0, m, \ldots, 0)^t \\
&\vdots \\
\mathbf{v}_t &= (0, 0, \ldots, 0, 0, \ldots, m)^t
\end{aligned}$$

sont une base pour L_t.

En connaissant cette base, on peut trouver une base pour $L_t(I)$, un choix d'indices général $I = \{i_1, \ldots, i_t\}$, toujours pour un GCM. Pour cela, on doit d'abord construire la base de L_{i_t}. Il est possible de déduire que le réseau généré par les colonnes $i_1, \ldots, i_t$ des la matrice de base de L_{i_t} engendrent $L_t(I)$ **[7]**. Il n'a cependant pas été présenté par quelle méthode il est possible d'obtenir une base d'un réseau à partir d'un ensemble générateur.

Algorithme 2.2.3. *Soient des vecteurs $\mathbf{v}_1, \ldots, \mathbf{v}_\ell$ dans $\mathbb{Z}^t$, avec $\ell \geq t$, générant un réseau L_t. On met ces vecteurs comme lignes d'une matrice $\mathbf{V}$.*

BaseRéseau(V, t, l):

pour $i = 1$, $i \leq t$**:**

pour $j = i$, $j \leq l$**:**

tant que $\mathbf{V}_{ji} \neq 0$**:**

échanger $\mathbf{v}_j$ et $\mathbf{v}_i$

$q = \lfloor \mathbf{V}_{ji} / \mathbf{V}_{ii} \rfloor$

$\mathbf{v}_j = \mathbf{v}_j - q\mathbf{v}_i$

si $\mathbf{V}_{ii} < 0$**:**

$\mathbf{v}_i = -\mathbf{v}_i$

enlever les lignes $t + 1$ à l de $\mathbf{V}$

La matrice $\mathbf{V}$ résultante est une base de L_t.

Remarque 2.2.4. *Dans l'algorithme précédent, $\mathbf{V}_{ij}$ représente le j-ème élément de la ligne $\mathbf{v}_i$. Quand on dit que l'on échange $\mathbf{v}_i$ et $\mathbf{v}_j$, on les échanges comme lignes de la matrice. En code, cela veut dire que l'on échange leurs références. Dans la suite de l'algorithme, quand on fait référence à $\mathbf{v}_i$ ($\mathbf{v}_j$), on fait référence à ce qui était anciennement $\mathbf{v}_j$ (ce qui était anciennement $\mathbf{v}_i$).*

Cet algorithme consiste à appliquer l'algorithme d'Euclide sur le i-ème élément de chaque vecteur $\mathbf{v}_i, \ldots, \mathbf{v}_l$ pour $1 \leq i \leq t$ croissant. On applique cependant les opérations sur le vecteur complet. À la fin de la procédure on retire des lignes de la matrice, mais ces lignes sont nulles.

2.2.1. Obtenir la base duale

On s'intéresse maintenant à trouver une matrice de base $\mathbf{W}$ du dual $L_t^*(I)$ de $L_t(I)$. Cette matrice n'est généralement pas entière avec le choix de $L_t(I)$ entier que l'on a fait précédemment. Par contre, $m\mathbf{W}$ est toujours entière. On manipulera donc plutôt le m-dual de $L_t(I)$.

Grâce à la proposition (2.1.6), il est facile de déterminer une forme explicite pour une base du dual : on sait qu'il existe une base $\mathbf{W}^* = m\mathbf{W}$ avec $\mathbf{V}(\mathbf{W}^*)^t = m\mathbf{I}$. Il est possible de résoudre ce système en utilisant une modification de l'algorithme de Gauss-Jordan, les propriétés d'existence garantissant qu'il se terminera. Par contre, l'algorithme qui suit est plus intéressant puisqu'il est plus efficace.

La matrice $\mathbf{V}$ résultant de l'algorithme (2.2.3) est une matrice triangulaire supérieure, on sait donc que $\mathbf{W}^*$ est triangulaire inférieure. Si on réécrit explicitement$\mathbf{V}(\mathbf{W}^*)^t = m\mathbf{I}$

$$\begin{bmatrix} \mathbf{V}_{11} & \mathbf{V}_{12} & \cdots & \mathbf{V}_{1t} \\ 0 & \mathbf{V}_{22} & \cdots & \mathbf{V}_{2t} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{V}_{tt} \end{bmatrix} \begin{bmatrix} \mathbf{W}_{11} & \mathbf{W}_{21} & \cdots & \mathbf{W}_{t1} \\ 0 & \mathbf{W}_{22} & \cdots & \mathbf{W}_{t2} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{W}_{tt} \end{bmatrix} = m\mathbf{I},$$

on obtient un forme explicite pour les coefficients de $\mathbf{W}^*$:

$$\mathbf{W}_{ii}\mathbf{V}_{ii} = m \Rightarrow \mathbf{W}_{ii} = m/\mathbf{V}_{ii}, \quad 1 \leq i \leq t$$

$$\sum_{k=j}^{i} \mathbf{V}_{jk}\mathbf{W}_{ik} = 0 \Rightarrow \mathbf{W}_{ij} = \frac{-1}{\mathbf{V}_{jj}} \sum_{k=j+1}^{i} \mathbf{V}_{jk}\mathbf{W}_{ik}, \quad 1 \leq j < i \leq t.$$

Tout comme pour L_t, le réseau pour les indices séquentiels $I = \{0, \ldots, t-1\}$, il est possible de donner la forme explicite des vecteurs de L_t^* **[32]**.

Proposition 2.2.5 (Base dual d'un GCM)**.** *Les vecteurs*

$$\begin{aligned}
\mathbf{w}_1 &= (m, 0, \ldots, 0, 0, \ldots, 0)^t \\
&\vdots \\
\mathbf{w}_k &= (0, 0, \ldots, m, 0, \ldots, 0)^t \\
\mathbf{w}_{k+1} &= (-x_{1,k}, -x_{2,k}, \ldots, -x_{k,k}, 1, \ldots, 0)^t \\
&\vdots \\
\mathbf{w}_t &= (-x_{1,t-1}, -x_{2,t-1}, \ldots, -x_{k,t-1}, 0, \ldots, 1)^t
\end{aligned}$$

forment une base de mL_t^* *pour* $I = \{0, \ldots, t-1\}$.

2.2.2. Un mot sur les autres types de GLMm

Ce que l'on vient de présenter permet de construire un base pour le primal et le dual des GCM et des GCL. Dans le cas des GMR et GCM combinés, on contourne en quelque sorte le problème de leur construire un réseau explicitement : on construira d'abord le GCM qui leur est équivalent et on étudiera le réseau de ce GCM.

Pour ce qui est des GCLM, on peut en fait utiliser les même bases explicites que dans les propositions 2.2.2 et 2.2.5 si on utilise la suite $\{x_n\}$ définie par l'équation (1.1.9) pour construire l'ensemble Ψ_t.

Pour les GCL avec $c \neq 0$, il faut un peu plus de justification, mais la construire est tout aussi simple.

Dans ce cas, il s'avère que l'étude du réseau généré par ce générateur est identique à l'étude d'un GCL avec le même m et le même a, mais $c = 0$ **[15]**. Les vecteurs $\mathbf{v} = (x_0, \ldots, x_{t-1})^t$ du réseau généré par le générateur se trouvent à avoir la forme explicite suivante en fonction de x_0 :

$$\mathbf{v} = \left(x_0, ax_0 + c, \ldots, a^{t-1}x_0 + \frac{(a^{t-1}-1)c}{a-1}\right)^t \bmod m.$$

Il est possible de prendre $x_0 = 0$, ce qui veut dire que le vecteur $\hat{\mathbf{v}} = \left(0, c, \ldots, \frac{(a^{t-1}-1)c}{a-1}\right)^t \bmod m$ est également dans le réseau. Il suit que $(x_0, ax_0, \ldots, a^{t-1}x_0) =$

$\mathbf{v} - \hat{\mathbf{v}}$ est dans le réseau pour tout $x_0 \in \mathbb{Z}_m$. Cela veut dire que la même base de réseau est admissible pour ce générateur que pour un GCL avec les mêmes a et m, mais $c = 0$.

2.3. Étudier l'uniformité d'un GLMm avec les réseaux

Dans cette section, on présente de quelle manière la structure de réseau est importante pour étudier l'uniformité des générateurs linéaires modulo m. On présente la notion d'empilement compact et son utilisation pour mesurer l'uniformité d'un réseau. Ensuite, on explique comment utiliser plusieurs de ces mesures pour évaluer un réseau.

2.3.1. Empilement compact

Puisque l'objectif de ce mémoire n'est pas d'étudier ce problème en soi, on le présente de manière informelle. Un empilement compact est une répétition périodique de n-sphères (de rayon identique et sans superposition) dans l'espace à $n + 1$ dimensions. Un problème récurrent avec les empilements compacts est de trouver un empilement compact tel que la proportion de l'espace qu'il couvre est optimale (ou, de manière équivalente, tel que la proportion de vide entre les sphère est minimisée) [**4**].

Il est possible de construire un empilement compact à partir d'un réseau. Spécifiquement, écrivons $\lambda(L)$ le plus court vecteur non-nul d'un réseau. En positionnant une sphère de rayon $|\lambda(L)|/2$ à chaque point d'un réseau L, on obtient un empilement compact.

On introduit quelques notations et propriétés géométriques.

- On note Δ_t la densité d'un empilement compact en dimension t. Il s'agit de la proportion de l'espace couvert par les sphères. Par extension, quand l'empilement compact est obtenu à partir d'un réseau, on appelle Δ_t la densité de ce réseau. On note Δ_t^* une borne supérieure sur la plus grand densité qu'il est possible d'obtenir en t dimensions.
- On note le volume d'une sphère de rayon 1 de dimension t comme $V_t = \frac{\pi^{t/2}}{\Gamma(1+t/2)}$. On sait que le volume d'une sphère de dimension t et de rayon ρ est $V_t\rho^t$.
- On définit le volume d'un réseau $\text{vol}(L) = \det(\mathbf{V})$. Cette mesure ne dépend pas du choix de base. Cette mesure correspond au volume du plus petit parallélépipède dont les sommets sont des points du réseau. Il est possible d'associer une de ces figures à chaque point, il s'agit donc du volume par point du réseau.

- Le nombre de points par unité de volume dans le réseau est $n = \frac{1}{\det(L_t)}$.

Une partie de la littérature sur les empilements compacts s'intéresse à construire des bornes Δ_t^*. On peut comparer la densité de l'empilement compact obtenu d'un réseau avec ces bornes pour avoir une idée de l'uniformité qu'il possède. Pour un nombre de points donné, un réseau plus dense est plus uniforme. Dans cette situation, pour augmenter la densité, il faut centrer des sphères plus grandes autour des points. Cela veut dire que les points sont plus éloignés les uns des autres et maximisent l'utilisation de l'espace.

Avec toutes ces définitions on peut maintenant calculer la densité de l'empilement compact associé à un réseau. On place une sphère de volume $V_t \frac{\lambda(L)^t}{2^t}$ à chaque point du réseau. Le volume par point dans le réseau est $\mathrm{vol}(L)$. Il suit donc que

$$\Delta_t = \frac{V_t \lambda(L)^t}{2^t \det(\mathbf{V})}.$$

Il est possible d'obtenir une mesure de l'uniformité de ce réseau en prenant le ratio Δ_t/Δ_t^*. En pratique, dans notre logiciel, on n'utilise pas ce ratio directement : on construit plutôt des bornes $\lambda^*(L)$ sur la valeur de $\lambda(L)$ et on calcule le ratio $\lambda(L)/\lambda^*(L)$.

En connaissant des valeurs de Δ_t^*, on obtient la borne $\lambda^*(L)$ suivante

$$\lambda(L) = 2\left(\frac{\Delta_t \det(\mathbf{V})}{V_t}\right)^{1/t} \leq 2\left(\frac{\Delta_t^* \det(\mathbf{V})}{V_t}\right)^{1/t}. \tag{2.3.1}$$

Dans cette équation, le seul terme qui ne dépend pas de la dimension est $\det(\mathbf{V})$. On définit $\gamma_t = 4(\Delta_t^*/V_t)^{2/t}$, que l'on appelle constante d'Hermite [**4**]. On utilise

$$\lambda(L) \leq \gamma_t^{1/2} \det(L_t)^{1/t} = \lambda^*(L) \tag{2.3.2}$$

comme borne [**22**].

L'utilisation de la longueur de $\lambda(L)$ pour mesurer l'uniformité est due à l'interprétation géométrique de cette longueur pour le dual. Il s'avère que l'inverse de la longueur du plus court vecteur dans le dual, $1/\lambda(L_t^*)$, correspond à la distance entre les hyperplans les plus éloignés entre lesquels il n'y a pas de points du réseau. Avec un bon générateur, on veut que cette distance soit petite, autrement l'ensemble des points qu'il génère contient de grandes sections de « vide ».

On note le ratio $M(L) = \lambda(L)/\lambda^*(L)$. Son utilisation avec la longueur du plus court vecteur dans le dual pour mesurer la performance d'un générateur est appelé le test spectral. Il s'agit du test le plus utilisé pour étudier l'uniformité des GLMm [**15, 32**].

2.3.2. Figures de mérite

On ne présente pas immédiatement comment il est possible d'obtenir les valeurs que l'on vient d'introduire, ce sera fait dans la section 2.4. On s'intéresse plutôt à classer les réseaux à partir de celles-ci. L'idée est simplement d'agglomérer beaucoup d'information sur un générateur afin d'obtenir un nombre résumant sa performance, que l'on appelle une figure de mérite. Si le regroupement est fait de manière pertinente, alors le nombre que l'on obtient donne une idée générale de la qualité du générateur et peut même permettre de classer divers générateurs entre eux.

Un bon GNA est réparti uniformément pour tout ensemble et sous-ensemble de points qu'il peut générer. Un générateur avec une seule mauvaise projection peut faire échouer une expérience en utilisant cette projection par inadvertance [**11**]. Dans le cas des GLMm, on peut tester ceci indirectement : on veut qu'ils soient répartis uniformément sur le plus grand nombre de projections possible [**15, 23**].

En calculant le ratio $M(L)$, on obtient une mesure entre 0 et 1 de la performance relative du générateur pour telle ou telle projection. Ces nombres peuvent significativement être comparés entre eux. Puisque l'on veut que le générateur se comporte bien partout, on détermine la qualité des GLMm en fonction de leur rendement dans leur pire projection. Prenons $d \in \mathbb{N}$ et $t_1, \ldots, t_d \in \mathbb{N}$. On définit le mérite au test spectral d'un générateur comme

$$M_{t_1,\ldots,t_d} = \min\left\{\min_{t\leq t_1} M(L_t); \min_{2\leq o\leq d}\min_{0\leq i_1<\cdots<i_o\leq t_o} M(L_o(\{i_1,\ldots,i_o\}))\right\}, \tag{2.3.3}$$

ou encore

$$M_{\mathcal{I}} = \min_{I\in\mathcal{I}} M(L_{|I|}(I)), \tag{2.3.4}$$

avec $\mathcal{I} = \{\{0,\ldots,t\} : 1 \leq t \leq t_1\} \cup \{I : |I| = 2; I \subseteq \{0,\ldots,t_2\}\} \cup \cdots \cup \{I : |I| = d; I \subseteq \{0,\ldots,t_d\}\}$. La mesure $M_{t_1,\ldots,t_d}$ teste la structure de réseau pour tous les sous-ensembles de cardinalité o de $\{0,\ldots,t_o\}$. Elle n'est pas indicative du fait que les points du GNA auront l'air indépendants, mais un générateur avec une mauvaise performance à ce test possède une mauvaise uniformité et devrait donc être évité.

Remarque 2.3.1. *Pour un générateur de pleine période, il est possible d'éviter de tester certaines de ces projections. Cela est dû au fait que le test spectral examine l'union de tous les sous-cycles, parfois disjoints, de la période du générateur. Un générateur de pleine période possède un seul cycle avec quelques propriétés supplémentaires. Dans ce cas, on peut exiger*

de toujours avoir $i_1 = 0$ *puisque la pleine période implique que* $\Psi_{\{i_1+t,\ldots,i_o+t\}} = \Psi_{\{i_1,\ldots,i_o\}}$. *Sans la pleine période, il est possible que ce ne soit pas le cas, par exemple si* $i_1 \geq k$. *On peut aussi exiger que* $\max\{i_2,\ldots,i_o\} \geq k$, *sinon le réseau sera engendré par la matrice identité : tous les vecteurs contenant moins de* k *composantes sont engendrés par les projections sur* $\{0,\ldots,k-1\}$.

On aimerait que les générateurs aient une bonne performance à ce test pour d et $t_1,\ldots,t_d$ les plus grands possible. Un nombre de projections trop important n'est cependant pas pratique. Il peut être trop long de calculer la figure de mérite et le résultat du test sera presque assurément mauvais pour l'une d'elles. Pour contourner ces défauts, on calcule la figure de mérite plusieurs fois. D'abord sur moins de projections que l'on juge plus importantes, et ensuite sur un grand nombre, mais seulement pour les générateurs ayant eu de bons résultats au premier test [**22**].

2.3.3. Une autre mesure

Il est possible de recouvrir un réseau par plusieurs familles d'hyperplans. Considérons la famille de ces hyperplans les plus éloignés les uns des autres. Le nombre de ces hyperplans couvrant les points du réseau dans le cube [0,1) peut également être utilisé comme une mesure de l'uniformité du réseau : plus il faut d'hyperplans pour couvrir les points, plus l'espace est couvert uniformément.

Il est possible de déterminer le nombre minimal d'hyperplans couvrant le réseau d'une manière assez semblable à la distance entre les hyperplans les plus éloignés du réseau. Il suffit de calculer la longueur du plus court vecteur dans le dual $\mathbf{h}$ en utilisant la norme L1 ($\|(x_1,\ldots,x_t)\|_1 = |x_1|+\cdots+|x_t|$) plutôt que la norme euclidienne. On a alors que le nombre minimal d'hyperplans couvrant les points du réseau est $\|\mathbf{h}\|_1 - 1$ [**9**].

Il est possible de construire des figures de mérite utilisant cette mesure de la même manière que pour la norme euclidienne. Il s'avère que $\|\mathbf{h}\|_1 - 1 \leq (t!\Delta_t)^{1/t}$, en prenant Δ_t la densité du réseau primal [**44**]. Cette borne n'est pas particulièrement serrée, mais peut tout de même faciliter la comparaison entre des générateurs de périodes différentes. Cette mesure a aussi historiquement été utilisée pour démontrer que certains générateurs avaient un comportement particulièrement indésirable pour certaines projections [**5**].

2.4. Trouver le plus court vecteur

Tel qu'il a été dit dans la section précédente, pour avoir une mesure normalisée entre 0 et 1 permettant d'évaluer l'uniformité d'un réseau, il faut être en mesure de calculer la longueur du plus court vecteur non-nul dans un réseau. Si on prend $\mathbf{v}_1, \ldots, \mathbf{v}_t$, une base du réseau, on peut formuler ce problème comme un problème d'optimisation avec l'objectif :

$$\min_{\mathbf{0} \neq \mathbf{v} \in L_t} \|\mathbf{v}\|^2, \tag{2.4.1}$$

sujet aux conditions

$$\begin{aligned} \mathbf{v} &= z_1\mathbf{v}_1 + \cdots + z_t\mathbf{v}_t, \\ z_1, \ldots, z_t &\in \mathbb{Z}, \\ |z_1| + \cdots + |z_t| &\neq 0. \end{aligned} \tag{2.4.2}$$

Ce problème est un problème d'optimisation en nombre entiers à objectif quadratique. On présentera plus loin comment le résoudre avec un algorithme de séparation et évaluation (S&E). Résoudre ce genre de problème prend, dans le pire cas, un temps croissant exponentiellement avec t **[15]**.

On utilise deux stratégies afin de résoudre ce problème pour des dimensions relativement grandes. D'abord, on réduit préalablement la taille des vecteurs du réseau si cela est possible. Cela permet d'obtenir des bornes sur $\|\mathbf{v}\|$, réduisant l'espace de recherche. On présente 3 méthodes accomplissant cela. Ensuite, l'algorithme de S&E présenté plus bas utilise une méthode de coupes. Cela permet de réduire l'espace de recherche pour les $z_1, \ldots, z_t$.

Comme décrit dans la section 2.3.3, il peut aussi être intéressant de modifier l'objectif (2.4.1) par

$$\min_{\mathbf{0} \neq \mathbf{v} \in L_t} \|\mathbf{v}\|_1. \tag{2.4.3}$$

Dans ce cas, la méthode de coupe utilisée avec la norme euclidienne ne peut pas être utilisée. La toute fin du chapitre décrit comment adapter l'algorithme de S&E à ce cas et explique comment obtenir des approximations de la solution à (2.4.3).

2.4.1. Méthodes de réduction de réseau

Les algorithmes que l'on présente maintenant servent à réduire un réseau. Ici, le terme *réduire* n'a pas de signification exacte. Chacun des algorithmes que présenté modifie la base d'une manière différente avec des implications différentes. Les algorithmes de réduction

doivent assister à la recherche du plus court vecteur d'un réseau. Ils seront donc utilisés avec l'objectif d'obtenir une base dont les vecteurs ont une norme plus courte qu'avant la réduction.

On commence par présenter la réduction par paire. Il s'agit d'une méthode de réduction de base plutôt basée sur une heuristique que sur un résultat fort. L'idée en est d'adapter l'orthogonalisation de Gram-Schmidt aux réseaux. On note $\hat{\mathbf{v}}_1, \ldots, \hat{\mathbf{v}}_t$, les vecteurs orthogonaux obtenus comme

$$\begin{aligned}
\hat{\mathbf{v}}_1 &= \mathbf{v}_1; \\
\hat{\mathbf{v}}_i &= \mathbf{v}_i - \sum_{j<i} \mu_{ij} \hat{\mathbf{v}}_j, \quad 1 < i \leq t; \\
\mu_{ij} &= \frac{\mathbf{v}_i \cdot \hat{\mathbf{v}}_j}{\hat{\mathbf{v}}_j \cdot \hat{\mathbf{v}}_j}, \quad 1 \leq j < i \leq t.
\end{aligned}$$

Puisqu'en général les μ_{ij} ne sont pas entiers, les vecteurs $\hat{\mathbf{v}}_i$ ne sont pas dans le réseau. Par contre il suffit d'arrondir les μ_{ij} pour obtenir une base presque orthogonale de celui-ci. L'idée d'appliquer cette réduction du réseau avant de chercher le plus court vecteur dans la base a été proposée par Dieter en 1975 [**9**]. L'algorithme considéré dans le cadre de ce mémoire est le suivant.

Algorithme 2.4.1. *Soit une base de réseau* $\mathbf{v}_1, \ldots, \mathbf{v}_2$ *et une base du m-dual* $\mathbf{w}_1, \ldots, \mathbf{w}_t$ *avec* $\mathbf{V}\mathbf{W}^t = m\mathbf{I}$ *et l'algorithme :*

RedPaires($\mathbf{v}_1$, …, $\mathbf{v}_t$, $\mathbf{w}_1$, …, $\mathbf{w}_t$):

$a = 0$, $k = 1$

 # a : le nombre d'étapes successives sans changement

 # k : l'indice du vecteur que l'on veut modifier

tant que $a < 2t$**:**

temp = 0

pour $1 \leq j \leq t$, $j \neq k$**:**

$q = \lfloor 1/2 + \frac{\mathbf{v}_j \cdot \mathbf{v}_k}{\mathbf{v}_k \cdot \mathbf{v}_k} \rfloor$

$\mathbf{v}_j = \mathbf{v}_j - q\mathbf{v}_k$

$\mathbf{w}_k = \mathbf{w}_k + q\mathbf{w}_j$

$temp = temp + |q|$

si $temp = 0$**:**

$a = a + 1$

sinon:

$a = 0$

$k = k + 1$

si $k > t$**:**

$k = 1$

pour $1 \leq j \leq t$**:**

échanger $\mathbf{w}_j$ *et* $\mathbf{v}_j$

Les bases résultantes de **RedPaires**, $\mathbf{v}_1, \ldots, \mathbf{v}_t$ *et* $\mathbf{w}_1, \ldots, \mathbf{w}_t$, *sont des bases du réseau et de son m-dual avec* $\mathbf{V}\mathbf{W}^t = m\mathbf{I}$ *(voir remarque 2.4.2).*

Remarque 2.4.2. *L'algorithme tel qu'il est écrit ne préserve pas la base et le dual dans* $\mathbf{V}$ *et* $\mathbf{W}$ *respectivement. Cela a été fait afin d'alléger l'écriture. À la fin de l'algorithme, selon le nombre d'itérations, on ne sais pas ce que* $\mathbf{V}$ *contient. En pratique il faut être capable de déterminer quelle matrice est la base primale et quelle matrice est la base duale.*

Étant donné que cet algorithme manipule le dual et le primal, il n'y a pas de garantie que les normes $\|\mathbf{v}_i\|$ seront réduites. Aussi, il est utile de mettre une clause permettant à l'algorithme d'arrêter la boucle **tant que** après un certain nombre d'itérations. Aucun résultat ne permet de s'assurer que la norme des vecteur de la base obtenue est réduite. Sa seule garantie est que, à chaque étape de la boucle **tant que**, les normes $\|\mathbf{v}_i\|$ (avant d'échanger les vecteurs) sont plus petites ou égales à avant l'étape.

La valeur de $\|\mathbf{v}_i - q\mathbf{v}_k\|$ explose si q grandit, il existe donc une valeur de q minimisant cette norme. Cette valeur respecte les deux inégalités

$$\|\mathbf{v}_i - q\mathbf{v}_k\|^2 \leq \|\mathbf{v}_i - (q+1)\mathbf{v}_k\|^2 \quad \text{et} \quad \|\mathbf{v}_i - q\mathbf{v}_k\|^2 \leq \|\mathbf{v}_i - (q-1)\mathbf{v}_k\|^2.$$

En développant, on obtient

$$\begin{aligned} -\mathbf{v}_k \cdot \mathbf{v}_k &\leq 2(\mathbf{v}_i - q\mathbf{v}_k) \cdot \mathbf{v}_k \leq \mathbf{v}_k \cdot \mathbf{v}_k \\ \Rightarrow -\frac{1}{2} + \frac{\mathbf{v}_i \cdot \mathbf{v}_j}{\mathbf{v}_j \cdot \mathbf{v}_j} &\leq q \leq \frac{1}{2} + \frac{\mathbf{v}_i \cdot \mathbf{v}_j}{\mathbf{v}_j \cdot \mathbf{v}_j}, \end{aligned}$$

ce qui justifie notre choix pour q.

On peut maintenant s'intéresser aux autres algorithmes implémentés afin de réduire les réseaux. Les deux réductions qu'il reste à présenter sont relativement similaires, il s'agit de la réduction de Lenstra-Lenstra-Lovàsz (LLL) [**42**] et de la réduction de Korkine-Zolotarev par

blocs [**51**]. Contrairement à la réduction précédente, ces deux réductions ont une fondation théorique solide couverte avant de présenter les algorithmes.

Définition 2.4.3 (Réduction LLL)**.** *Soit une base de réseau* $\mathbf{v}_1, \ldots, \mathbf{v}_t$ *avec* $\|\mathbf{v}_i\| \leq \|\mathbf{v}_{i+1}\|$ *pour* $1 \leq i < t$*. On définit* $\hat{\mathbf{v}}_j$ *et* $\mu_{ij} = \frac{\mathbf{v}_i \cdot \hat{\mathbf{v}}_j}{\hat{\mathbf{v}}_j \cdot \hat{\mathbf{v}}_j}$ *pour* $1 \leq j \leq i \leq t$ *comme pour l'orthogonalisation de Gram-Schmidt. On définit aussi par récurrence* $\mathbf{v}_i(j) = \mathbf{v}_i - \sum_{k=1}^{j-1} \mu_{ik}\mathbf{v}_k(k)$*, la composante de* $\mathbf{v}_i$ *orthogonale à* $\mathbf{v}_1, \ldots, \mathbf{v}_{j-1}$*, avec* $\mathbf{v}_i(i) = \hat{\mathbf{v}}_i$ *pour* $1 \leq i \leq j$*.*

Soit $1/4 < \delta \leq 1$*, on dit que* $\mathbf{v}_1, \ldots, \mathbf{v}_t$ *est LLL-réduite avec un facteur* δ *si*

(1) $|\mu_{ij}| \leq 1/2$*, pour* $1 \leq j < i \leq t$*,*

(2) $\delta\|\mathbf{v}_i(i)\|^2 \leq \|\mathbf{v}_{i+1}(i)\|^2$*, pour* $1 \leq i < t$*.*

Nous avons le résultat suivant sur les bases LLL-réduites avec un facteur δ.

Proposition 2.4.4 ([**42**])**.** *Une base LLL-réduite avec un facteur* δ *satisfait*

$$\left(\frac{1}{\delta - 1/4}\right)^{1-i} \leq \left(\frac{\|\mathbf{v}_i\|}{\lambda_i}\right)^2 \leq \left(\frac{1}{\delta - 1/4}\right)^{t-1}, \quad 1 \leq i \leq t, \tag{2.4.4}$$

où λ_i *est la longueur du* i*-ème plus court vecteur de* L_t*.*

Ce résultat nous donne des bornes relativement serrées sur le ratio entre $\|\mathbf{v}_i\|$ et λ_i. Plus δ sera près de 1, plus ces bornes seront serrées. Cela veut aussi dire que le plus court vecteur de la base, après qu'elle soit LLL-réduite avec un facteur δ, est relativement près du plus court vecteur non-nul dans le réseau. Ceci justifie l'usage de la réduction LLL afin de faciliter la recherche du plus court vecteur dans le réseau.

Algorithme 2.4.5 (Réduction LLL [**51**])**.** *Soit* $\mathbf{v}_1, \ldots, \mathbf{v}_t$ *une base d'un réseau et* $1/4 < \delta \leq 1$ *et l'algorithme :*

LLL($\mathbf{v}_1$, ..., $\mathbf{v}_t$, δ):

$k = 2$

calculer μ_{ij}*,* $\hat{\mathbf{v}}_i$ *(Gram-Schmidt) pour tout* $1 \leq j < i \leq t$*.*

tant que $k \leq m$**:**

pour $k > j \geq 1$**:**

si $|\mu_{kj}| > 1/2$**:**

$\mathbf{v}_k = \mathbf{v}_k - \lfloor 1/2 + \mu_{kj} \rfloor \mathbf{v}_j$*:*

pour $1 \leq i \leq m$**:**

$\mu_{ki} = \mu_{ki} - \lfloor 1/2 + \mu_{kj} \rfloor \mu_{ji}$

si $\delta\|\mathbf{v}_{k-1}(k-1)\|^2 > \|\mathbf{v}_k(k-1)\|^2$**:**

échanger $\mathbf{v}_k$ *et* $\mathbf{v}_{k-1}$

$k = \max(k-1,2)$

sinon :

$k = k+1$

Après l'application de **LLL**, *la base* $\mathbf{v}_1, \ldots, \mathbf{v}_t$ *est LLL-réduite avec un facteur* δ.

Instinctivement, cet algorithme est une manière ordonnée de d'appliquer la réduction par paires. On ajoute la propriété (2) de la définition 2.4.3 afin d'obtenir des résultats plus forts. En pratique, on utilise une version légèrement modifiée de l'algorithme pour prendre en compte la précision finie que l'on a pour représenter μ_{ij} et $\|\mathbf{v}_i\|$.

On s'intéresse maintenant à la réduction Korkine-Zolotarev par blocs (BKZ). On introduit d'abord les différentes définitions.

Définition 2.4.6 (Réduction BKZ [**50**])**.** *Notons* $R_j(i) = \left\{\sum_{k=j}^{t} z_k \mathbf{v}_k(i) : z_j \in \mathbb{Z}\right\}$. *On dit que* $\mathbf{v}_1, \ldots, \mathbf{v}_t$ *est KZ-réduit si* $|\mu_{ij}| \leq 1/2$ *pour* $1 \leq j < i \leq t$ *et* $\|\mathbf{v}_i(i)\| \leq \lambda(R_i(i))$ *pour* $1 \leq i \leq t$.

On dit que $\mathbf{v}_1, \ldots, \mathbf{v}_n$ *est BKZ-réduit avec des blocs de taille* k *si*

(1) $|\mu_{ij}| \leq 1/2$, *pour* $1 \leq j < i \leq t$,

(2) $\|\mathbf{v}_i(i)\| \leq \lambda(R_i(i) - R_{\min(i+k-1,t)}(i))$, *pour* $1 \leq i < n-k$.

Soit $1/4 < \delta \leq 1$, *on dit que* $\mathbf{v}_1, \ldots, \mathbf{v}_n$ *est BKZ-réduit avec un facteur* δ *et des blocs de taille* k *si*

(1) $|\mu_{ij}| \leq 1/2$, *pour* $1 \leq j < i \leq t$,

(2) $\delta\|\mathbf{v}_i(i)\| \leq \lambda(R_i(i) - R_{\min(i+k-1,t)}(i))$, *pour* $1 \leq i < n-k$.

Remarque 2.4.7. *La condition* $\|\mathbf{v}_i(i)\| \leq \lambda(R_i(i) - R_{\min(i+k-1,t)}(i))$ *pour* $1 \leq i < n-k$ *est équivalente à dire que* $\mathbf{v}_i(i), \ldots, \mathbf{v}_{i+k-1}(i)$ *est KZ-réduit.*

Proposition 2.4.8. *Une base KZ-réduite satisfait*

$$\frac{4}{i+3} \leq \left(\frac{\|\mathbf{v}_i\|}{\lambda_i}\right)^2 \leq \frac{i+3}{4}, \quad 1 \leq i \leq t. \tag{2.4.5}$$

Une base BKZ-réduite avec des blocs de taille k *et un facteur* δ *satisfait*

$$\|\mathbf{v}_1\|^2 \leq \alpha_k^{(t-1)/(k-1)} \lambda(L_t)^2, \tag{2.4.6}$$

si $k-1$ *divise* $t-1$, *en prenant* $\alpha_k = \max \frac{\|\mathbf{v}_1\|^2}{\|\mathbf{v}_k(k)\|^2}$ *avec le maximum sur tous les ensembles* $\mathbf{v}_1, \ldots, \mathbf{v}_k$ *KZ-réduits* [**50**].

La réduction BKZ est une généralisation de la réduction LLL. Plutôt que d'imposer une restriction sur $\|\mathbf{v}_i(i)\|^2$ par rapport au seul vecteur $\mathbf{v}_{i+1}(i)$, on généralise aux $k-1$ vecteurs $\mathbf{v}_{i+1}(i), \ldots, \mathbf{v}_{i+k-1}(i)$. Par contre, si $k = 2$ et $\delta > 1/3$, les deux réductions sont équivalentes. La force de la réduction BKZ dépend beaucoup de la valeur de α_k, qui n'est pas connue exactement. Par contre, même avec $\delta = 1$ (la valeur de δ donnant les meilleures bornes), LLL a la borne supérieure $\left(\frac{4}{3}\right)^{(t-1)} \lambda(L_t)^2$ sur $\|\mathbf{v}_1\|^2$. Cette borne est moins forte que celle de BKZ puisque $\lim_{k\to\infty} \alpha_k^{1/(k-1)} = 1$ **[50]**, ce qui veut dire que la borne rapetisse lorsque k augmente. On peut trouver le pseudo-code pour la réduction BKZ avec un facteur δ dans **[51]**.

2.4.2. Algorithme de S&E

Avant de présenter notre algorithme de S&E pour trouver le plus court vecteur non-nul d'un réseau, on présente et justifie la méthode de coupe, basée sur **[12]**, que nous utilisons afin de réduire l'espace de recherche sur le z_j dans le problème à l'équation (2.4.1).

La matrice des produits scalaires $\mathbf{V}\mathbf{V}^t$ est toujours définie positive. De ce fait, il est possible d'en faire une décomposition de Cholesky $\mathbf{V}\mathbf{V}^t = \mathbf{U}^t\mathbf{U}$ avec $\mathbf{U}$ triangulaire supérieure

$$\mathbf{U} = \begin{bmatrix} u_{11} & \cdots & u_{1k} \\ \vdots & \ddots & \vdots \\ 0 & \cdots & u_{kk} \end{bmatrix}.$$

Prenons $1 \leq j \leq t$. Alors, tout candidat de plus court vecteur $\mathbf{v}^* = \mathbf{V}^t\mathbf{z}^*$ avec $\mathbf{z}^* = (z_1, \ldots, z_j, z_{j+1}^*, \ldots, z_t^*)^t$, où $z_{j+1}^*, \ldots, z_t^*$ sont fixés, est tel que

$$\begin{aligned}
(\mathbf{v}^*)^t\mathbf{v}^* &= (\mathbf{z}^*)^t\mathbf{V}\mathbf{V}^t\mathbf{z}^* = (\mathbf{z}^*)^t\mathbf{U}\mathbf{U}^t\mathbf{z}^* \\
&= \sum_{k=1}^{j-1}\left(\sum_{l=k}^{j} u_{kl}z_l + \sum_{l=j+1}^{t} u_{kl}z_l^*\right)^2 + \sum_{k=j}^{t}\left(\sum_{l=k}^{j} u_{kl}z_l + \sum_{l=j+1}^{t} u_{kl}z_l^*\right)^2 \\
&\geq \sum_{k=j}^{t}\left(\sum_{l=k}^{j} u_{kl}z_l + \sum_{l=j+1}^{t} u_{kl}z_l^*\right)^2 \\
&\geq \left(u_{jj}z_j + \sum_{l=j+1}^{t} u_{jl}z_l^*\right)^2 + \sum_{k=j+1}^{t}\left(\sum_{l=k}^{t} u_{kl}z_l^*\right)^2.
\end{aligned}$$

On définit $r_k = \sum_{l=j+1}^{t} u_{kl} z_l^*$ et $s_k = \sum_{l=k+1}^{t} (\sum_{m=l}^{t} u_{lm} z_m^*)^2$, ce qui permet de réécrire

$$(\mathbf{v}^*)^t \mathbf{v}^* \geq (u_{jj} z_j + r_j)^2 + s_j.$$

Finalement, si le candidat $\mathbf{v}^*$ a une norme plus courte que le plus court vecteur connu $\mathbf{v}_1$ dans le réseau, alors

$$(u_{jj} z_j + r_j)^2 + s_j \leq (\mathbf{v}^*)^t \mathbf{v}^* < \mathbf{v}_1^t \mathbf{v}_1.$$

On réécrit l'inégalité pour borner z_j

$$\left\lceil \frac{-(\mathbf{v}_1^t \mathbf{v}_1 - s_j)^{1/2} - r_j}{u_{jj}} \right\rceil \leq z_j \leq \left\lfloor \frac{(\mathbf{v}_1^t \mathbf{v}_1 - s_j)^{1/2} - r_j}{u_{jj}} \right\rfloor. \tag{2.4.7}$$

Il s'avère donc qu'en fixant $z_{j+1}, \ldots, z_t$ il est possible d'avoir des bornes sur z_j. Dans le cadre d'un algorithmes de S&E, on construit ces bornes en fonction des variables qui ont déjà été fixées.

On peut maintenant présenter un pseudo-code décrivant le fonctionnement de notre algorithme de S&E.

Algorithme 2.4.9. *Soit une base $\mathbf{v}_1, \ldots, \mathbf{v}_t$. On considère aussi $\mathbf{z} \in \mathbb{Z}^t$, un candidat de combinaison linéaire pour obtenir un plus court vecteur, et $\mathbf{z}^* \in \mathbb{Z}^t$ la combinaison linéaire permettant d'obtenir le plus court vecteur connu pour l'instant. On calcule la factorisation de Cholesky (u_{ij}) ainsi que r_j et s_j et on la garde en mémoire.*

SE($\mathbf{v}_1$, ..., $\mathbf{v}_t$, $\mathbf{z}$, $\mathbf{z}^*$, j):

si $j = 0$**:**

si $\mathbf{z} \neq 0$ **et** $\|z_1\mathbf{v}_1 + \cdots + z_t\mathbf{v}_t\| < \|z_1^*\mathbf{v}_1 + \cdots + z_t^*\mathbf{v}_t\|$**:**

$\mathbf{z}^* = \mathbf{z}$

retourne z

sinon:

retourne $\mathbf{z}^*$

$\mathbf{v}^* = z_1^*\mathbf{v}_1 + \cdots + z_t^*\mathbf{v}_t$

$min = \left\lceil \frac{-(\mathbf{v}^* \cdot \mathbf{v}^* - s_j)^{1/2} - r_j}{u_{jj}} \right\rceil$

$max = \left\lfloor \frac{(\mathbf{v}^* \cdot \mathbf{v}^* - s_j)^{1/2} - r_j}{u_{jj}} \right\rfloor$

si $max < min$*:*

retourne `faux`

si $j = t$*:*

$min = 0$

$up = \lceil (min + max)/2 \rceil$

$down = up - 1$

pour $up \leq max$ **ou** $down \geq min$:

$z_j = down$; $down = down - 1$

$\mathbf{z}^* = \mathbf{SE}(\mathbf{v}_1, \ldots, \mathbf{v}_t, \mathbf{z}, \mathbf{z}^*, j-1)$

$z_j = up$; $up = up + 1$

$\mathbf{z}^* = \mathbf{SE}(\mathbf{v}_1, \ldots, \mathbf{v}_t, \mathbf{z}, \mathbf{z}^*, j-1)$

retourne $\mathbf{z}^*$

On prend initialement $\mathbf{z}^* = (1, 0, \ldots, 0)^t$, $\mathbf{z} = \mathbf{0}$ *et* $j = t$. *Après l'application de* **SE**, *le vecteur* $\mathbf{v}^* = z_1^*\mathbf{v}_1 + \cdots + z_t\mathbf{v}_t$ *est le plus court vecteur non nul du réseau.*

Il s'agit d'un algorithme de S&E classique. De ce fait, en pire cas, son exécution prend un temps exponentiel par rapport à t. Puisque ce temps peut être très long, il est assez typique d'ajouter une close permettant d'arrêter l'algorithme après un certain nombre d'appels de la fonction **SE**. Il est aussi souvent utile d'appliquer un des algorithmes de réduction dont il a été question plus haut avant d'appeler **SE**.

La section 2.3.3 explique qu'il est possible de calculer des figures de mérite en calculant le plus court vecteur dans le dual avec la norme L1. Il n'est pas possible de simplement modifier l'objectif dans l'algorithme **SE** pour calculer ce vecteur. Les bornes sur les valeurs de z_j de l'équation (2.4.7) ne sont pas valides puisqu'elles assument que la norme de $\mathbf{v}$ est la norme euclidienne.

Si les lignes de $\mathbf{V}$ et $\mathbf{W}$ sont telles que $\mathbf{V}\mathbf{W}^t = \mathbf{I}$, alors

$$z_j = \sum_{1 \leq i \leq t} z_i \mathbf{v}_i \cdot \mathbf{w}_j.$$

Posons $(z_1, \ldots, z_t)$, une solution à l'équation (2.4.3), alors

$$\begin{aligned}
|z_j| &= \left| \sum_{1 \le i \le t} z_i \sum_{1 \le l \le t} v_{il} w_{jl} \right| \\
&= \left| \sum_{1 \le l \le t} \sum_{1 \le i \le t} z_i v_{il} w_{jl} \right| \\
&\le \sum_{1 \le l \le t} \left| w_{jl} \sum_{1 \le i \le t} z_i v_{il} \right| \\
&\le (\max_{1 \le i \le t} |w_{ij}|) \left\| \sum_{1 \le i \le t} z_i \mathbf{v}_i \right\|_1 .
\end{aligned}$$

On déduit donc que si $(z_1^*, \ldots, z_t^*)$ est une solution optimale au problème

$$|z_j^*| \le (\max_{1 \le i \le t} |v_{ij}|) \left\| \sum_{1 \le i \le t} z_i \mathbf{w}_i \right\|_1 \tag{2.4.8}$$

pour tout candidat de solution $(z_1, \ldots, z_t)$.

Pour modifier la procédure de S&E, il suffit de changer le calcul des bornes à l'endroit approprié. Il faut également donner la matrice de base pour le dual du réseau puisqu'elle est nécessaire pour le calcul des nouvelles bornes. Les bases que l'on soumet à l'algorithme doivent satisfaire $\mathbf{V}\mathbf{W}^t = \mathbf{I}$.

Il est à noter que les bornes obtenues dans ce cas sont beaucoup moins serrées qu'avec la norme euclidienne. La S&E ne peut être appliquée que sur des dimensions relativement plus petites. Dans les cas où il n'est plus possible de calculer le plus court vecteur en norme L1 directement, il est tout de même possible d'en faire une approximation. Typiquement, on estimera le plus court vecteur en norme L1 ($\mathbf{h}$) par le plus court vecteur en norme euclidienne ($\lambda(L)$). On a alors que $\|\lambda(L)\|_1 \ge \|\mathbf{h}\|_1$ et que les points du réseau se trouvent dans au plus $\|\lambda(L)\|_1$ hyperplans. Il est possible que ce nombre soit petit de toute façon et permette de montrer qu'un générateur a un mauvais comportement.

Chapitre 3

LatMRG, un logiciel pour étudier la structure de réseau

LatMRG est un logiciel et une librairie écrits en C++ permettant d'étudier la structure de réseau des GLMm. Ce logiciel possède deux utilités principales : la recherche de nouveaux GLMm ayant une bonne structure de réseau et l'évalutation de la qualité de la structure de réseau des GLMm utilisés en pratique. Il implémente toute la logique permettant de décrire les différents générateurs présentés au chapitre 1. De cela, on peut construire leur réseau comme décrit dans le chapitre 2 et calculer les figures de mérites décrites dans la section 2.3. LatMRG est en développement dans le Laboratoire de simulation et d'optimisation de l'Université de Montréal et est disponible en ligne à l'addresse https://github.com/savamarc/LatMRG.

Dans ce chapitre, on présente l'essentiel du contenu du logiciel : quelles structures peuvent être représentées, quels algorithmes sont implémentés et quelles fonctionnalités sont accessibles avec le logiciel exécutable. Dans la deuxième partie du chapitre, on décrit et justifie le déroulement du programme en détail.

3.1. Librairie LatMRG

LatMRG existe, sous une forme ou une autre, depuis plus de 20 ans. Dès 1997, L'Ecuyer et Couture ont publié un article décrivant l'essentiel des fonctions du logiciel [**32**]. Durant toutes ces années, les raisons d'être du logiciel n'ont pas changé : il reste aussi important d'avoir de bons générateurs de nombres aléatoires en simulation stochastique. Par contre, les fonctionnalités les plus importantes ont été modifiées pour permettre de travailler avec de nouveaux types de générateurs.

La version originale de LatMRG était écrite avec le langage Modula-2. Au cours des dernières années, le logiciel a été traduit en C++ puisque c'est un langage mieux supporté. C'est cette version traduite que ce mémoire améliore.

La version qui était disponible précédemment comportait plusieurs trous : l'intégralités des fonctions n'était pas encore disponible, la documentation incomplète et l'exécutable n'était pas en état de fonctionnement. C'est d'abord un travail de maintenance et de documentation qu'il a fallu faire. Les fonctionnalités suivantes, qui étaient historiquement présentes dans le logiciel, ont été vérifiées :

- la représentation des divers types de GLMm : GCL, GCM, GCLM, GCMC et GMR,
- la contruction de L_t et $L_t(I)$ pour différents générateurs,
- la recherche du plus court vecteur des réseaux L_t et $L_t(I)$,
- le calcul de figures de mérite sur les réseaux.

Dans LatMRG, les GLMm sont représentés par diverses classes distinctes en fonction du type de générateur que l'on veut représenter. Ces classes contiennent les paramètres du générateur, ainsi que la base de son réseau et du réseau dual pour une dimension donnée. Puisque chaque type de générateur peut être étudié comme un GCM, chacune des ces classes hérite de la classe permettant de représenter les GCM. Cela permet d'avoir une interface standard pour interagir avec tous les générateurs.

Il a fallu retravailler l'implémentation de la représentation des GMR, des GCMC et des GCLM. Les modules permettant de faire des calculs sur les GMR n'avaient pas été traduits. Ceux pour les GCMC l'étaient, mais n'utilisaient pas l'interface des GCM. Il a fallu écrire une classe au complet pour permettre au programme d'interagir avec ce type de générateur d'une manière standard. Pour les GCLM, la classe qui avait été implémentée a également dû être modifiée pour hériter des GCM.

La construction de L_t et $L_t(I)$ se fait à partir de la classe représentant le générateur. Chaque type de générateur implémente, avec ses particularités, l'interface de fonctions permettant de construire L_t. La construction de $L_t(I)$ se fait cependant différemment, elle ne dépend pas du type de GLMm, mais seulement de la base de L_t et de I. Cette construction a été uniformisée pour que tous les types de générateurs héritent de la même méthode permettant de facilement construire $L_t(I)$.

La réduction des réseaux et la recherche du plus court vecteur d'un réseau se fait par l'utilisation de LatticeTester (https://github.com/umontreal-simul/LatticeTester), une autre librairie du Laboratoire de simulation et d'optimisation. Cette librairie contient une classe permettant de faire tous ces calculs et les classes représentant les GLMm ont été modifiées pour optimiser l'utilisation de cette classe. Les anciennes implémentations modifiaient toujours à la fois la base du dual et la base du primal lors des calculs. Comme ce n'était pas nécessaire, le programme modifie maintenant seulement la base qui intéresse l'utilisateur.

Le calcul des figures de mérite a également fait l'objet d'une refonte. À l'échelle de la librairie, il fallait interagir avec une classe nécessitant tous les détails de la figure de mérite pour être utilisée. Cette classe était redondante du programme exécutable puisqu'elle n'ajoutait pas de fonctionnalité. Les différentes parties du calcul, réduction de réseau et calcul du mérite à partir d'un réseau réduit, ont été morcelées. Ces nouvelles fonctions simples s'appliquent séparément pour une seule projection à la fois. La librairie n'offre toujours rien permettant d'éviter de réécrire la logique pour calculer les figures de mérites qui ne sont pas déjà accessibles dans le programme exécutable. Cela représente cependant un avancement par rapport à la précédente version, il est plus simple d'utiliser les fonctions faisant la grosse part des calculs.

Finalement, la documentation de LatMRG a été révisée. Plusieurs méthodes et attributs de classes ayant une description sommaire ou non existante ont été détaillés. Le guide d'utilisation disponible en ligne avec le logiciel [**48**] a également été réécrit presque entièrement. Une description du nouveau guide est diponible à l'annexe B.

En plus d'améliorer les fonctions essentielles de LatMRG, ce mémoire en propose aussi d'autres qui n'étaient pas disponibles précédemment.

Cette version de LatMRG utilise NTL [**52**] comme librairie. Celle-ci permet de manipuler des entiers de précision arbitraire et implémente l'arithmétique sur les vecteurs et les matrices. De plus, elle implémente les réductions LLL et BKZ de manière optimisée sur des types qui lui sont spécifiques. Dorénavant, la transition vers l'utilisation de cette librairie est complétée. La précédente dépendance à Boost pour stocker et représenter les vecteurs et matrices est également complètement effacée. La transition complète en faveur de NTL permet de représenter ces structures et d'accéder à certains algorithmes sur celles-ci utilisés dans LatMRG.

Complètement intégrer NTL a nécessité la surcharge de plusieurs fonctions. LatMRG est conçu pour s'exécuter sur divers types permettant de représenter les entiers et les nombres en virgule flottante. NTL tend à n'implémenter ses algorithmes que pour les représentations qu'elle implémente également. Il a fallu surcharger et implémenter plusieurs fonctions d'arithmétique de base pour avoir la même interface pour les types de base de C++ et ceux de NTL.

Dans un autre ordre d'idée, LatMRG peut maintenant adéquatement tester la période maximale de tous les types de GLMm mentionés. Cela implique l'ajout du test de période maximale pour les GCLM et pour les GCL avec $c \neq 0$ (proposition 1.2.3).

Finalement, même s'il était précédemment possible de construire $L_t(I)$, il n'y avait pas moyen de le faire automatiquement pour une variété d'ensembles I. La logique permettant de générer séquentiellement les ensembles I a été réécrite. Cela permet, entre autres, de profiter des conditions avantageuses dans le cas où les générateurs ont une pleine période (section 2.3.2).

3.2. Programme exécutable LatMRG

En plus des différentes modifications au code de la librairie, ce mémoire présente également le nouveau programme exécutable de LatMRG. Celui-ci a été presque entièrement réécrit avec pour objectif d'être plus facile à développer et utiliser.

Il était important que ce nouvel exécutable conserve les diverses utilités de LatMRG. Précédemment, LatMRG contenait 4 différents programmes exécutables avec différentes fonctions :

(1) chercher des combinaisons de m et k facilitant la recherche de générateurs de pleine période;
(2) tester si un générateur possède une pleine période;
(3) tester la structure de réseau d'un générateur;
(4) chercher de nouveaux générateurs et les classer par le mérite de leur structure de réseau.

Toutes ces fonctions sont encore présentes et sont intégrées dans un seul programme comme différents modes de fonctionnement. Pour simplifier la référence à ces différents modes un peu plus loin, la même numérotation sera utilisée.

Le principal changement modifiant la facilité d'utilisation du programme est la modification du format des fichiers de configuration. Les anciens fichiers de configuration du programme étaient rudimentaires avec chaque ligne contenant une information, dans un ordre spécifique. Il était difficile d'écrire un fichier de configuration correct et cela créait des problèmes lorsque l'on modifiait le code. Modifier le code pour lire une nouvelle information ailleurs que sur la dernière ligne faisait en sorte que tous les fichiers de configuration déjà existants n'étaient plus compatibles avec le programme.

Pour changer cela, les fichiers de configuration sont maintenant en format XML. La principale particularité du langage est que chaque information que le fichier contient doit être mise entre des bornes spécifiant sa nature. Il est possible de définir différentes bornes pour contenir divers paramètres. Cette extensibilité donne donc la possibilité d'ajouter autant d'options que souhaité.

Il est aussi possible de spécifier la structure des fichiers afin d'être moins rigides. Il est maintenant possible d'ajouter des fonctionnalités au programme exécutable, et même des paramètres au programme, en conservant la compatibilité avec les anciens fichiers de configuration. Lorsque des erreurs surviennent le programme est en mesure d'identifier leur source, que ce soit un paramètre manquant ou un autre mal spécifié. Il est aussi beaucoup plus facile de spécifier des valeurs par défaut au programme qui est maintenant capable de reconnaître si une information est absente.

Avant de présenter des exemples de l'utilisation de LatMRG plus loin dans le chapitre, on décrit ici le fonctionnement des divers modes de l'exécutable.

La période des générateurs peut être étudiée en utilisant les modes (1) et (2). Dans le mode (2), on teste la pleine période directement en vérifiant les conditions de la proposition 1.2.1 ou la proposition 1.2.3. Le mode (1) est utile lorsque l'on cherche de nouveaux générateurs.

Le mode (1) prend une valeur de k et un intervalle (a,b) en entrée et cherche tous les $a < m < b$ premiers dans cet intervalle. Il est possible d'exiger que $r = (m^k - 1)/(m - 1)$ ou $(m - 1)/2$ soient des nombres premiers. Notons que pour que r soit premier, il faut que k soit impair. Autrement, la factorisation $m^k - 1 = (m^{k/2} - 1)(m^{k/2} + 1)$ existe. Vérifier r premier est particulièrement utile afin de trouver des valeurs de m pour les GCMC, puisque cela permet de s'assurer que la combinaison est de période maximale. Il est possible de

demander au programme factoriser $m-1$ et r après avoir trouvé m. L'objectif de ce mode est de chercher des modulos m pouvant être utilisés pour les GLMm d'ordre k.

Le mode (2) vérifie si le générateur qu'on lui passe possède une pleine période. Pour cela, il faut vérifier soit la proposition 1.2.1 pour les GCM, GMR ou GCLM ou la proposition 1.2.3 s'il s'agit d'un GCM avec $c \neq 0$. Dans la proposition 1.2.1, pour vérifier les conditions (1) et (3), il faut connaître une factorisation pour $m-1$ et pour r. Il est parfois difficile de les calculer, cette difficulté est évitée en choisissant m et k adéquatement. Si la factorisation de $m-1$ et r ne peut pas être faite en temps raisonnable on les choisit tels que $(m-1)/2$ et r sont premiers. C'est pourquoi il est possible de s'assurer qu'ils le soient dans le mode (1).

Il est à noter que pour les GCMC, le programme ne calcule pas la période de la combinaison qui, elle, n'est jamais de pleine période. Dans ce cas, il faut que l'utilisateur vérifie que chacune de ses composantes soit de pleine période. Avec cette condition, si r et $(m-1)/2$ sont premiers pour chaque composante, alors la période du générateur sera $\rho_1 \cdots \rho_J / 2^{J-1}$, la période maximale pour ce type de générateurs.

Pour **tester un GLMm**, il faut d'abord en fournir une description à LatMRG. Les tests implémentés par LatMRG se basent tous sur la longueur du plus court vecteur non-nul dans le réseau, tel que discuté dans la section 2.3.

Pour fonctionner, le programme a besoin de la description du test qui s'articule en deux parties. D'abord, il faut lui dire ce que l'on veut calculer sur le réseau. Que ce soit sur le primal ou le dual, en norme euclidienne ou en norme L1, le programme calculera la longueur du plus court vecteur dans le réseau, ou une approximation de celle-ci. Pour faire ces approximations, le programme se contente d'appliquer la réduction LLL ou la réduction BKZ sans faire la procédure de S&E. Le programme peut alors retourner diverses valeurs en fonction de la norme : la valeur brute de celle-ci, son inverse, ou alors une version normalisée.

Ensuite, le programme nécessite des valeurs de d et $t_1, \ldots, t_d$ dont il peut se servir pour construire des ensembles de projections $\mathcal{I}$ comme dans 2.3.2. Sur chacune des projections de cet ensemble, le programme calculera la mesure demandée, ce qui permet de faire le calcul de figures de mérite telles que décrites dans cette même section.

Ce mode peut être utilisé pour comparer divers générateurs entre eux grâce aux figures de mérite normalisées. On peut aussi l'utiliser pour obtenir de l'information auxiliaire sur un générateur. Que ce soit le nombre d'hyperplans qui couvrent ses points, ou encore les

projections sur lesquelles il est moins uniforme. Il est possible d'obtenir le résultat détaillé du test pour chaque projection. Cela permet de vérifier si certains ensembles de coordonnées causent des échecs systématiques et de trouver des failles comme cela a déjà été fait dans la littérature [**36, 39, 40**].

Chercher de nouveaux générateurs se fait similairement au test d'un générateur. Il faut spécifier quel test on veut que le programme applique et l'ensemble de projections que l'on veut utiliser. Par contre, la spécification du générateur se fait différemment.

Pour rechercher des générateurs avec LatMRG, il faut obligatoirement spécifier le modulo et l'ordre du générateur que l'on veut obtenir. Il est aussi possible de spécifier si on veut un générateur de pleine période ou non. Le cœur du sujet se trouve cependant dans la manière qu'aura le programme de chercher les coefficients. Une recherche s'exécute comme suit.

(1) Lire les conditions du programme.

(2) Trouver un générateur respectant les conditions

(3) Tester ce générateur.

(4) Comparer le mérite de ce générateur avec ceux déjà obtenus.

(5) Si le temps alloué n'est pas terminé, retourner en (2), sinon imprimer les résultats.

Dans l'ancienne version de LatMRG, la partie (2) faisait des appels récursifs à des fonctions qui contruisaient les multiplicateurs du générateur un par un. Ce code était difficile à suivre et modifier et un des objectifs de la réécriture de LatMRG est de permettre de modifier facilement cette procédure. Maintenant, cette recherche cruciale est effectuée par l'une ou l'autre de multiples fonctions. Chacune de ces fonctions permet de chercher des générateurs selon une spécification différente. De même, pour construire des générateurs avec des spécifications différentes, il suffit d'écrire une fonction construisant les multiplicateurs de ceux-ci. Cela rend le programme plus linéaire et facilite l'ajout de types de recherches.

Présentement, il est possible de chercher les types de générateurs suivants.

- Des GCM avec des coefficients aléatoires ou nuls dans $[0,m-1]$;
- des GCM aux coefficients construits comme des sommes d'un certain nombre de puissances de 2;
- des GCM aux coefficients aléatoires respectant les conditions de factorisation approximative pour m;
- des GCL à coefficient aléatoire pour lesquel $c \neq 0$;

- des GMR à coefficients aléatoires;
- des GCMC dont les composantes sont des GCM ou des GMR. Cette recherche peut construire les GCM de toutes les manière précédemment mentionnées.

Il n'est pas présentement possible de chercher des GCLM avec LatMRG, principalement parce qu'il n'est pas efficace de chercher des matrices aléatoires si on veut obtenir un générateur de pleine période. En pratique, on voudra plutôt trouver les meilleurs paramètres pour une forme générale de matrices. Les travaux effectués avec LatMRG jusqu'à maintenant n'ont pas eu comme objectif de chercher de tels générateurs. De telles formes de matrices n'étaient donc pas précédemment disponibles dans le logiciel.

3.3. Exemples

On présente deux exemples impliquant LatMRG. Ces deux exemples sont des recherches de générateurs dans des contextes très différents. Leur objectif est d'illustrer la flexibilité du logiciel et le nombre d'options disponibles.

Le premier exemple travaille avec des contraintes sur la dimension du modulo et de l'état du générateur. Les contraintes définies ici sont artificielles, il s'agit d'un exemple pédagogique afin d'illustrer l'utilisation du logiciel.

Exemple 3.3.1. *Supposons que l'on cherche un générateur avec* $m < 2^{16}$ *et dont on veut que l'état ne prenne pas plus de* 64 *bits. Dans ce cas, la période du générateur n'excédera pas* 2^{64}.

Pour 5 paramétrisations différentes, on cherche le meilleur générateur pour M_{35} *pendant 60 secondes avec LatMRG. Ensuite, on compare les générateurs trouvés selon le test spectral pour un plus grand nombre de projections, la longueur de leur période et leur vitesse.*

(1) La première option que l'on peut considérer est de faire un GCL avec pour modulo 2^{16} *et* c *impair. Puisque le nombre de candidat est relativement petit, on fait une recherche exhaustive. On trouve le meilleur multiplicateur* $a = 53283$ *avec* $M_{35} = 0{,}661438$ *au bout de 32s. Puisque l'on n'étudie pas la performance statistique du générateur,* c *n'a pas d'importance, on fixe* $c = 12345$ *pour l'implémentation.*

(2) Une autre option est de prendre un GCM avec m *près de* 2^{16} *et* $k = 4$. *Avec LatMRG, on trouve que le* m *premier le plus près de* 2^{16} *est* $2^{16} - 15$. *On trouve les multiplicateurs* $a_1 = 25326$, $a_2 = 64600$, $a_3 = 46104$ *et* $a_4 = 24819$ *avec* $M_{35} = 0{,}648333$.

(3) On cherche un GCM avec $a_2 = a_3 = 0$, $m = 2^{16} - 15$ *et* $k = 4$. *On retient* $a_1 = 32907$ *et* $a_4 = 17770$ *avec* $M_{35} = 0{,}202752$.

(4) On cherche un GCMC avec 2 composantes d'ordre 2. On choisit les modulos près de 2^{16}, *on retient* $m_1 = 2^{16} - 269$ *et* $m_2 = 2^{16} - 389$. *On obtient* $a_{1,1} = 29602$, $a_{1,2} = 44944$, $a_{2,1} = 10445$ *et* $a_{2,2} = 7526$ *avec* $M_{35} = 0{,}645561$.

(5) On cherche cette fois un GCM avec m près de 2^8 *avec* $k = 8$. *Avec LatMRG, on trouve que le m premier le plus près de* 2^8 *est* $2^8 - 5$. *On trouve* $a_1 = 44$, $a_2 = 0$, $a_3 = 60$, $a_4 = 63$, $a_5 = 218$, $a_6 = 102$, $a_7 = 0$ *et* $a_8 = 142$ *avec* $M_{35} = 0{,}64377$.

	(1)	(2)	(3)	(4)	(5)
M_{35}	0,661438	0,648333	0,202752	0,645561	0,64377
Période	2^{16}	$(2^{16}-15)^4$	$(2^{16}-15)^4$	$\frac{1}{2}(2^{16}-269)^2(2^{16}-389)^2$	$(2^8-5)^8$
Période	$6{,}55 \cdot 10^4$	$1{,}84 \cdot 10^{19}$	$1{,}84 \cdot 10^{19}$	$9{,}04 \cdot 10^{18}$	$1{,}58 \cdot 10^{19}$
$M_{45,50,50}$	0,0541266	0,648333	0,111183	0,0423965	0,64377
t_{10^9}	1,659s	12,872s	6,980s	10,515s	6,243s
Somme	499993179	499985112	499981114	500000300	498011672

Tab. 3.1. Comparaisons des générateurs de l'exemple 3.3.1

Dans le tableau 3.1, t_{10^9} est le temps nécessaire pour générer et additionner 10^9 nombres en virgule flottante dans l'intervalle [0,1) sur un processeur i5-7200U (x86_64) et *Somme* est la somme de ces nombres. Les nombres affichés pour ces deux catégories sont la moyennes de 10 essais utilisant des germes différentes et aléatoires. La somme permet, entre autres de vérifier que le générateur n'est pas biaisé. Cela ne nous assure pas que l'implémentation est sans faute, mais permet d'éviter certaines erreurs. Par exemple, on remarque avec la somme que le GCM (5) utilisant $m = 251$ est biaisé s'il n'est pas modifié. Puisque les états $0 \leq x_i \leq 250$ sont uniformes pour toutes les valeurs de l'intervalle, l'espérance de la somme est effectivement différente

$$\mathbb{E}\left[\sum u_i\right] = 10^9 \sum_{k=0}^{250} k/251^2 = 498\ 007\ 968.$$

Pour corriger cette défaillance, on peut prendre $u_i = (x_i + 1)/(m + 1)$, alors

$$\mathbb{E}\left[\sum u_i\right] = 10^9 \sum_{k=1}^{251} k/(251 \cdot 252) = 500\ 000\ 000.$$

En pratique, on ne prendra jamais m aussi petit et ce genre de défaillance ne surviendra pas. Par contre, cela permet d'illustrer comment modifier légèrement la sortie du générateur pour contourner certains problèmes. Il est possible de prendre cette même fonction de sortie $u_i = (x_i + 1)/(m + 1)$ pour obtenir des nombres dans (0,1) plutôt que [0,1) ou de prendre $u_i = (x_i + 1)/m$ pour obtenir des nombres dans (0,1]. Il est important, dans certaines applications, de ne pas pouvoir générer les valeurs 0 ou 1.

Tous les générateur, hormis le GCL, ont des périodes de longueur similaires. Les résultats illustrent bien le compromis qu'il faut souvent faire entre les bons résultats aux tests théoriques et la vitesse des générateurs. Le générateur performant le mieux au test spectral est (2), mais celui-ci est également le plus lent. On peut gagner beaucoup de performance en réduisant le nombre de coefficients non-nuls, mais les résultats au test spectral de (3) sont plutôt médiocres. Puisque (4) est équivalent à un GCM d'ordre 2, ses résultats au test spectral sont mauvais. Finalement, (5) est relativement rapide, car il faut appliquer beaucoup moins d'opérations modulo pour éviter les débordements. Sa performance au test spectral est elle aussi plutôt bonne, mais en pratique le petit modulo peut causer problème : ce générateur a seulement 251 sorties possibles.

Il existe peu de GCM utilisant les 64 bits des processeurs modernes. Avec ceux-ci, il est possible d'obtenir des générateurs de périodes similaires à ceux existant, mais en utilisant moins d'opérations. Il est aussi possible d'obtenir une plus grande variété de nombres uniformes, ce qui peut aider à la performance aux tests statistiques. LatMRG peut être utilisé pour trouver de nouveaux générateurs 64 bits efficaces avec une bonne uniformité pour la simulation.

La section 1.3 présente ce qu'il faut prendre en considération pour le choix des multiplicateurs d'un générateur. L'exemple 3.3.2 étudie les générateurs dont les coefficients sont des puissances de 2. Les multiplications effectuées par des décalages de bits peuvent permettre d'améliorer l'efficacité des générateurs **[38]**. Par contre, comme avec les générateurs *xorshift*, les décalages de bits ne « mélangent » parfois pas suffisamment les bits des générateurs **[34, 38, 47]**.

Exemple 3.3.2. *On cherche des générateurs de cette forme avec différents nombres de puissances de 2 dans les multiplicateurs et on compare leur performance en termes de vitesse et dans le test spectral. Un des objectifs de cette recherche est de trouver le nombre de puissances de 2 que l'on peut utiliser avant que le gain en efficacité s'estompe par rapport à un GCM standard.*

Il faut faire un certains nombre de choix pour paramétriser les générateurs avant de chercher des paramètres. On cherchera des GCMC d'ordre 3. Pour simplifier l'implémentation (et la rendre plus efficace) on utilisera $m < 2^{59}$. *Spécifiquement, on retient* $m_1 = 2^{59} - 140769$ *et* $m_2 = 2^{59} - 194745$ *(qui sont tels que* $(m-1)/2$ *et* $(m^3-1)/(m-1)$ *sont premiers). On veut des composantes de période maximale. Les générateurs auront donc une période de longueur d'environ* 2^{352}. *On impose* $a_{1,1} = a_{2,2} = 0$.

Supposons que l'on utilise $m = 2^e - h$ *et que l'on veuille calculer* $2^q x \bmod m$ *avec* $q < e$. *Comme vu à la section 1.3.1, on décompose* $x = x_0 + 2^{e-q}x_1$ *et on calcule* $2^q x \bmod m = (2^q x_0 + h x_1) \bmod m$. *Les résultats des deux produits seront plus petits que* m *et ne vont pas déborder à condition que* $h < 2^q$ *et que* $h(2^q - (h+1)2^{q-e}) < m$. *Le tableau 3.2 présente les intervalles de valeurs possibles pour* q *avec ces contraintes pour nos choix de* m.

e	h	*Intervalle pour* q
59	140769	$18 \leq q \leq 41$
59	194745	$18 \leq q \leq 41$

Tab. 3.2. Intervalle des valeurs de q

On cherche un GCM avec la contrainte $a^2 < m$. *On cherche aussi des générateurs utilisant chacun soit 2, 3 ou 4 puissances de 2 pour construire les multiplicateurs. Chacune des recherche est exécutée en 2 minutes. Les multiplicateurs trouvés sont présentés dans le tableau 3.3.*

Pour chacun de ces 4 générateurs, on propose une implémentation et on compare sa vitesse d'exécution. Le code des générateurs est présenté à l'annexe A dans les figures A.1, A.2, A.3 et A.4. On calcule aussi $M_{45,50,50,50,25}$. *Les résultats sont présentés dans le tableau 3.4.*

	GCM	*2*	*3*	*4*
$a_{1,2}$	756990256	$-2^{36} - 2^{21}$	$2^{37} - 2^{32} + 2^{21}$	$-2^{40} - 2^{25} + 2^{22} + 2^{20}$
$a_{1,3}$	443897406	$-2^{22} + 2^{19}$	$2^{41} + 2^{34} - 2^{20}$	$2^{38} - 2^{30} + 2^{25} + 2^{23}$
$a_{2,1}$	512643533	$2^{35} + 2^{18}$	$-2^{40} - 2^{33} - 2^{32}$	$2^{41} - 2^{36} - 2^{28} - 2^{24}$
$a_{2,3}$	516293997	$-2^{41} + 2^{27}$	$2^{31} + 2^{22} - 2^{18}$	$-2^{38} - 2^{25} + 2^{24} + 2^{18}$

Tab. 3.3. Multiplicateurs trouvés selon le nombre de puissances

	GCMC	2	3	4
$M_{45,50,50,50,25}$	0,000935819	0,0256324	0,0205169	0,0586996
Pire projection	{0,2,3}	{0,20,45,49}	{0,30,42,48}	{0,8,12,45}
t_{10^9}	12,07s	11,40s	13,89s	16,17s

Tab. 3.4. Multiplicateurs trouvés selon le nombre de puissances

Il était à prévoir que, pour un si grand nombre de projections, les générateurs auraient une figure de mérite petite. On constate que celle du GCMC n'utilisant pas des puissances de 2 est beaucoup plus petite, ce qui est inacceptable. Il ne semble pas y avoir de raison particulière causant cela; le problème est probablement causé par cette instance spécifique de générateur. Pour ce qui est des générateurs avec des coefficients qui sont des puissances de 2, il n'y a pas de corrélation claire entre le nombre de puissance de 2 et le mérite du générateur avec autant de projections. Comme on teste des générateurs combinés, la combinaison cache la faiblesse des composantes.

On remarque aussi que le nombre de puissances de 2 semble faire augmenter linéairement le temps d'exécution des générateurs. Il y a certainement une différence entre les générateur utilisant des coefficients en puissances de 2 et les générateurs utilisant un produit régulier. Utiliser deux puissances de 2 accélère légèrement le générateur par rapport à la factorisation approximative, mais le gain s'estompe dès la troisième.

Il peut être intéressant de relativiser ces résultats en les comparant à ceux de d'autres générateurs de construction similaire, mais pour les processeurs 32 bits. On considère MRG32k3a, un bon GCMC à 32 bits [**22**] dont plusieurs implémentations sont disponibles.

On en teste deux différentes : l'implémentation originale du générateur, utilisant les nombres en virgules flottantes pour avoir plus de précision dans le représentation des nombres, et une implémentation récente proposée par Vigna disponible dans la figure A.6 de l'annexe A qui utilise l'arithmétique sur les entiers de 64 bits. On teste aussi une version modifiée (à l'annexe A dans la figure A.5) du générateur MRG31k3p **[38]**, un GCMC utilisant des puissances de 2 dans la construction de ces coefficients. Les résultats sont présentés dans le tableau 3.5.

	MRG32k3a virgule flottante	MRG32k3a entiers	MRG31k3p
$M_{45,50,50,50,25}$	0,0532135	0,0532135	0,0248037
Pire projection	{0,39,42,44}	{0,39,42,44}	{0,2,3}
t_{10^9}	27,65s	5,41s	7,53s

Tab. 3.5. Multiplicateurs trouvés selon le nombre de puissances

En termes de performance, il est clair que l'accès à des entiers de 64 bits peut être exploité afin de trouver des générateurs plus efficaces. L'implémentation « historique » de MRG32k3a est 70% plus lente que celle de l'autre générateur le plus lent testé dans cette section. Sur les processeurs modernes, ce ne sont pas les même opérations qu'autrefois qui s'effectuent rapidement. Par exemple, non seulement l'implémentation en entiers de MRG32k3a est 5 fois plus rapide que celle en virgule flottante, mais elle est aussi plus rapide qu'une implémentation comparable de MRG31k3p. À l'origine, c'était pourtant le second qui était plus rapide. C'est cela que le dernier exemple explore.

Exemple 3.3.3. *On cherche des GLMm rapides possédant une bonne performance au test spectral utilisables en simulation Monte-Carlo. On choisit de chercher des générateurs combinés d'ordre 3 avec* $a_{1,1} = a_{2,2} = 0$. *Ce choix permet de limiter le nombres d'opérations du générateur, mais ne contraint pas la forme du générateur équivalent à la combinaison. Il s'agit donc d'un compromis entre la performance théorique et la vitesse.*

Pour obtenir plus de vitesse, on exige que les multiplications du générateur se fassent directement sur un processeur 64 bits. C'est-à-dire qu'il faut que $a_{i,j}m_i < 2^{63} - 1$. *Pour cela, on cherchera des générateurs avec des modulos beaucoup plus petit que* 2^{64}*, mais tout de même plus grands que* 2^{32}.

On veut des générateurs qui performeront mieux au test spectral que dans l'exemple 3.3.2. Pour cela, on utilise une figure de mérite plus exigeante lors des recherches : on choisit $M_{35,15,15,15}$. *Cette figure de mérite est beaucoup plus longue à calculer que le* M_{35} *des exemples précédent. Elle reste cependant relativement accessible à calculer; il est possible de tester quelques milliers de générateurs par heure. Elle devrait permettre de ne pas retenir de générateur dont la performance est très mauvaise en augmentant le nombre de projections. Cela a été le cas dans l'exemple 3.3.2 pour le GCMC utilisant des multiplications sans puissances de 2.*

On fait une expérience préliminaire pour déterminer quels modulos utiliser. Pour chaque $32 \leq e \leq 41$, *on cherche 2 modulos tels que* r *et* $(m-1)/2$ *sont premiers et plus petits que* 2^e. *Pour chacune de ces paires, on cherche des générateurs pendant 180 secondes. On trace une courbe de la moyenne du mérite des 5 meilleurs générateurs pour chaque* e *dans la figure 3.1.*

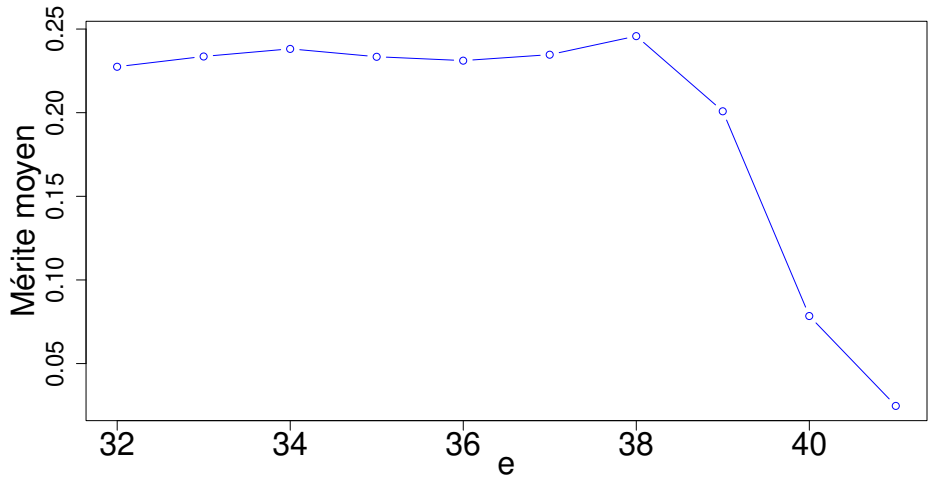

Fig. 3.1. Courbe de $M_{35,15,15,15}$ en fonction de l'exposant

Idéalement, on veut prendre m *le plus grand possible. On voit que, lorsqu'il augmente, la qualité des générateurs diminue. Cela est lié au fait que la taille de multiplicateurs possibles devient relativement petite et que de trop petits multiplicateurs influencent négativement les résultats du test spectral* [**32**].

On décide, pour faire des recherches approfondies, d'utiliser les trois combinaisons de modulos suivantes : $m_1 = 2^{37} - 20745$ *et* $m_2 = 2^{37} - 29313$; $m_1 = 2^{38} - 4625$ *et* $m_2 = 2^{38} - 21257$;

$m_1 = 2^{39} - 32385$ *et* $m_2 = 2^{39} - 76221$. *Les périodes des générateurs seront, respectivement, d'environ* 2^{221}, 2^{227} *et* 2^{233}, *toutes plus grandes que* 2^{200}. *Pour chaque combinaison, on fait une recherche de 2 heures et on retient les 5 meilleurs générateurs. Parmi les 15 générateurs obtenus, on retient les 3 meilleurs pour* $M_{45,50,50,50,25}$. *Ils sont présentés dans le tableau 3.6.*

e	*37*	*39*	*37*
$M_{35,15,15,15}$	0,255319	0,221912	0,251118
$M_{45,50,50,50,25}$	0,0802494	0,0762487	0,0715412
$a_{1,2}$	18997718	15600308	64485701
$a_{1,3}$	38584692	11962985	28633419
$a_{2,1}$	412406	11353736	19480496
$a_{2,3}$	31336619	15446194	58151419
t_{10^9}	4,58400s	6,07290s	4,58630s

Tab. 3.6. Multiplicateurs des 3 meilleurs générateurs trouvés

La figure de mérite des générateurs obtenus n'est pas très bonne, mais est déjà bien meilleure que pour les générateurs trouvés à l'exemple 3.3.2. On remarque que ces 3 générateurs ont une meilleure performance au test spectral que MRG32k3a pour $M_{45,50,50,50,25}$, ce qui est prometteur. Cette performance seule n'est pas suffisante afin qu'ils soient recommendables pour la simulation. Les générateurs n'ont pas encore subi de tests statistiques et leurs résultats au test préliminaire sont assez faibles.

En termes de vitesse, les générateurs sont tous très rapides, ce qui atteint l'objectif fixé. Il est à noter que celui utilisant des modulos plus grand ($e = 39$) est considérablement plus lent. Cela est surprenant étant donné que les 3 générateurs ont essentiellement la même implémentation (disponibles à l'annexe A dans les figures A.7, A.8 et A.9).

Il apparaît évident qu'il est possible de trouver de nouveaux générateurs intéressants d'un point de vue pratique en utilisant cette stratégie de recherche. Il est difficile d'optimiser le résultat au test spectral pour une figure de mérite considérant autant de projections. Des résultats plus probants nécessiteront un budget de temps plus important.

Conclusion

Ce mémoire présente le logiciel LatMRG et la théorie derrière ce logiciel. Sa principale contribution au logiciel est à l'accessibilité de celui-ci. LatMRG est maintenant plus simple d'utilisation, grâce au changement des fichiers de configuration. Il est aussi plus aisé de travailler sur LatMRG puisque le nouvel exécutable a maintenant un comportement linéaire plus facile à comprendre. Finalement, la documentation de LatMRG est plus complète pour permettre de simplifier l'utilisation de la librairie.

Les exemples présentés au chapitre 3 illustrent très bien le fait que l'architecture des ordinateur a un impact significatif sur l'efficacité d'un GNA. Dans ce contexte, il est essentiel d'avoir une suite d'outils efficaces permettant de bâtir des GNA adaptés à différentes applications. LatMRG s'incrit très bien dans cette optique. Ce logiciel contient un ensemble flexible et compréhensif de fonctionnalité permettant de chercher et étudier les GLMm. Cette grande flexibilité est probablement le principal atout du logiciel.

Les travaux effectués sur LatMRG devraient permettre de l'utiliser pour explorer plus en détails les GLMm pour les processeurs modernes à 64 bits. Ce type de générateur a décliné en popularité dans les dernières années et peu de travaux ont été effectués pour présenter de nouveaux générateurs adaptés. Il existe donc de nombreuses avenues à explorer.

LatMRG peut aussi permettre de rechercher des GLMm de toutes les tailles qui pourraient être implémentés de manière matérielle. Il est également possible de s'assister de LatMRG pour étudier la performance des différents types de GLMm en fonction des processeurs.

Il y a également plusieurs voies à explorer dans LatMRG même. Il peut s'avérer intéressant de chercher des formes de matrices qui permettraient d'obtenir des GCLM de pleine période et efficaces. Il pourrait aussi être possible d'utiliser LatMRG et TestU01 [**35**] conjointement.

Bibliographie

[1] L. Afflerbach et H. Grothe : The lattice structure of pseudo-random vectors generated by matrix generators. *Journal of Computational and Applied Mathematics*, 23:127–131, 1988.

[2] Thomas Bradley, Jacques du Toit, Robert Tong, Mike Giles et Paul Woodhams : Parallelization techniques for random number generations. *In GPU Computing Gems Emerald Edition*, pages 231–246. Morgan Kaufmann, 2011. Chapter 16.

[3] E. Buist et P. L'Ecuyer : *ContactCenters: A Java Library for Simulating Contact Centers*, 2012. Software user's guide, available at `http://www.simul.umontreal.ca/contactcenters`.

[4] J. H. Conway et N. J. A. Sloane : *Sphere Packings, Lattices and Groups*. Grundlehren der Mathematischen Wissenschaften 290. Springer-Verlag, New York, 3rd édition, 1999.

[5] R. Couture et P. L'Ecuyer : On the lattice structure of certain linear congruential sequences related to AWC/SWB generators. *Mathematics of Computation*, 62(206):798–808, 1994.

[6] R. Couture et P. L'Ecuyer : Linear recurrences with carry as random number generators. *In Proceedings of the 1995 Winter Simulation Conference*, pages 263–267, 1995.

[7] R. Couture et P. L'Ecuyer : Orbits and lattices for linear random number generators with composite moduli. *Mathematics of Computation*, 65(213):189–201, 1996.

[8] R. Couture et P. L'Ecuyer : Distribution properties of multiply-with-carry random number generators. *Mathematics of Computation*, 66(218):591–607, 1997.

[9] U. Dieter : How to calculate shortest vectors in a lattice. *Mathematics of Computation*, 29(131):827–833, 1975.

[10] David S Dummit et Richard M. Foote : *Abstract Algebra*. John Wiley and sons, Inc., third édition, 2004.

[11] A. M. Ferrenberg, D. P. Landau et Y. J. Wong : Monte Carlo simulations: Hidden errors from "good" random number generators. *Physical Review Letters*, 69(23):3382–3384, 1992.

[12] U. Fincke et M. Pohst : Improved methods for calculating vectors of short length in a lattice, including a complexity analysis. *Mathematics of Computation*, 44:463–471, 1985.

[13] M. Goresky et A. Klapper : Efficient multiply-with-carry random number generators with maximal period. *ACM Transactions on Modeling and Computer Simulation*, 13(4):310–321, 2003.

[14] A. Grube : Mehrfach rekursiv-erzeugte Pseudo-Zufallszahlen. *Zeitschrift für angewandte Mathematik und Mechanik*, 53:T223–T225, 1973.

[15] D. E. Knuth : *The Art of Computer Programming, Vol. 2.* Addison-Wesley, Reading, MA, second édition, 1981.

[16] P. L'Ecuyer : Efficient and portable 32-bit random variate generators. *In Proceedings of the 1986 Winter Simulation Conference*, pages 275–277, 1986.

[17] P. L'Ecuyer : Efficient and portable combined random number generators. *Communications of the ACM*, 31(6):742–749 and 774, 1988. See also the correspondence in the same journal, 32, 8 (1989) 1019–1024.

[18] P. L'Ecuyer : Uniform random number generation. *Annals of Operations Research*, 53:77–120, 1994.

[19] P. L'Ecuyer : Combined multiple recursive random number generators. *Operations Research*, 44(5): 816–822, 1996.

[20] P. L'Ecuyer : Maximally equidistributed combined Tausworthe generators. *Mathematics of Computation*, 65(213):203–213, 1996.

[21] P. L'Ecuyer : Bad lattice structures for vectors of non-successive values produced by some linear recurrences. *INFORMS Journal on Computing*, 9(1):57–60, 1997.

[22] P. L'Ecuyer : Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47(1):159–164, 1999.

[23] P. L'Ecuyer : Good parameters and implementations for combined multiple recursive random number generators. Available as file `combmrg2.ps` at URL `www.iro.umontreal.ca/~lecuyer`. The C code is in file `combmrg2.c`, 1999.

[24] P. L'Ecuyer : Variance reduction's greatest hits. *In Proceedings of the 2007 European Simulation and Modeling Conference*, pages 5–12, Ghent, Belgium, 2007. EUROSIS.

[25] P. L'Ecuyer : Random number generation. *In* J. E. Gentle, W. Haerdle et Y. Mori, éditeurs : *Handbook of Computational Statistics*, pages 35–71. Springer-Verlag, Berlin, second édition, 2012.

[26] P. L'Ecuyer : Random number generation with multiple streams for sequential and parallel computers. *In Proceedings of the 2015 Winter Simulation Conference*, pages 31–44. IEEE Press, 2015.

[27] P. L'Ecuyer : History of uniform random number generation. *In Proceedings of the 2017 Winter Simulation Conference*, pages 202–230. IEEE Press, 2017.

[28] P. L'Ecuyer et F. Blouin : Linear congruential generators of order $k > 1$. *In Proceedings of the 1988 Winter Simulation Conference*, pages 432–439. IEEE Press, 1988.

[29] P. L'Ecuyer, F. Blouin et R. Couture : A search for good multiple recursive random number generators. *ACM Transactions on Modeling and Computer Simulation*, 3(2):87–98, 1993.

[30] P. L'Ecuyer et E. Buist : Variance reduction in the simulation of call centers. *In Proceedings of the 2006 Winter Simulation Conference*, pages 604–613. IEEE Press, 2006.

[31] P. L'Ecuyer et S. Côté : Implementing a random number package with splitting facilities. *ACM Transactions on Mathematical Software*, 17(1):98–111, 1991.

[32] P. L'Ecuyer et R. Couture : An implementation of the lattice and spectral tests for multiple recursive linear random number generators. *INFORMS Journal on Computing*, 9(2):206–217, 1997.

[33] P. L'Ecuyer, D. Munger, B. Oreshkin et R. Simard : Random numbers for parallel computers: Requirements and methods, with emphasis on GPUs. *Mathematics and Computers in Simulation*, 135:3–17, 2017. Open access at `http://dx.doi.org/10.1016/j.matcom.2016.05.005`.

[34] P. L'Ecuyer et R. Simard : Beware of linear congruential generators with multipliers of the form $a = \pm 2^q \pm 2^r$. *ACM Transactions on Mathematical Software*, 25(3):367–374, 1999.

[35] P. L'Ecuyer et R. Simard : TestU01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 33(4):Article 22, August 2007.

[36] P. L'Ecuyer et R. Simard : On the lattice structure of a special class of multiple recursive random number generators. *INFORMS Journal on Computing*, 26(2):449–460, 2014.

[37] P. L'Ecuyer et S. Tezuka : Structural properties for two classes of combined random number generators. *Mathematics of Computation*, 57(196):735–746, 1991.

[38] P. L'Ecuyer et R. Touzin : Fast combined multiple recursive generators with multipliers of the form $a = \pm 2^q \pm 2^r$. *In* J. A. Joines, R. R. Barton, K. Kang et P. A. Fishwick, éditeurs : *Proceedings of the 2000 Winter Simulation Conference*, pages 683–689, Piscataway, NJ, 2000. IEEE Press.

[39] P. L'Ecuyer et R. Touzin : On the Deng-Lin random number generators and related methods. *Statistics and Computing*, 14:5–9, 2004.

[40] P. L'Ecuyer, P. Wambergue et E. Bourceret : Spectral analysis of the MIXMAX random number generators. *INFORMS Journal on Computing*, 2019. To appear.

[41] D. H. Lehmer : Mathematical methods in large scale computing units. *The Annals of the Computation Laboratory of Harvard University*, 26:141–146, 1951.

[42] A. K. Lenstra, H. W. Lenstra et L. Lovász : Factoring polynomials with rational coefficients. *Math. Ann.*, 261:515–534, 1982.

[43] R. Lidl et H. Niederreiter : *Introduction to Finite Fields and Their Applications*. Cambridge University Press, Cambridge, revised édition, 1994.

[44] G. Marsaglia : Random numbers fall mainly in the planes. *Proceedings of the National Academy of Sciences of the United States of America*, 60:25–28, 1968.

[45] G. Marsaglia et A. Zaman : A new class of random number generators. *The Annals of Applied Probability*, 1:462–480, 1991.

[46] H. Niederreiter : *Random Number Generation and Quasi-Monte Carlo Methods*, volume 63 de *SIAM CBMS-NSF Reg. Conf. Series in Applied Mathematics*. SIAM, 1992.

[47] F. Panneton et P. L'Ecuyer : On the xorshift random number generators. *ACM Transactions on Modeling and Computer Simulation*, 15(4):346–361, 2005.

[48] Marc-Antoine Savard, Pierre L'Ecuyer, Erwan Bourceret, Paul Wamberge et Richard Simard : *LatMRG Guide*. Université de Montréal, Montréal, December 2019. Available at `https://savamarc.github.io/latmrg`.

[49] K. G. Savvidy : The MIXMAX random number generator. *Computer Physics Communications*, 196: 161–165, 2015.

[50] C. P. Schnorr : A hierarchy of polynomial time lattice basis reduction algorithms. *Theoretical Computer Science*, 53(2):201–224, 1987.

[51] C. P. Schnorr et M. Euchner : Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *In* L. Budach, éditeur : *Fundamentals of Computation Theory: 8th International Conference*, pages 68–85, Berlin, Heidelberg, 1991. Springer-Verlag.

[52] V. Shoup : *NTL: A Library for doing Number Theory*. Courant Institute, New York University, New York, NY, July 2018. Available at `https://shoup.net/ntl/`.

[53] S. Tezuka : *Uniform Random Numbers: Theory and Practice*. Kluwer Academic, 1995.

[54] S. Tezuka, P. L'Ecuyer et R. Couture : On the add-with-carry and subtract-with-borrow random number generators. *ACM Transactions of Modeling and Computer Simulation*, 3(4):315–331, 1993.

Annexe A

Implémentations des générateurs trouvés

Dans chacun des fichiers de code a accès à des définitions telles que `P2_A` vaut $2^A - 1$.

```
#define NORM 1.7347234759772307989996896732795790534286186840100e-18 // 1.0/MOD1
// NORM could be more precise but would not fit the line
#define MOD1 576460752303282719 // 2^59 - 140769
#define MOD2 576460752303228743 // 2^59 - 194745
#define a12 756990256
#define a13 443897406
#define a21 512643533
#define a23 516293997
#define SEED  1234

int64_t x11 = SEED, x12 = SEED, x13 = SEED, x21 = SEED, x22 = SEED, x23 = SEED;

double next (void) {
  const int64_t q12 = MOD1/a12;
  const int64_t r12 = MOD1 - MOD1/a12 * a12;
  const int64_t q13 = MOD1/a13;
  const int64_t r13 = MOD1 - MOD1/a13 * a13;
  const int64_t q21 = MOD2/a21;
  const int64_t r21 = MOD2 - MOD2/a21 * a21;
  const int64_t q23 = MOD2/a23;
  const int64_t r23 = MOD2 - MOD2/a23 * a23;

  int64_t y1, y2, r, k;
  r = (x11 + x21)%MOD1;

  /* First component*/
  k = x12/q12;
  y1 = (a12*(x12 - k*q12) + MOD1 - k*r12);
  k = x13/q13;
  y2 = (a13*(x13 - k*q13) + MOD1 - k*r13);
  y1 = (y1 + y2)%MOD1;

  x13 = x12; x12 = x11; x11 = y1;

  /* Second component*/
  k = x21/q21;
  y1 = (a21*(x21 - k*q21) + MOD2 - k*r21);
  k = x23/q23;
  y2 = (a23*(x23 - k*q23) + MOD2 - k*r23);
  y1 = (y1 + y2)%MOD2;

  x23 = x12; x22 = x21; x21 = y1;

  return r*NORM;
}
```

Fig. A.1. GCMC à coefficients aléatoires

```
#define NORM 1.73472347597723079899968967327957905342861868401 00e-18 // 1.0/MOD1
// NORM could be more precise but would not fit the line
#define MOD1 576460752303282719 // 2^59 - 140769
#define MOD2 576460752303228743 // 2^59 - 194745
#define H1 140769
#define H2 194745
#define MASK121 P2_23 // 2^23-1
#define MASK122 P2_38 // 2^38-1
#define MASK131 P2_37 // 2^37-1
#define MASK132 P2_40 // 2^40-1
#define MASK211 P2_24 // 2^24-1
#define MASK212 P2_41 // 2^41-1
#define MASK231 P2_18 // 2^18-1
#define MASK232 P2_32 // 2^32-1
#define SEED 1234

int64_t x11 = SEED, x12 = SEED, x13 = SEED, x21 = SEED, x22 = SEED, x23 = SEED;

double next (void) {
  const int64_t corr1 = 6*MOD1;
  const int64_t corr2 = 2*MOD2;

  int64_t y1, r; /* For intermediate results */
  r = (x21 + x11)%MOD1);

  /* First component */
  y1 = -((x12 & MASK121) << 36) - ((x12 & MASK122) << 21)
    - ((x13 & MASK131) << 22) + ((x13 & MASK132) << 19);
  y1 += H1*(-(x12 >> 23) - (x12 >> 38)
      - (x13 >> 37) + (x13 >> 40));
  y1 = (y1+corr1)%MOD1;

  x13 = x12; x12 = x11; x11 = y1;

  /* Second component */
  y1 = ((x21 & MASK211) << 35) + ((x21 & MASK212) << 18)
    - ((x23 & MASK231) << 41) + ((x23 & MASK232) << 27);
  y1 += H2*((x21 >> 24) + (x21 >> 41)
      - (x23 >> 18) + (x23 >> 32));
  y1 = (y1+corr2)%MOD2;

  x23 = x22; x22 = x21; x21 = y1;

  return r*NORM;
}
```

Fig. A.2. GCMC utilisant 2 puissances de 2 pour coefficients

```
#define NORM 1.73472347597723079899968967327957905342861868401 00e-18 // 1.0/MOD1
// NORM could be more precise but would not fit the line
#define MOD1 576460752303282719 // 2^59 - 140769
#define MOD2 576460752303228743 // 2^59 - 194745
#define H1 140769
#define H2 194745
#define MASK121 P2_22 // 2^22-1
#define MASK122 P2_27 // 2^27-1
#define MASK123 P2_38 // 2^38-1
#define MASK131 P2_18 // 2^18-1
#define MASK132 P2_25 // 2^25-1
#define MASK133 P2_39 // 2^39-1
#define MASK211 P2_19 // 2^19-1
#define MASK212 P2_26 // 2^26-1
#define MASK213 P2_27 // 2^27-1
#define MASK231 P2_28 // 2^28-1
#define MASK232 P2_37 // 2^37-1
#define MASK233 P2_41 // 2^41-1
#define SEED 1234

int64_t x11 = SEED, x12 = SEED, x13 = SEED, x21 = SEED, x22 = SEED, x23 = SEED;

double next (void) {
  const int64_t corr1 = 4*MOD1;
  const int64_t corr2 = 8*MOD2;

  int64_t y1, r; /* For intermediate results */
  r = (x21 + x11)%MOD1);

  /* First component */
  y1 = ((x12 & MASK121) << 37) - ((x12 & MASK122) << 32)
    + ((x12 & MASK123) << 21)
    + ((x13 & MASK131) << 41) + ((x13 & MASK132) << 34)
    - ((x13 & MASK133) << 20);
  y1 += H1*((x12 >> 22) - (x12 >> 27) + (x12 >> 38)
      + (x13 >> 18) + (x13 >> 25) - (x13 >> 39));
  y1 = (y1+corr1)%MOD1;

  x13 = x12; x12 = x11; x11 = y1;

  /* Second component */
  y1 = (-(x21 & MASK211) << 40) - ((x21 & MASK212) << 33)
    - ((x21 & MASK213) << 32)
    + ((x23 & MASK231) << 31) + ((x23 & MASK232) << 22)
    - ((x23 & MASK233) << 18);
  y1 += H2*(-(x21 >> 19) - (x21 >> 26) - (x21 >> 27)
      + (x23 >> 28) + (x23 >> 37) - (x23 >> 41));
  y1 = (y1+corr2)%MOD2;

  x23 = x22; x22 = x21; x21 = y1;

  /* Combination */
  return r*NORM;
}
```

Fig. A.3. GCMC utilisant 3 puissances de 2 pour coefficients

```
#define NORM 1.73472347597723079899968967327957905342861868401 00e-18 // 1.0/MOD1
// NORM could be more precise but would not fit the line
#define MOD1 576460752303282719 // 2^59 - 140769
#define MOD2 576460752303228743 // 2^59 - 194745
#define H1 140769
#define H2 194745
#define MASK121 P2_19 // 2^19-1
#define MASK122 P2_34 // 2^34-1
#define MASK123 P2_37 // 2^37-1
#define MASK124 P2_39 // 2^39-1
#define MASK131 P2_21 // 2^21-1
#define MASK132 P2_29 // 2^29-1
#define MASK133 P2_34 // 2^34-1
#define MASK134 P2_36 // 2^36-1
#define MASK211 P2_18 // 2^18-1
#define MASK212 P2_23 // 2^23-1
#define MASK213 P2_31 // 2^31-1
#define MASK214 P2_35 // 2^35-1
#define MASK231 P2_21 // 2^21-1
#define MASK232 P2_34 // 2^34-1
#define MASK233 P2_35 // 2^35-1
#define MASK234 P2_41 // 2^41-1
#define SEED 1234

int64_t x11 = SEED, x12 = SEED, x13 = SEED, x21 = SEED, x22 = SEED, x23 = SEED;
double next (void) {
  const int64_t corr1 = 6*MOD1;
  const int64_t corr2 = 10*MOD2;

  int64_t y1, r; /* For intermediate results */
  r = (x21 + x11)%MOD1);

  /* First component */
  y1 = -((x12 & MASK121) << 40) - ((x12 & MASK122) << 25)
    + ((x12 & MASK123) << 22) + ((x12 & MASK124) << 20)
    + ((x13 & MASK131) << 38) - ((x13 & MASK132) << 30)
    + ((x13 & MASK133) << 25) + ((x13 & MASK134) << 23);
  y1 += H1*(-(x12 >> 19) - (x12 >> 34) + (x12 >> 37) + (x12 >> 39)
      + (x13 >> 21) - (x13 >> 29) + (x13 >> 34) + (x13 >> 36));
  y1 = (y1+corr1)%MOD1;

  x13 = x12; x12 = x11; x11 = y1;

  /* Second component */
  y1 = ((x21 & MASK211) << 41) - ((x21 & MASK212) << 36)
    - ((x21 & MASK213) << 28) - ((x21 & MASK214) << 24)
    - ((x23 & MASK231) << 38) - ((x23 & MASK232) << 25)
    + ((x23 & MASK233) << 24) + ((x23 & MASK234) << 18);
  y1 += H2*((x21 >> 18) - (x21 >> 23) - (x21 >> 31) - (x21 >> 35)
      - (x23 >> 21) - (x23 >> 34) + (x23 >> 35) + (x23 >> 41));
  y1 = (y1+corr2)%MOD2;

  x23 = x22; x22 = x21; x21 = y1;

  return r*NORM;
}
```

Fig. A.4. GCMC utilisant 4 puissances de 2 pour coefficients

```
#define norm 4.656612875245796923096008868014905601739883422851562 5e-10
#define m1 2147483647
#define m2 2147462579
#define mask12 511
#define mask13 16777215
#define mask21 65535
#define SEED 1234

int64_t x10 = SEED, x11 = SEED, x12 = SEED, x20 = SEED, x21 = SEED, x22 = SEED;

static double inline next (void) {
  int64_t y1, r; /* For intermediate results */
  r = x10 - x20;
  r -= m1 * ((y1-1)>>63);

  /* First component */
  y1 = ((((x11 & mask12) << 22) + (x11 >> 9))
    + (((x12 & mask13) << 7) + (x12 >> 24)) + x12) % m1;
  // if (y1 > m1) y1 -= m1;
  x12 = x11; x11 = x10; x10 = y1;

  /* Second component */
  y1 = (((x20 & mask21) << 15) + ((x22 & mask21) << 15)
    + 21069 * ((x22 >> 16) + (x20 >> 16)) + x22) % m2;
  x22 = x21; x21 = x20; x20 = y1;

  /* Combinaison */
  return r*norm;
}
```

Fig. A.5. Version modifiée de MRG31k3p

```
/*  Written in 2019 by Sebastiano Vigna (vigna@acm.org)
To the extent possible under law, the author has dedicated all copyright
and related and neighboring rights to this software to the public domain
worldwide. This software is distributed without any warranty.
See <http://creativecommons.org/publicdomain/zero/1.0/>. */

#include <stdint.h>
#include <stdlib.h>
#include <time.h>

#define SEED 1234

int64_t __MRG32k3a_s10 = SEED, __MRG32k3a_s11 = SEED,
  __MRG32k3a_s12 = SEED, __MRG32k3a_s20 = SEED,
  __MRG32k3a_s21 = SEED, __MRG32k3a_s22 = SEED;

static double inline next(void) {
  const int64_t m1 = INT64_C(4294967087);
  const int64_t m2 = INT64_C(4294944443);
  const int32_t a12 = INT32_C(1403580);
  const int32_t a13 = INT32_C(810728);
  const int32_t a21 = INT32_C(527612);
  const int32_t a23 = INT32_C(1370589);
  const int64_t corr1 = (m1 * a13);
  const int64_t corr2 = (m2 * a23);
  const double norm = 0x1.000000d00000bp-32;

  int64_t p, r;

  r = __MRG32k3a_s12 - __MRG32k3a_s22;
  r -= m1 * ((r - 1) >> 63);

  p = (a12 * __MRG32k3a_s11 - a13 * __MRG32k3a_s10 + corr1) % m1;
  __MRG32k3a_s10 = __MRG32k3a_s11;
  __MRG32k3a_s11 = __MRG32k3a_s12;
  __MRG32k3a_s12 = p;

  p = (a21 * __MRG32k3a_s22 - a23 * __MRG32k3a_s20 + corr2) % m2;
  __MRG32k3a_s20 = __MRG32k3a_s21;
  __MRG32k3a_s21 = __MRG32k3a_s22;
  __MRG32k3a_s22 = p;

  return r * norm;
}
```

Fig. A.6. Version de MRG32k3a de Vigna

```
#include <stdint.h>
#include <stdlib.h>

#define m1 137438933327
#define m2 137438924159
#define a12 18997718
#define a13 38584692
#define a21 412406
#define a23 31336619
#define norm 7.27595868065100276763982074353101438431157266251148030278272 92e-12

#define SEED 1234

uint64_t x10 = SEED, x11 = SEED, x12 = SEED, x20 = SEED, x21 = SEED, x22 = SEED;

static double inline next(void) {
  uint64_t p, r;

  /* Combination */
  r = (x12 + x22)%m1;
  r %= m1;

  /* Component 1 */
  p = (a12 * x11 + a13 * x10) % m1;
  x10 = x11;
  x11 = x12;
  x12 = p;

  /* Component 2 */
  p = (a21 * x22 + a23 * x20) % m2;
  x20 = x21;
  x21 = x22;
  x22 = p;

  return r * norm;
}
```

Fig. A.7. Premier générateur de l'exemple 3.3.3

```
#include <stdint.h>
#include <stdlib.h>

#define m1 549755781503
#define m2 549755737667
#define a12 15600308
#define a13 11962985
#define a21 11353736
#define a23 15446194
#define norm 1.8189895106988394882978186983709919036233948475000943290069699e-12

#define SEED 1234

uint64_t x10 = SEED, x11 = SEED, x12 = SEED, x20 = SEED, x21 = SEED, x22 = SEED;

static double inline next(void) {
  uint64_t p, r;

  /* Combination */
  r = (x12 + x22)%m1;
  r %= m1;

  /* Component 1 */
  p = (a12 * x11 + a13 * x10) % m1;
  x10 = x11;
  x11 = x12;
  x12 = p;

  /* Component 2 */
  p = (a21 * x22 + a23 * x20) % m2;
  x20 = x21;
  x21 = x22;
  x22 = p;

  return r * norm;
}
```

Fig. A.8. Second générateur de l'exemple 3.3.3

```
#include <stdint.h>
#include <stdlib.h>

#define m1 137438933327
#define m2 137438924159
#define a12 64485701
#define a13 28633419
#define a21 19480496
#define a23 58151419
#define norm 7.27595868065100276763982074353101438431157266251148030278272 92e-12

#define SEED 1234

uint64_t x10 = SEED, x11 = SEED, x12 = SEED, x20 = SEED, x21 = SEED, x22 = SEED;

static double inline next(void) {
  uint64_t p, r;

  /* Combination */
  r = (x12 + x22)%m1;
  r %= m1;

  /* Component 1 */
  p = (a12 * x11 + a13 * x10) % m1;
  x10 = x11;
  x11 = x12;
  x12 = p;

  /* Component 2 */
  p = (a21 * x22 + a23 * x20) % m2;
  x20 = x21;
  x21 = x22;
  x22 = p;

  return r * norm;
}
```

Fig. A.9. Troisième générateur de l'exemple 3.3.3

Annexe B

Résumé du guide de LatMRG

Le guide d'utilisation de LatMRG contient toute l'information permettant d'utiliser la librairie et l'exécutable. Le logiciel LatMRG est disponible à l'addresse https://github.com/savamarc/LatMRG, alors que son guide d'utilisation se trouve à l'addresse https://savamarc.github.io/LatMRG/. LatMRG est un logiciel libre et à code source ouvert distribué avec la license Apache 2.0.

Contenu du guide

Le guide contient les éléments suivants.

- Un résumé du contenu et des cas d'utilisation du logiciel intégré à la page d'accueil.
- Un résumé, relativement complet, mais sans les raisonnement et les démonstrations, du contenus présenté des la chapitres 1 et 2.
- Les instructions pour compiler le programme et la librairie, incluant une liste des dépendances.
- Une description détaillée de l'usage des programmes exécutables. Cela comprend notamment une liste de tous les tags XML définit par LatMRG, une présentation de l'organisation des fichiers de configuration et quelques exemples.
- Un tutoriel qui présente l'utilisation et les fonctions les plus importantes de la librairie.
- Une documentation complète de l'interface de programmation de LatMRG, à la fois pour les classes et pour les fonctions générales disponibles avec la librairie.

Le guide est rédigé en anglais et la documentation de l'API est générée automatiquement à partir des commentaires dans le code avec Doxygen.