UNIVERSITÉ DE MONTRÉAL

GÉNÉRATION DE NOMBRES PSEUDO-ALÉATOIRES SUIVANT UNE DISTRIBUTION NON-UNIFORME PAR CIRCUITS INTÉGRÉS PROGRAMMABLES

TAREK OULD BACHIR
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(MICROÉLECTRONIQUE)
AOÛT 2008

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

GÉNÉRATION DE NOMBRES PSEUDO-ALÉATOIRES SUIVANT UNE DISTRIBUTION NON-UNIFORME PAR CIRCUITS INTÉGRÉS PROGRAMMABLES

présenté par: OULD BACHIR Tarek
en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de:

M. DAVID Jean-Pierre, Ph.D., président
M. SAWAN Mohamad, Ph.D., membre et directeur de recherche
M. BRAULT Jean-Jules, Ph.D., membre

Science is the belief in the ignorance of experts.

– Richard Feynman

REMERCIEMENTS

Les remerciements qui suivent s'adressent à toutes ces personnes qui, de près ou de loin, m'ont aidé à faire d'une idée griffonnée sur une feuille de papier un sujet de recherche sérieux. Aussi, une pensée particulière est adressée au professeur Sawan, mon directeur de recherche, qui m'a donné la liberté de poursuivre mes chimères avec l'assurance de sa patience, de son appui intellectuel et matériel. C'est grâce à son ouverture d'esprit aux idées nouvelles et grâce aux moyens dont dispose son laboratoire Polystim que ce travail a pu être mené à bien et je ne trouverai jamais les mots pour exprimer toute ma gratitude.

J'aimerais aussi adresser mes remerciements aux professeurs Jean-Pierre David et Jean-Jules Brault qui, en plus d'accepter de payer de leur temps et de siéger sur mon jury, ont eu l'amabilité de me gratifier de leurs conseils et suggestions durant la poursuite de mes travaux. Leurs contributions informelles à elles seules suffisent pour me laisser penser que le monde académique a encore pour lui la force du travail collégial qui manque tant à l'industrie. Qu'ils sachent à quel point j'ai pu estimer leur aide précieuse et apprécié que nos discussions franches et honnêtes n'aient jamais été précédées par la signature de NDA...

J'aimerais aussi remercier les étudiants stagiaires qui ont contribué par leurs efforts à donner chair à certaines technicalités de ce travail. Je pense particulièrement à Laurent Faniel, Majed Benmahfoudh, Jonathan Boisvert et Laurent Wacker. Leur aide a été substantielle et les citer ici va de soi.

J'aimerais remercier les étudiants du laboratoire Polystim qui ont su me faire une place dans le cercle des étudiants du GRM. Ils ont su créer une atmosphère propice à rendre moins gris les couloirs des bibliothèques et instaurer un climat de camaraderie chaleureuse de par leur vie sociale active. Je pense particulièrement à Jonathan Coulomb, dont l'érudition et l'articulation scientifique m'ont plus que bénificié, je pense à Amer-Elias Ayoub, lui qui garde le sourire en toutes les circonstances (et on sait parfois lesquelles !)

et qui se les dore aujourd'hui à Antibes, je pense à Félix Chénier, lui qui n'a pas rechigné à aller se les geler pour accompagner sous la pluie un fumeur déshérité et finalement — bien sûr ! — Roula Ghannoum qui battra cette année le record des citations dans la section des remerciements.

Je ne pourrais manquer de remercier les amis et les proches qui ont largement fait la preuve de leur indéfectible appui. À chacun d'eux revient une petite phrase qui ponctue mon parcours : Guillaume Pichenot qui préconisait « Fonce ! » au moment de faire le saut vers ce sujet de recherche, Imad Safi qui me raillait en me disant « Ça t'a pris trois ans et demi pour faire un bac et un Pierre Laurent pour faire ta maîtrise » , Kamila Belhocine et son interjection « Vous soutenez la maîtrise ? » et puis, finalement, ma petite Fanny Lalonde qui ne cachait pas son impatience avec son inoubliable « J'ai hâte que ça finisse ». À quoi j'espère répondre aujourd'hui : « Voilà, c'est fini. »

RÉSUMÉ

L'accélération matérielle au moyen de la technologie FPGA connaît un intérêt croissant dans le domaine du calcul à haute performance. Cet intérêt est motivé par les possibilités dynamiques de ces dispositifs et par leur capacité à paralléliser les calculs. Dans le cas des méthodes de Monte Carlo, il a été démontré que les FPGA permettent d'accélérer les calculs par plusieurs ordres de grandeurs. Ce fait a stimulé l'activité de recherche portant sur les architectures matérielles des générateurs de nombres aléatoires issus de distributions non-uniformes qui, jusqu'alors, n'avaient été que marginales.

La génération de distributions non-uniformes en matériel comporte plusieurs défis, principalement dus aux difficultés que posent l'évaluation de fonctions transcendentales telles que le logarithme, l'exponentielle et les fonctions trigonométriques. Les techniques qui ont cours aujourd'hui tentent de câbler les algorithmes connus et d'optimiser leur implémentation sur FPGA par des artifices de calcul tels que l'interpolation polynomiale, le recours aux architectures Cordic ou encore l'approximation linéaire à base d'une segmentation non-linéaire.

Le présent travail propose d'explorer un nouvel algorithme de génération des distributions non-uniformes qui part du principe que la génération d'une variable aléatoire peut se faire un bit à la fois. Cette méthode prend tout son sens quand on songe que tous les bits peuvent être générés simultanément dans un circuit numérique. Nous posons alors le modèle mathématique associé à cet algorithme et mesurons l'étendue de ses capacités. De là, nous aboutissons à une architecture matérielle générale et universelle que nous déclinons pour les distributions normale et exponentielle. Nous étudions alors le comportement empirique sur FPGA de ces générateurs, tant du point de vue des propriétés statistiques que des critères qualitatifs telle que la corrélation sérielle, le tout avec des résultats concluants.

ABSTRACT

Hardware acceleration by means of FPGA technology is of growing interest to the realm of High Peformance Computing. This interest is justified by the dynamic possibilities of these devices and their capacity to parallelize computation. Numerous work have been recently done providing positive insights that Monte Carlo methods can be enhanced by the resort to FPGAs – from a speed performance point of view. This fact has stimulated the research for hardware non-uniform variate generators which hitherto had only been marginal.

Hardware non-uniform variates generation comprises several challenges, mainly the evaluation of transcendental functions such as the logarithm, sine, cosine and the exponential functions. The techniques suggested up to now try to wire the known algorithms and to optimize their implementation on FPGA by digital methods such as polynomial interpolation, Cordic implementation and non-linear segmentation.

We rather suggest an investigation path towards an algorithm generating the non-uniform variate one bit at a time. The algorithm is hardware-dedicated since all bits can be generated in parallel in a digital circuit. We thus introduce the mathematical model associated with this algorithm and measure the extent of its application. We then suggest a generic and universal architecture that we optimize for the normal and exponential distributions. The empirical behavior of our design is implemented on FPGA and considered as well from the statistical point of view as of qualitative criteria such as the serial correlation, the whole with conclusive results.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES ANNEXES

LISTE DES NOTATIONS ET DES SYMBOLES

α_i : pente de la fonction nodale du nœud i de la normale

β_i : ordonnée en zéro de la fonction nodale du nœud i de la normale

φ_i : fonction nodale

λ : paramètre de la distribution exponentielle

μ : espérance de la distribution normale

ρ : période d'un générateur aléatoire

ρ_a : ratio de bits uniformes sur les bits non-uniformes

σ : écart-type de la distribution normale

a_i : état du nœud A_i d'un réseau bayésien

$\boldsymbol{a}$: état du réseau bayésien

$\hat{\boldsymbol{a}}_i$: état des parents du nœud A_i d'un réseau bayésien

A_i : nœud i d'un réseau bayésien

$f(x)$: fonction de densité d'une loi de probabilité

$F(x)$: fonction cumulative d'une loi de probabilité

$g(x)$: fonction de densité d'une loi de probabilité majorante

h : pas de discrétisation de la variable aléatoire non uniforme

$\mathcal{H}_0$: hypothèse zéro d'un test statistique

i : indice des nœuds d'un réseau bayésien

M^{-1} : facteur de normalisation d'un générateur uniforme

$\mathcal{N}$: distribution normale

N : taille d'une population aléatoire

u : variable aléatoire uniforme

U : distribution uniforme

x : variable aléatoire non-uniforme

X_n : état d'un générateur uniforme à la séquence n

INTRODUCTION

L'intérêt scientifique de disposer de séquences de nombres aléatoires est contemporain au développement des calculateurs sophistiqués que sont nos ordinateurs modernes. Les premières machines (l'EDVAC par exemple) étaient lentes et peu performantes en comparaison des standards d'aujourd'hui. Aussi, effectuer des calculs complexes était une tâche tellement difficile que l'option de recourir à des processus stochastiques fut considérée attentivement pour éviter les écueils qui en découlaient. Aussi saugrenue que l'idée puisse sembler, confier les sciences exactes au hasard allait donner des résultats plus qu'impressionnants.

Le problème de la complexité des calculs s'était posé avec acuité aux chercheurs du laboratoire national de Los Alamos dans le cadre du projet entourant la bombe atomique et connu sous son nom de code : « Projet Manhattan ». Les physiciens tentaient de simuler des phénomènes au niveau subatomique pour mettre au point la bombe ; mais ils ne disposaient que de machines peu performantes et difficiles à programmer. Les esprits les plus brillants furent engagés et mis à contribution — parmi lesquels Fermi, Metropolis et von Neumann — et ces derniers proposèrent une méthode de calcul stochastique qu'ils baptisèrent méthode Monte Carlo en référence à la principauté européenne et ses célèbres casinos où les jeux de hasard sont légion.

Au fur des années, les méthodes Monte Carlo furent étudiées en profondeur et se répandirent rapidement de la physique à d'autres domaines scientifiques. On les retrouve aujourd'hui dans des branches du savoir aussi diverses que la biologie, l'ingénierie, l'économie et la finance entre autres. Les méthodes Monte Carlo exploitent le hasard pour améliorer les performances des algorithmes de calcul tout en réduisant leur complexité. Mais pour ce faire, les méthodes Monte Carlo ont besoin de générateurs de nombres aléatoires rapides, fiables et statistiquement robustes.

On distingue deux grandes familles de générateurs aléatoires :

1. Les générateurs uniformes ;
2. Les générateurs non-uniformes.

Les premiers visent à générer des nombres de la distribution uniforme $U(0,1)$. Ils forment la base fondamentale de toutes les théories stochastiques. Aussi, un corpus scientifique important leur est consacré. Les seconds sont désignés par une définition négative car ils visent à générer toutes les distributions autres que l'uniforme. Ces générateurs exploitent généralement les générateurs uniformes pour parvenir à leur fin[1].

Les générateurs uniformes étant essentiels à nombre d'utilisations scientifiques et industrielles, des implémentations matérielles ont été très tôt étudiées et mises en œuvre avec succès. À l'inverse, les nombres issus des distributions non-uniformes ont toujours été générés en logiciel. Néanmoins, la réduction de la taille des circuits numériques intégrés et la croissance de leurs capacités ont récemment stimulé la recherche portant sur les implémentations matérielles des générateurs de distributions non-uniformes. Une des motivations importantes de ces travaux est l'accélération matérielle des méthodes Monte Carlo au moyen de FPGA, particulièrement pour des applications en finance, en physique et en biologie.

L'approche unanimement admise pour réaliser ces générateurs non-uniformes consiste à « câbler » les algorithmes logiciels connus. Mais les FPGA imposent des restrictions quant à la taille de la mémoire et à la représentation des réels qui contraignent notablement cette approche. Une autre difficulté de cette avenue réside dans l'évaluation efficiente des fonctions transcendantales, inhérentes au fonctionnement des algorithmes de génération des distributions les plus populaires.

[1] Certains algorithmes de génération de nombres aléatoires suivant une distribution non-uniforme n'ont pas recours à des générateurs uniformes. Nous considérerons la question plus en détail au chapitre 1.

Le travail qui suit tente d'ouvrir le champ à une nouvelle avenue de recherche dans l'algorithmie des générateurs de distributions non-uniformes, particulièrement orientée vers la mise en œuvre matérielle. Partant de la prémisse que les nombres aléatoires puissent être générés un bit à la fois, nous examinons le modèle mathématique sous-jacent et proposons une architecture matérielle universelle pour les générateurs non-uniformes dont nous proposons d'étudier les capacités et les limites.

Le restant de ce mémoire s'articule comme suit. Au chapitre 1, nous passons en revue les notions de base de la génération aléatoire de nombres suivant les distributions uniformes et non-uniformes. Au chapitre 2, nous présentons une revue de la littérature moderne afin de bien cerner l'état de l'art de la génération aléatoire matérielle, énoncer les motivations qui ressortent des travaux étudiés et bien comprendre les défis qui se posent aujourd'hui. Au chapitre 3, nous détaillons notre approche algorithmique et développons le modèle mathématique qui s'y rattache. Le chapitre 4 est consacré à l'architecture universelle tirée de notre algorithme et son application aux distributions normale et exponentielle. Le chapitre 5 détaille la mise en œuvre physique de l'architecture des distributions normale et exponentielle et discute les résultats expérimentaux obtenus à la lumière des techniques vues au chapitre 2. Nous terminons par une conclusion générale où nous suggérons également quelques pistes à venir.

CHAPITRE 1

NOTIONS DE BASE DES GÉNÉRATEURS ALÉATOIRES

1.1 Préambule au propos

Le sujet de la génération de nombres aléatoires en matériel comporte deux aspects distincts : le premier concerne l'algorithmie, l'autre l'architecture. Nous englobons dans l'algorithmie les notions théoriques et mathématiques impliquées dans la formulation d'un modèle de calcul, les critères statistiques à même de qualifier ces modèles et les considérations pratiques qui les entourent (l'utilisation de la mémoire par exemple). L'aspect architectural est dès lors vu comme la transposition du corpus algorithmique dans la mise en œuvre de circuits numériques, en considérant davantage les aspects matériels tels que les contraintes technologiques, les limitations techniques ou la représentation des nombres.

Pour cette raison, nous avons jugé opportun de débuter le mémoire par un survol des notions de base de la génération des nombres aléatoires. Notre désir est de faciliter la lecture des chapitres subséquents et d'explorer la littérature moderne en évitant de s'empêtrer dans les technicalités des trop nombreux algorithmes exploités par les auteurs. Le domaine de la génération des nombres aléatoires englobe en effet une pluralité de développements mathématiques, riches et complexes, auxquels on pourrait facilement consacrer des ouvrages complets. Il nous eût été difficile de prétendre à l'exhaustivité si nous l'avions ambitionné, aussi avons-nous jugé plus que raisonnable de ne couvrir de cette matière que les parties qui nous seront utiles dans la mise en application matérielle.

1.2 Introduction

La génération aléatoire de nombres issus d'une distribution non-uniforme repose sur la génération préalable de nombres indépendants et identiquement distribués (i.i.d.) suivant l'uniforme $U(0,1)$. Supposons que nous voulions générer des nombres suivant une densité de probabilité $f(x)$. Nous commencerons par observer que la fonction de densité de probabilité $f(x)$ admet une fonction de répartition[1] $F(x)$, donnée par :

$$F(x) = \int_{-\infty}^{x} f(t)dt \tag{1.1}$$

En supposant que nous disposions de nombres i.i.d. u suivant l'uniforme $U(0,1)$, il devient aisé de générer x en prenant $x = F^{-1}(u)$ [15]. Aussi, la génération d'une distribution non-uniforme $f(x)$ est une tâche simple, en autant que l'on dispose d'un générateur uniforme fiable et que l'on puisse surtout évaluer l'inverse de la fonction de répartition $F^{-1}(u)$.

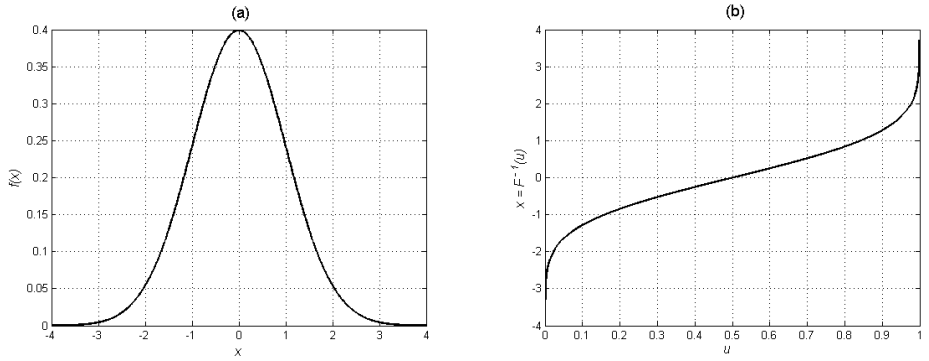


FIG. 1.1 Illustration de la méthode de l'inverse : (a) Fonction de densité $f(x)$, (b) Inverse de la fonction de répartition $x = F^{-1}(u)$.

[1] Dans le cas où x est une variable discrète, $F(x)$ est appelé une distribution de masse.

Cette aisance de calcul n'est pas toujours garantie et il faut alors faire preuve d'ingéniosité pour résoudre certains problèmes difficiles. Il arrive aussi que le calcul de l'inverse soit possible mais ne puisse se faire en des temps raisonnables. Considérant le fait qu'une simulation stochastique puisse impliquer la génération de milliards d'échantillons d'une distribution donnée, un gain de temps d'exécution de quelques fractions de micro-seconde par échantillon suffit à justifier le recours à des méthodes moins universelles mais aux performances meilleures.

Néanmoins, toutes les méthodes développées jusqu'ici, autant les méthodes universelles (telle que la méthode de l'inverse) ou des méthodes spécifiques (telle que la Ziggurate [41]), toutes, à quelques rares exceptions près, usent de la distribution uniforme $U(0,1)$ pour parvenir à leur fin. Aussi, la connaissance adéquate de la génération de cette dernière est importante pour le travail qui nous concerne ici. La section 1.3 couvre cette matière pour que soient bien saisies les subtilités de la génération des nombres aléatoires. La section 1.4 suit et complète cette étude en abordant la théorie de la génération des distributions non-uniformes.

1.3 Génération de la distribution uniforme

Quand on considère la génération de distributions non-uniformes, le choix d'un générateur uniforme est généralement laissé à la discrétion de l'usager. Aussi, la disponibilité d'un générateur uniforme idéal est souvent considérée comme une hypothèse de travail raisonnable. Ce qui importe avant tout au concepteur, c'est l'algorithme qui permet de faire le pont entre des échantillons i.i.d. de l'uniforme et la distribution visée. Cependant, les considérations rattachées à la génération de la distribution uniforme dans un environnement numérique (autant logiciel que matériel) se généralisent aisément à la génération de distributions non-uniformes. Nous tenterons par conséquent de couvrir cette matière de façon à ce qu'elle éclaire au mieux les problématiques que nous relèverons par la

suite.

La distribution uniforme $U(0,1)$ est définie par sa fonction de densité de probabilité :

$$f(u) = \begin{cases} 1 & \text{Si} \quad 0 < u < 1 \\ 0 & \text{Autrement} \end{cases} \tag{1.2}$$

Pour générer u, il est d'usage de générer un entier X entre 1 et $M-1$ (on choisit M très grand) et d'obtenir u par la fraction $u = X/M$ [24]. La question qui reste posée est : comment générer X de manière équiprobable sur l'intervalle des entiers de 1 à $M-1$?

Une technique célèbre dont l'énoncé de base date de 1949 [24] consiste à utiliser la séquence des entiers X_n obtenus par la récurrence :

$$X_{n+1} = (aX_n + c) \bmod M, \quad \text{n} \geq 0 \tag{1.3}$$

L'élément X_0 qui débute la séquence est appelé la graine. Comme on peut le remarquer, une telle séquence n'a rien d'aléatoire : dès qu'un nombre se répète dans la séquence, un cycle se produit et introduit une période. Néanmoins, pour certaines conditions sur a, c, M et X_0, le générateur peut exhiber un comportement quasi-aléatoire (l'ellipse *séquence aléatoire* est d'usage courant) qui ne le rende pas suspect à un observateur extérieur. Par exemple, le choix $a = 16807$, $c = 0$, $M = 2^{31} - 1$ et n'importe quelle graine $0 < X_0 < M$ produit une séquence de période de $M - 1$ et un comportement aléatoire intéressant. Lorsque le générateur a une période de $M-1$, on dit que sa période est maximale. La section 1.3.2 discute les critères de qualité à même de qualifier de tels générateurs.

1.3.1 Formulation générale

La grande majorité des générateurs pseudo-aléatoires repose sur l'approche fractionnaire $u = X_n/M$ présentée à la section 1.3. Si il est vrai que tous les générateurs uniformes ne fonctionnent pas sur la base d'une congruence linéaire multiplicative, ils présentent tous un comportement périodique et une fonction de transition liant l'élément X_{n+1} à son prédécesseur X_n ou à l'ensemble de ses k prédécesseurs X_n, X_{n-1}, X_{n-2} ... X_{n-k}.

Pierre L'Écuyer [26, 28] propose de définir les générateurs pseudo-aléatoires comme suit : un générateur (pseudo-)aléatoire est une structure $(\mathcal{S}, \mu, \mathcal{F}, \mathcal{U}, \mathcal{G})$ où $\mathcal{S}$ est un ensemble fini d'états, μ une distribution de probabilité sur $\mathcal{S}$ permettant de choisir l'état de départ $\mathcal{S}_0$ de la séquence aléatoire à produire, $\mathcal{F} : \mathcal{S} \rightarrow \mathcal{S}$ la fonction de transition utilisée pour passer d'un état $\mathcal{S}_i$ à l'état suivant $\mathcal{S}_{i+1}$, $\mathcal{U}$ l'ensemble image de la fonction de sortie $\mathcal{G} : \mathcal{S} \rightarrow \mathcal{U}$, faisant correspondre à chaque état $\mathcal{S}_i$ un échantillon de $\mathcal{U}$.

L'ensemble d'états étant fini et la séquence étant générée par la fonction de transition $\mathcal{F}$, un générateur uniforme possède une période ρ telle que $X_{i+\rho} = X_i$ pour tout $i \geq l$ et $l \geq 0$. La constante ρ est appelée la période de la séquence et la constante l la transitoire. Les constantes ρ et l sont des entiers et ρ ne peut excéder le cardinal de l'ensemble d'états $|\mathcal{S}|$. Les distributions non-uniformes étant générées à partir d'échantillons i.i.d. de $U(0, 1)$, la structure $(\mathcal{S}, \mu, \mathcal{F}, \mathcal{U}, \mathcal{G})$ s'applique tout aussi bien à eux. De plus, les distributions non-uniformes ainsi générées héritent en quelque sorte des propriétés du générateur uniforme qu'elles exploitent (transitoire et période). Les performances statistiques des générateurs non-uniformes sont quant à elles sujettes aux subtilités de leur conception algorithmique. Par exemple, il n'est pas impossible qu'un générateur non-uniforme — même faisant usage d'un générateur uniforme idéal — introduise une corrélation entre les échantillons qu'il produit par le fait même de sa formulation algorithmique. Nous considérons cette question avec attention plus loin.

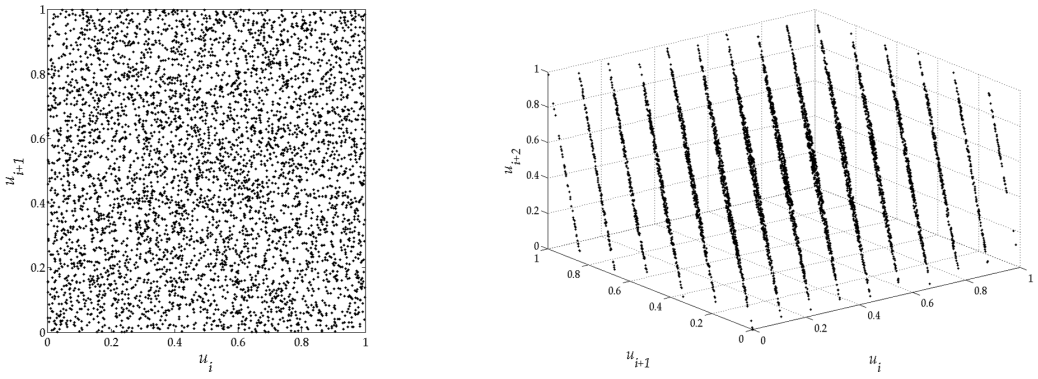
1.3.2 Critères de qualité d'un générateur uniforme

Les générateurs aléatoires étant utilisés pour des fins de simulation, une période très longue (bien supérieure à 2^{32}) est un critère important qui ne peut être écarté. Ce critère tend à devenir crucial eu égard aux fréquences de fonctionnement extrêmement élevées des systèmes numériques exploités. Ainsi, un générateur ayant une période ρ de l'ordre de 2^{32} épuiserait l'ensemble de ses états $\mathcal{S}_i$ en quelques secondes et tournerait ensuite en boucle sur un processeur cadencé à 3 GHz. De plus, les générateurs aléatoires étant utilisés comme éléments de structures algorithmiques plus importantes, il est d'usage d'exiger une implémentation compacte (consommant peu de mémoire) et simple de par son expression mathématique quand vient le temps de la traduire en termes d'effort de calcul [26]. Finalement, la répétabilité de la simulation est une caractéristique jugée importante dans le contexte de la simulation où les expériences aléatoires doivent pouvoir être vérifiées dans les mêmes conditions d'exécution. Ce facteur est un des plus importants parmi ceux qui donnent l'avantage aux générateurs pseudo-aléatoires sur de véritables générateurs aléatoires exploitant des phénomènes physiques tel que le bruit thermique [28].

Un autre critère important concerne la corrélation sérielle des échantillons consécutifs. Knuth [24] a été le premier à souligner l'importance de ce facteur et à proposer des tests de corrélation sérielle. Mais c'est à Marsaglia [39] que l'on doit la théorisation de la problématique de corrélation sérielle des générateurs de nombres aléatoires. Un cas célèbre que révéla le travail de Marsaglia est celui du générateur RANDU, disponible durant les années soixante sur les machines IBM. Le RANDU faisait usage d'une congruence multiplicative linéaire donnée par le triplet ($a = 65539$, $c = 0$, $M = 2^{31}$). La figure 1.2.a laisse croire qu'il n'existe pas de corrélation entre les échantillons consécutifs générés par le RANDU. Cependant, la figure 1.2.b, qui représente l'espace tridimensionnel rempli par les triplets de trois échantillons successifs, montre qu'il existe

bel et bien une corrélation qui incrimine le RANDU. En vérité, une structure planaire est implicitement exprimée dans la formulation mathématique du RANDU. Marsaglia avait en fait démontré que tous les générateurs à base d'une congruence linaires possèdent une structure d'hyperplan de ce type [37], c'est-à-dire que l'on retrouve toujours à l'intérieur de l'hypercube k-dimensionnel une structure d'hyperplan produite par les k-tuplets d'échantillons consécutifs.

L'Écuyer part de ce résultat et requiert dans son test qu'un générateur uniforme ait tous ses k-tuplets uniformément répartis sur l'hypercube k-dimensionnel unitaire, et ce pour tout $k > 0$. Néanmoins, la structure algorithmique des générateurs étant déterministe, le critère est appliqué en l'assouplissant par une borne supérieure donnée à k. Cette borne est définie par la structure algorithmique du générateur. Par exemple, le célèbre algorithme de Mersenne Twister [44] respecte cette condition pour tout $0 < k < 624$: il comprend en effet 624 mots de 32 bits pour mémoriser son état et exhibe une période de $2^{19937} - 1$ ($19937 = 32 \times 624 - 1$).



(a) (b)

FIG. 1.2 Nuages de points issus de 5000 échantillons consécutifs générés par le RANDU : (a) disposition 2D, (b) disposition 3D.

L'ensemble de ces critères peuvent être vérifiés de manière empirique. Il existe pour

cela des librairies permettant de qualifier un générateur aléatoire, dont la plus populaire a été proposée par Marsaglia sous le nom DIEHARD [40]. Plus récemment, L'Écuyer a proposé la librairie TestU01 [30] pour combler certaines lacunes et limitations de la librairie DIEHARD, telle que la représentation 32 bits. Néanmoins, bien que les générateurs aléatoires soient de nos jours passés au crible par ces batteries de tests, seule l'étude théorique préalable à une vérification empirique est jugée acceptable pour qualifier tout nouveau modèle [30].

1.3.3 Quelques modèles populaires

L'impératif de fournir une preuve théorique pour qualifier un générateur a tendance à populariser les modèles d'énoncé simple telle qu'une récurrence linéaire [30]. Le générateur à congruence linéaire multiplicative (CLM), exprimé par l'équation (1.3), est très répandu car il s'exprime simplement et présente d'excellentes propriétés statistiques pour certaines conditions sur le triplet (a, c, M) [40]. Les générateurs à multiples récurrences linéaires multiplicatives (MRLM) sont également très prisés, notamment en raison de la grande période qui les caractérise. Les MRLM sont caractérisés par la fonction récursive :

$$X_n = (a_1 X_{n-1} + a_2 X_{n-2} + \ldots + a_k X_{n-k}) \bmod M, \;\; \mathrm{n} \geq k \tag{1.4}$$

Pour M premier, il est possible de trouver des coefficients a_i ($1 \leq i \leq k$) garantissant une période $\rho = M^k - 1$ [24]. Il est de plus possible d'obtenir une telle période en n'utilisant que deux coefficients, ce qui réduit considérablement le nombre d'opérations arithmétiques nécessaires à l'exécution de l'algorithme. Cependant, il est difficile de concilier la simplicité de l'expression mathématique avec les performances statistiques et une solution possible est alors de considérer la combinaison de plusieurs MRLM [28].

D'autres modèles populaires à base de récurrences linéaires se fondent sur la représentation binaire des entiers. Nous les appellerons générateurs à récurrence linéaire binaire (GRLB). Les GRLB ont l'avantage d'une implémentation matérielle aisée. On les exprime par une formulation matricielle sur le corps de Galois $\mathbb{Z}/2$ — noté CG(2) — où l'addition se réduit à un XOR et la multiplication à un ET logique. En considérant que l'état du générateur X_n est contenu sur N bits, la séquence d'états est obtenue par la formule matricielle de l'équation (1.5), où **A** et **B** sont respectivement des matrices de tailles $N \times N$ et $M \times N$ ($N \gg M$) peuplées de 0 et de 1 [29]. La sortie Y_n (fonction de X_n) est ainsi exprimée sur M bits, tout comme l'est l'échantillon uniforme $u = Y_n/2^M$. Les matrices **A** et **B** sont choisies de sorte que la séquence produite ait une période maximale $\rho = 2^N - 1$. Marsaglia a démontré que pour qu'une telle structure ait une période maximale, il faut que son polynôme caractéristique $P(z) = det(z\mathbf{I} - \mathbf{A})$ soit primitif dans CG(2) [42].

$$X_n = \mathbf{A}X_{n-1}, Y_n = \mathbf{B}X_n \tag{1.5}$$

1.3.4 Implémentations matérielles des générateurs uniformes

Les LFSR (registres à décalage à rétroaction linéaire) forment une famille de GRLB dont l'utilisation comme générateur aléatoire remonte aux années soixante grâce au travail que Golombe leur a consacré [20]. La figure 1.3 illustre le schéma général de la construction d'un LFSR. Les LFSR ont une implémentation matérielle à la fois simple et compacte puisqu'un simple XOR suffit à la réalisation du chemin de rétroaction. Le nombre d'entrées du XOR est fonction du nombre de bascules utilisées et du polynôme caractéristique $P(z)$. Les manufacturiers de FPGA comme Xilinx offrent des implémentations optimisées du LFSR et des tables des polynômes irréductibles [64, 65].

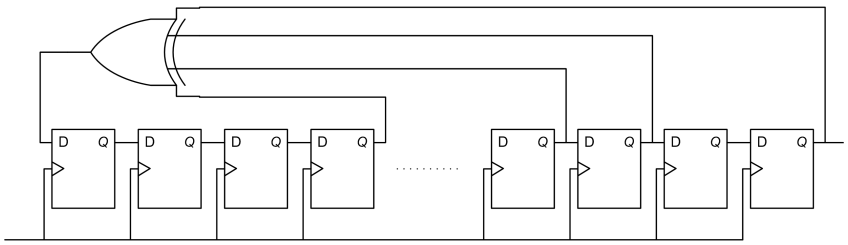


FIG. 1.3 Structure générale d'un LFSR.

L'algorithme Tausworthe [54] fait également partie de la famille des GRLB. L'Écuyer a proposé une formulation permettant une utilisation pratique de l'algorithme Tausworthe (Tausworthe combiné) dans le contexte de la simulation stochastique [27] ; la période de la séquence est $\rho = 2^{88}$ et la sortie Y_n est définie par trois mots de 32 bits :

$$Y_n = (X_{1,n} \oplus X_{2,n} \oplus X_{3,n}) \tag{1.6}$$

où, ayant $C_1 =$ 0xFFFFFFFE, $C_2 =$ 0xFFFFFFF8, et $C_3 =$ 0xFFFFFFF2, nous aurons :

$$\begin{cases} X_{1,n} = (((X_{1,n-1} \cdot C_1) \ll 12) \oplus (((X_{1,n-1} \ll 13) \oplus X_{1,n-1}) \gg 19)) \\ X_{2,n} = (((X_{2,n-1} \cdot C_2) \ll 4) \oplus (((X_{2,n-1} \ll 2) \oplus X_{2,n-1}) \gg 25)) \\ X_{3,n} = (((X_{3,n-1} \cdot C_3) \ll 17) \oplus (((X_{3,n-1} \ll 3) \oplus X_{3,n-1}) \gg 11)) \end{cases} \tag{1.7}$$

Les performances statistiques des générateurs aléatoires matériels ont récemment été testées au moyen de DIEHARD et TestU01 [43, 57]. Il en ressort que l'utilisation d'un LFSR donne de piètres résultats, alors que la solution Tausworth combiné donne des résultats très satisfaisants. Une autre approche suggérée par [43] est d'utiliser un LFSR d'une longueur $N \times M$ et d'en utiliser N-bits à chaque cycle.

1.3.5 Considérations sur la représentation machine des nombres réels

Jusqu'ici, nous avons considéré les générateurs aléatoires sous une représentation à virgule fixe implicite dû à la normalisation X_n/M. La figure 1.4 montre la modification de l'histogramme $U(0,1)$ si on utilise ce modèle avec une représentation à virgule flottante des réels. Ce type d'effets secondaires dus à l'implémentation ont très tôt été indiqués par les travaux de Marsaglia [37]. Les implémentations matérielles des générateurs aléatoires évitent cependant ces considérations à cause de la courte longueur des mots binaires et de la parcimonie dont ils font preuve dans l'utilisation des ressources matérielles. Aussi, dans le cadre du développement d'architectures matérielles, une représentation à virgule fixe est suffisante pour les besoins des applications. Les effets secondaires qui pourraient résulter d'une conversion vers une représentation à virgule flottante peuvent être considérés hypothétiquement négligeables.

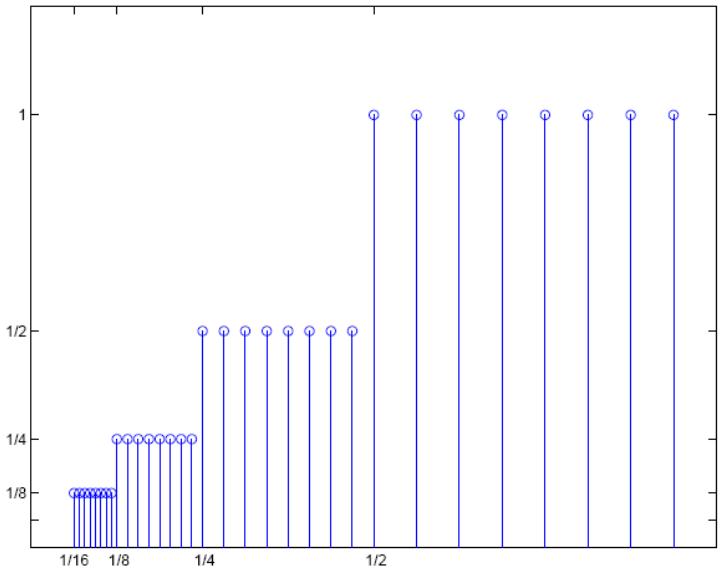


FIG. 1.4 Histogramme de l'uniforme modifié par la représentation à virgule flottante.

1.4 Génération des distributions non-uniformes

Les distributions non-uniformes peuvent être utilisées pour simuler différents comportements aléatoires. Par exemple, la distribution exponentielle est exploitée pour simuler l'intervalle de temps entre deux événements indépendants, tel que le temps entre deux appels téléphoniques, le temps entre deux requêtes adressées à une imprimante, etc.

Il existe différentes techniques pour générer ces distributions. Elles se subdivisent en deux catégories principales, nommément [15] :

1. Les méthodes universelles ;
2. Les méthodes spécifiques.

Les méthodes universelles sont des méthodes qui s'appliquent (à tout le moins en théorie) à toute distribution de probabilité. On compte quelques rares méthodes universelles que nous énumérons plus bas. Néanmoins, cette universalité de l'expression est contrebalancée par des performances variables suivant les distributions visées, l'environnement matériel où elles sont exécutées et le contexte d'application de leur utilisation. Aussi, il arrive qu'une méthode universelle applicable à une distribution en particulier soit écartée au profit d'une méthode spécifique aux performances plus attrayantes. Nous verrons des exemples où un tel choix se pose à la section 1.4.4.

Les méthodes spécifiques s'adressent quant à elles à une distribution en particulier, une famille de distributions (la famille des distributions monotones strictement décroissantes par exemple) ou encore à un cas de figure bien spécifique d'une distribution donnée (c'est le cas par exemple des distributions Beta et Gamma dont la forme varie du tout au tout en variant leurs paramètres). Il existe presque autant de méthodes spécifiques que de distributions de probabilité non-uniformes, aussi nous contenterons-nous ici de couvrir celles qui nous seront les plus utiles au chapitre 2.

1.4.1 Méthodes universelles

Nous considérons ici trois techniques universelles importantes. Les deux premières sont aussi les plus connues et les plus utilisées. La troisième est moins répandue mais nous sera utile pour articuler le modèle que nous proposons.

La première a été brièvement présentée au début de la section 1.2 — on l'appelle méthode de l'inversion. La méthode de l'inversion consiste à calculer l'inverse de la fonction de répartition $x = F^{-1}(u)$ pour générer la distribution $f(x)$, où u est issu de l'uniforme $U(0, 1)$. Certaines distributions se prêtent bien à l'implémentation suivant la méthode de l'inversion. Par exemple, la distribution exponentielle de paramètre réel $\lambda > 0$ — définie par $f(x) = \lambda e^{-\lambda x}$, $x \geq 0$ — s'applique bien à cette approche puisque l'inverse de sa fonction de répartition se calcule analytiquement ; elle est donnée par :

$$F^{-1}(u) = -\frac{\ln(1-u)}{\lambda} \tag{1.8}$$

Relevons ici que dans l'équation (1.8), il est possible de remplacer le paramètre $1 - u$ par u puisque les variables aléatoires u et $v = 1 - u$ suivent toutes deux la distribution uniforme $U(0, 1)$. Aussi, pour éviter une soustraction, la distribution exponentielle est plutôt générée par l'équation (1.9) — c'est le cas de son implémentation dans la révision 1.1 de la librairie scientifique GSL [19].

$$F^{-1}(u) = -\frac{\ln(u)}{\lambda} \tag{1.9}$$

D'autres distributions sont difficiles à produire par la méthode de l'inversion. Par exemple, la loi gaussienne, d'espérance μ et d'écart-type $\sigma > 0$ et exprimée par

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{1.10}$$

n'admet pas d'expression analytique de sa fonction de répartition, et encore moins d'inverse analytique de cette dernière. Il est alors possible d'approximer numériquement l'inverse de la fonction de répartition, mais cela ne peut être fait qu'au coût d'interpolations difficiles ou en recourant à des LUT dont la taille croît de manière exponentielle avec la résolution recherchée [15].

Une seconde technique universelle a été introduite par John von Neumann [62]. Elle est basée sur une tout autre approche, connue sous le nom de méthode de rejet. Comme son nom l'indique, cette méthode génère des échantillons avec des contraintes assouplies et en rejette une partie suite à l'application d'une contrainte plus sélective [15]. L'idée de von Neumann tire son intérêt de deux points :

1. Générer les échantillons d'une distribution simple à évaluer et produisant un grand nombre d'échantillons valides ;
2. Utiliser une fonction de discrimination tout aussi simple pour rejeter les échantillons coupables.

La méthode de rejet s'exprime comme suit : on génère x suivant une distribution $g(x)$, telle que la distribution visée $f(x)$ soit majorée par $g(x)$ à une constante multiplicative près ($f(x) < Mg(x)$, avec $M > 1$ réel). On génère alors un échantillon u suivant l'uniforme $U(0, 1)$.

– Si $u < f(x)/Mg(x)$, alors x est admis comme échantillon de $f(x)$;
– Sinon, x est rejeté et le processus de génération est poursuivi.

Knuth a fait la démonstration [25] que le ratio ρ_a d'échantillons générés sur les échantillons acceptés ne devrait pas dépasser 2 ($\rho_a < 2$). Cette démonstration de Knuth était accompagnée d'une expression générale des modèles de génération déterministes des distributions non-uniformes, basée sur la représentation binaire des nombres. Pour ce faire, Knuth partait du principe que les bits de u ont une probabilité de 0.5 de valoir 1. En construisant un arbre de décision binaire pour chaque bit à produire de l'échantillon non-uniforme x et en parcourant cet arbre de manière aléatoire avec une probabilité de 0.5 de choisir une branche d'un noeud donné, il lui était possible de compter le nombre de bits de u nécessaires à la génération de x et de chiffrer par conséquent la borne supérieure de ρ_a.

Bien qu'elle ne soit en vérité qu'une méthode de mesure de la complexité algorithmique des générateurs aléatoires, cette technique de l'arbre de décision binaire peut-être considérée comme une troisième méthode universelle. Elle sera pour nous un précédent dans la génération aléatoire basée sur le parcours un bit à la fois de la variable non-uniforme x. Nous verrons au chapitre 3 en quoi cette technique diffère de celle que nous proposons.

1.4.2 Méthodes Box-Muller

La distribution normale (ou loi gaussienne) est certainement la plus célèbre des distributions non-uniformes, mais elle est également l'une des plus utilisées. Étant donnée la difficulté de trouver pour elle une mise en œuvre simple dans le cadre des méthodes universelles (inversion et rejet), un important effort scientifique a été consenti durant les cinquante dernières années pour trouver des méthodes spécifiques à la gaussienne qui soient adéquates. La distribution normale de l'équation (1.10) a l'avantage de ne pas être

entièrement soumise à ses paramètres puisque toute variable x issue de la normale centrée réduite $\mathcal{N}(\mu = 0, \sigma^2 = 1)$ peut être manipulée pour obtenir $y \sim \mathcal{N}(\mu, \sigma^2)$ pour tous $\sigma > 0$ et μ. On applique simplement $y = \sigma x + \mu$. Aussi les algorithmes spécifiques à la distribution normale se concentrent sur la distribution normale centrée réduite :

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} \tag{1.11}$$

La plus célèbre des méthodes spécifiques à la gaussienne est due aux chercheurs Box et Muller et porte de ce fait leur nom. Elle consiste à utiliser un certain nombre d'opérations arithmétiques sur des échantillons de $U(0, 1)$ pour aboutir à produire des échantillons x de la fonction $f(x)$, donnée par l'équation (1.11) [7]. Plus exactement, en prenant u_0 et u_1, deux échantillons indépendants issus de $U(0, 1)$, Box et Muller relèvent que les variables x_0 et x_1, données par :

$$\begin{cases} x_0 = \sqrt{-2 \ln u_0} \cos 2\pi u_1 \\ x_1 = \sqrt{-2 \ln u_0} \sin 2\pi u_1 \end{cases} \tag{1.12}$$

sont indépendantes et suivent la normale $\mathcal{N}(0, 1)$. Cette expression est généralement connue comme la forme cartésienne de l'algorithme Box-Muller puisque x_0 et x_1 sont les coordonnées cartésiennes du vecteur de coordonnées polaires :

$$\begin{cases} R = \sqrt{-2 \ln u_0} \\ \Theta = 2\pi u_1 \end{cases} \tag{1.13}$$

Une forme dite polaire de l'algorithme Box-Muler existe aussi et permet quant à elle d'éviter le recours aux fonctions trigonométriques qui peuvent être coûteuses en temps de calcul [15]. Ainsi, en prenant les deux échantillons u_0 et u_1 de $U(0, 1)$ et leur contre-

partie respectives $X = 2u_0 - 1$ et $Y = 2u_1 - 1$, uniformément réparties sur l'intervalle $]-1, 0[\bigcup]0, 1[$, alors les variables x_0 et x_1 suivent $\mathcal{N}(0, 1)$ si $s < 1$.

$$\begin{cases} x_0 = X\sqrt{-\frac{2}{s}\ln s} \\ x_1 = Y\sqrt{-\frac{2}{s}\ln s} \\ s = X^2 + Y^2 \end{cases} \tag{1.14}$$

Cette forme polaire de l'algorithme Box-Muller appartient à la famille des méthodes de rejet. Dans la pratique, la forme polaire compense le rejet (dont le ratio $\rho_a = 1.2146$) par une accélération du calcul des variables x_0 et x_1 en évitant le calcul d'un $sinus$ et d'un $cosinus$.

1.4.3 Méthode de Wallace

L'intérêt de la méthode de Wallace est qu'elle n'a pas recours à des échantillons uniformes. Pour ce faire, cette méthode se fonde sur une formulation de congruence linéaire de type :

$$x_{n+1} = \frac{1}{2}(x_{n-a} + x_{n-b} + x_{n-c} + x_{n-d}),\ d > c > b > a \geq 0 \tag{1.15}$$

En disposant d'un bassin d'échantillons de départ issus de la gaussienne, il devient possible de générer une séquence d'échantillons de la normale en recourant à la récurrence de l'équation (1.15). En vérité, cette congruence nécessite quelques modifications car sa formulation matricielle offre des valeurs propres supérieures à 1, laissant craindre une divergence de la séquence. Dans son article [63], Wallace propose d'utiliser un bassin de 4×256 échantillons (tous les échantillons sont représentés sur 32-bits) tel que chaque groupe de quatre échantillons dispose d'une moyenne du carré unitaire. De plus, le bas-

sin d'échantillons est renouvelé constamment en choisissant parmi les nouveaux échantillons générés qui sont par la suite normalisés pour les besoins de la non-divergence.

Du propre aveux de Wallace, malgré une expression mathématique facilement exécutable en logiciel, cette méthode est difficile à analyser et il est impossible de chiffrer sa période ou d'assurer théoriquement que les échantillons produits soient exempts de corrélation. Nous verrons à la section 2.3.3 comment la méthode de Wallace fut implémentée en matériel et comment ces considérations furent traitées. Notons pour finir que la méthode de Wallace s'applique aussi à la distribution exponentielle.

1.4.4 La Ziggurat

Tout comme la méthode de Wallace, la Ziggurat [41] s'applique autant à la gaussienne qu'à la distribution exponentielle. Bien que la méthode de Wallace soit plus rapide que la Ziggurat (voir [35] pour les résultats de test temporel), la Ziggurat est néanmoins privilégiée à cause des garanties théoriques qui l'accompagnent. Ainsi, la Ziggurat est largement exploitée en simulation et sert de modèle au générateur de la distribution normale accompagnant la boîte à outils statistique de Matlab [46]. De plus, la Ziggurat peut s'appliquer à l'ensemble des distributions monotones strictement décroissantes ou symétriquement (comme c'est le cas pour la normale), tout en préservant des qualités de performance remarquables.

La Ziggurat repose sur un échantillonnage dans le plan dont l'origine de l'énoncé est crédité à Knuth [24]. Plusieurs énoncés intermédiaires furent étudiés (voir [46] pour une discussion de l'historique) avant d'aboutir à l'élégance de l'expression de la Ziggurat qu'offrit Marsaglia en 2000 [41].

L'échantillonnage dans le plan repose sur le découpage de la surface sous la courbe de la distribution visée $\mathcal{C}$ en zones rectangulaires dont l'union couvre une surface $\mathcal{Z}$

légèrement supérieure à $\mathcal{C}$ ($\mathcal{Z} \supset \mathcal{C}$). L'algorithme consiste ensuite à générer un point (x, y) de $\mathcal{Z}$ et de ne retourner x que si (x, y) est situé à l'intérieur de $\mathcal{C}$ (nous avons donc affaire à une méthode de rejet sensiblement identique à celle proposée par John von Neumann [62]).

Dans le cas de la Ziggurat, les zones rectangulaires sont empilées les unes sur les autres de sorte que l'abscisse de leur coin inférieur droit coïncide avec la courbe $f(x)$ et que les surfaces soient identiques. La fonction de densité $f(x)$ étant monotone strictement décroissante, les rectangles forment une pyramide qui rappelle la forme des temples mésopotamiens éponymes, comme l'illustre la figure 1.5. La Ziggurat possède une expression simplifiée du fait que les rectangles aient la même surface. Pour les courbes $f(x)$ ayant une queue se prolongeant à l'infini (comme c'est le cas de la distribution exponentielle et de la distribution normale), la zone supposément rectangulaire du bas coïncide avec la surface restante sous la courbe $f(x)$, comme l'indique la figure 1.5.b.

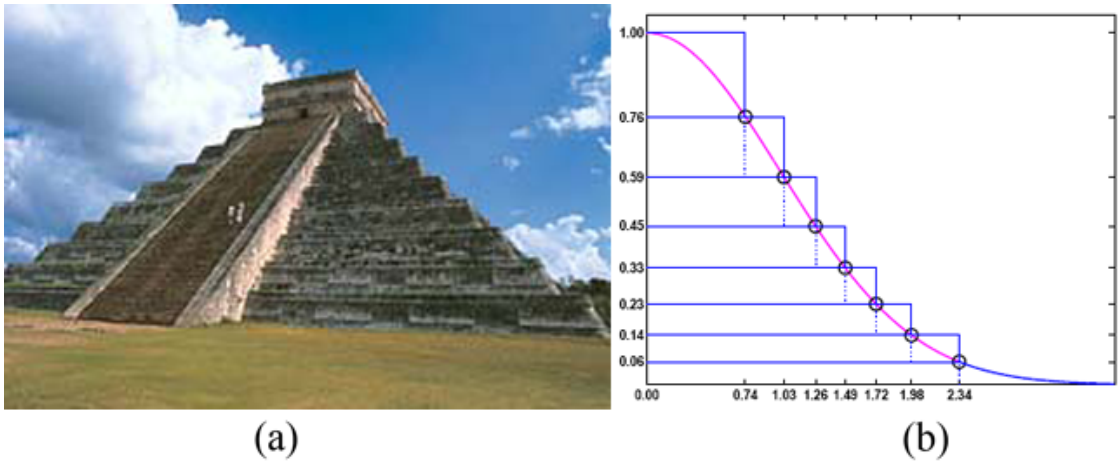


FIG. 1.5 Schéma de la construction de la Ziggurat : (a) Les temples Ziggurat ont une construction en escalier à laquelle le nom de la méthode Ziggurat réfère, (b) Schéma de la construction en escalier de la méthode Ziggurat. Images tirées de la documentation technique de Matlab [46].

Une fois le découpage en n zones rectangulaires égales établi, on dispose d'un ensemble de points d'abscisse ($0 = x_0 < x_1 < \ldots < x_n$) des coins inférieurs droits des zones rectangulaires. La Ziggurat consiste alors à choisir de façon équiprobable un rectangle i ($1 \leq i \leq n$) et de générer un x uniformément distribué sur $[x_0, x_i]$. Il est possible de ne pas générer y et de retourner directement x si $x \leq x_{i-1}$. Autrement, il faut générer y et on retiendra x si et seulement si $y \leq f(x)$.

Bien que la méthode Ziggurat soit très rapide, elle nécessite l'évaluation de $f(x)$, ce qui implique l'évaluation d'une fonction transcendantale pour les distributions exponentielle et normale.

1.5 Conclusion

Ce premier chapitre a permis de nous familiariser avec la génération aléatoire des nombres. Nous avons considéré aussi bien les générateurs aléatoires uniformes et certaines de leurs variantes matérielles que les générateurs non-uniformes — dont les implémentations matérielles seront discutées au chapitre 2. Nous avons pu apprécier les contraintes de qualité statistique que la pratique de la simulation impose et considérer les difficultés auxquelles les distributions non-uniformes contraignent. Nous sommes à présent prêts pour effectuer une revue de la littérature moderne des architectures matérielles des générateurs non-uniformes sans que la complexité des discussions algorithmiques ne nous contraignent.

CHAPITRE 2

REVUE DE LA LITTÉRATURE PERTINENTE

2.1 Introduction

Du chapitre 1, nous retiendrons que la grande majorité des algorithmes de génération des distributions les plus populaires recourent à des fonctions transcendantales. Ces opérations mathématiques ne sont pas évidentes à réaliser une fois transposées au monde de la microélectronique numérique. Les récentes recherches sur les architectures matérielles de générateurs non-uniformes ont principalement tenté d'optimiser cette partie du calcul sans réellement songer à explorer de nouvelles techniques de génération des distributions non-uniformes, éventuellement plus appropriées à l'exécution sur puce — où le parallélisme peut apporter une plus-value non négligeable.

Nous allons commencer par considérer les motivations qui ont stimulé la recherche dans ce domaine. Bien cerner le contexte d'utilisation nous permettra de mieux situer le cadre de nos travaux et d'articuler correctement les exigences en termes de distributions cibles, de vitesse d'exécution et d'espace sur puce par générateur. Nous présenterons par la suite le détail des travaux récemment publiés de sorte à brosser un portrait global de l'état de l'art. Nous verrons de cette façon quelles ont été les embûches rencontrées par les différentes équipes de recherche et les solutions que ces dernières ont privilégiées. Nous terminerons par une revue des techniques empiriques de qualification d'un générateur aléatoire non-uniforme et discuterons les différences qui les caractérisent, de façon à mieux nous positionner.

2.2 Contextes d'utilisation

Le recours aux générateurs de distributions non-uniformes en matériel se rencontre de plus en plus dans des domaines d'application aussi divers que la physique [21, 52], la biologie [49, 67], la finance [6, 55, 68], la testabilité sur puce des systèmes numériques [1, 2], l'intelligence artificielle [8, 23] ou la télécommunication [16, 17]. Certaines de ces applications sont plus exigeantes quant aux performances statistiques alors que d'autres considèrent avec plus d'attention la surface d'utilisation, la vitesse d'exécution ou les erreurs de quantification.

Par exemple, la caractérisation des canaux de communication numérique a recours à des générateurs de la distribution normale ayant une très longue période et pouvant couvrir un large spectre de valeurs possibles. Lee et al. [33, 34] proposent par exemple que le générateur couvre x de -8σ à 8σ et puisse générer plusieurs milliards d'échantillons pour couvrir correctement la queue de la distribution dans l'intervalle $6\sigma - 8\sigma$.

Dans le cas des applications où la visée première est le test et la caractérisation sur puce des systèmes numériques, c'est davantage la surface d'utilisation qui est mise de l'avant et le débit du générateur[1] est conséquemment réduit au profit de l'utilisation éventuelle de multiples instances concurrentes [1, 2].

À l'opposé, les applications dans le domaine de la simulation stochastique (méthodes Monte Carlo), employant l'accélération matérielle au moyen de FPGA, visent davantage une conjonction de vitesse d'exécution et de précision statistique de la distribution. On n'hésite pas dans ce cadre de la pratique scientifique à utiliser des ressources spécialisées des FPGA [6, 55, 68][2].

[1] Calculé comme le nombre d'échantillons par seconde.

[2] Les blocs spécialisés font référence ici aux blocs tels les multiplieurs, les blocs RAM et les blocs DSP que l'on retrouve gravés en dur dans les FPGA de nouvelles génération tels ceux de la gamme de Virtex des FPGA de Xilinx.

2.2.1 Héritage scientifique de l'arithmétique stochastique

Les premiers travaux faisant un usage prédominant de la probabilité dans les signaux digitaux remontent à Gaines [18] qui fut le pionnier de l'arithmétique stochastique. Dans les années soixante où Gaines publiait ses premiers travaux, les ressources matérielles étaient précieuses et la taille immodérée des composants tels que les additionneurs et les multiplieurs était un réel souci pour les praticiens qui voulaient mettre en œuvre nombre de ceux-là dans des systèmes numériques parallèles.

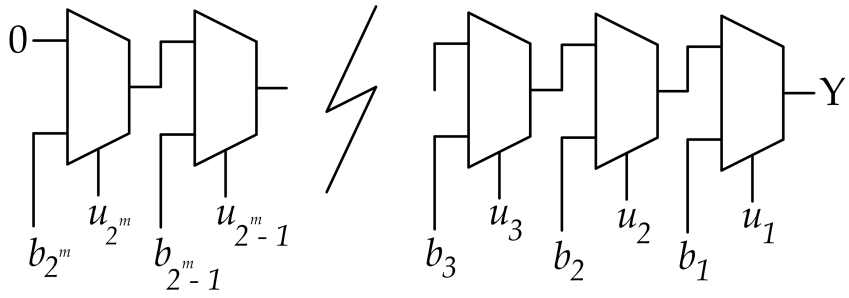


FIG. 2.1 Variante moderne du générateur de Bernoulli proposé par Gaines.

L'idée de Gaines partait du constat que si on considérait deux événements dichotomiques $(0/1)$ indépendants de probabilités p_1 et p_2, alors la conjonction des deux événements a une probabilité p_c, donnée par le produit $p_1 \cdot p_2$. Pour Gaines, cela signifiait qu'en effectuant une transposition des réels de $\mathbb{R}$ dans l'intervalle $[0, 1]$, la multiplication était réalisable au moyen d'une simple porte ET.

Pour ce faire, il fallait être en mesure de convertir une valeur numérique de $[0, 1]$ en un signal digital stochastique. Gaines propose alors le premier générateur matériel de distribution non-uniforme [18] : la distribution de Bernoulli. La version moderne de ce générateur telle que publiée dans [8] est présentée à la figure 2.1.

La sortie Y $(0/1)$ du générateur est caractérisée par une probabilité de valoir 1 que donnent les signaux d'entrée des multiples multiplexeurs cascadés, à savoir $P(Y = 1) = 0.b_1b_2b_3 \ldots b_{2^m-1}b_{2^m}{}_{(2)}$. Les signaux de commande $(u_1u_2u_3 \ldots u_{2^m-1}u_{2^m})$ quant à eux ont une probabilité de 0.5 de valoir 1 et peuvent par conséquent être tirés d'un LFSR ou de tout autre générateur uniforme.

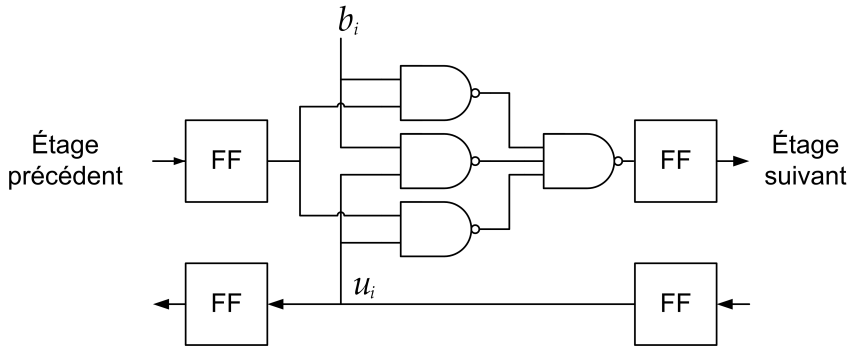


FIG. 2.2 Cellule constitutive de la variante du générateur matériel de la distribution de Bernoulli, due à Shawe-Taylor.

L'arithmétique stochastique a suscité beaucoup d'intérêt auprès de la communauté dite connexionniste de l'intelligence artificielle [8]. Dans le domaine des réseaux de neurones, l'implémentation matérielle est un défi important à cause du grand nombre de multiplications nécessaires par neurone. L'équipe de Shawe-Taylor proposait au début des années 1990 des architectures stochastiques de telles machines neuronales [60, 23, 53]. De ce groupe, van Daalen et al. proposèrent une version pipelinée du générateur de Gaines afin d'en réduire le long chemin critique qui le handicapait [61]. Chaque multiplexeur de la figure 2.1 y était remplacé par la cellule registrée de la figure 2.2. En plus de permettre un fonctionnement à haute fréquence, ce générateur a l'avantage de réduire la corrélation induite par les bits du LFSR en inversant les chemins de propagation des données et des échantillons uniformes.

2.2.2 Nouvelles avenues

Comme nous avons pu le voir, la génération matérielle des distributions non-uniformes a des applications dans des domaines très divers. Parmi les plus prometteuses, on citera l'accélération matérielle des algorithmes de simulation stochastiques telles que les méthodes Monte Carlo. On retrouve ainsi un intérêt partagé dans des domaines d'application aussi divers que la physique [21], la biologie [49, 67] et la finance [6, 55, 68].

L'idée d'une implémentation matérielle des méthodes Monte Carlo n'est pas nouvelle et date de près de quinze ans [47, 48] mais elle suscite un véritable engouement aujourd'hui. L'accélération matérielle réussie de la génération des nombres aléatoires [4, 45] fut un premier pas vers l'intérêt porté à de telles applications. Les capacités de reconfiguration accrues de ces coprocesseurs que peuvent devenir les FPGA ont abouti à créer un espoir réaliste et réalisable [12, 58, 68]. D'ailleurs, ce n'est plus uniquement une lubie de chercheurs que de rehausser les capacités calculatoires des ordinateurs puisque l'industrie a investi le terrain de cette avenue, notamment par la commercialisation d'ordinateurs de calcul à haute performance munis de coprocesseurs à base de FPGA, tels que ceux de Cray, SRC Computers, DRC Computers ou SGI Systems [21].

Dans le domaine des méthodes Monte Carlo où les algorithmes sont aisément parallélisables, c'est un véritable engouement qui en a résulté, notamment dans les domaine d'application bien subventionnés tels que le secteur bancaire et la médecine. Bien qu'exploitant souvent une méthodologie ou approche algorithmique louable [49, 67], voire même admirable dans la virtuosité architecturale dont elles font preuve [12, 56, 58, 68], ces applications paient par trop souvent le prix fort dans la consommation des ressources des FPGA. Des travaux de recherches marginales [1, 2, 16, 17] tentent encore de concilier les besoins d'économie de ressources qui s'imposent traditionnellement aux générateurs de nombres aléatoires [28], mais sans véritable succès.

2.2.3 Distributions visées

Les distributions visées par les implémentations matérielles dépendent grandement des applications qu'elles servent ; suivant ces applications, les critères de conception peuvent différer du tout au tout. On retiendra ici que seules quelques rares distributions ont mérité des tentatives d'implémentation matérielle.

TAB. 2.1 Récapitulatif des distributions implémentées en matérielles

Distribution	Fonction de densité	Fonction de répartition	Inverse analytique	Architecture matérielle
Normale	$\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{(x-\mu)^2}{2\sigma^2}}$	N/A	N/A	[1], [3], [5], [10] [13], [16], [34], [35] [56], [58], [69]
Exponentielle	$\lambda e^{-\lambda x}$	$1-e^{-\lambda x}$	$-\frac{\ln(1-u)}{\lambda}$	[10], [11], [13] [56], [58], [59]
Bernoulli	$P(x=1)=p$	N/A	N/A	[11], [18], [61]
Log-normale	$\frac{1}{x\sigma\sqrt{2\pi}}e^{-\frac{\ln^2(x-\mu)}{2\sigma^2}}$	N/A	N/A	[10], [56], [58]
Rayleigh	$xe^{-\frac{x^2}{2\sigma^2}}$	$1-e^{-\frac{x^2}{2\sigma^2}}$	$\sqrt{-2\sigma^2\ln(1-u)}$	[13]
Weibull	$\frac{k}{\lambda}(\frac{x}{\lambda})^{k-1}e^{-(\frac{x}{\lambda})^k}$	$1-e^{-(\frac{x}{\lambda})^k}$	$\lambda(-\ln(u))^{-k}$	[56], [58]

La table 2.1 récapitule les récentes implémentations matérielles de distributions non-uniformes, indiquant au passage celles qui admettent une inverse analytique de la fonction de répartition. La distribution normale est certainement celle qui a attiré l'attention du plus grand nombre, la raison étant explicable par sa très grande utilisation dans le domaine de la simulation. Elle est suivie par l'exponentielle en raison notamment de la simplicité de l'expression de l'inverse de sa fonction de répartition. Les implémentations

matérielles des autres distributions sont marginales. La distribution de Bernoulli mise à part, ces architectures s'inscrivent souvent dans le cadre de travaux ayant pour prétention l'implémentation de générateurs universels. Les distributions citées ne servent à toute fin utile que de preuve de concept. C'est le cas notamment de la distribution de Weibull [56, 58], générée pour des valeurs de paramètres totalement arbitraires[3], ou encore le cas de la distribution de Rayleigh traitée dans le contexte d'un prétendu générateur universel sans justification valable [13]. Nous aurons raison dans ces conditions de privilégier l'étude des deux distributions les plus matures : la normale et l'exponentielle.

2.3 Architectures matérielles des générateurs de nombres aléatoires

Nous allons nous pencher dès à présent sur les différentes avenues explorées par les chercheurs pour implémenter en matériel des générateurs de distributions non-uniformes. Nous commençons par aborder à la section 2.3.1 les architectures que nous appellerons universelles. Les architectures universelles engloberont pour nous tout à la fois les générateurs dont la conception architecturale s'applique de manière universelle mais qui n'ont mérité que des applications à des cas typiques, et les générateurs dont l'algorithme de génération est effectivement universel et qui permettent éventuellement une reconfiguration en temps d'exécution (*runtime reconfiguration*).

Nous nous concentrerons par la suite aux distributions normale et exponentielle. Dans le cas de la distribution exponentielle, nous procéderons à une revue des implémentations matérielles existantes et des méthodes utilisées pour évaluer la fonction logarithmique présente dans l'inverse de sa fonction de répartition. Nous nous attarderons dans le cas de la distribution normale aux approches explorées pour implémenter les algorithmes

[3] Ces paramètres sont $\lambda = 5.4$ et $k = 12$, mais ne sont pas donnés dans les articles cités. Contacté par courriel, l'auteur disait "*I've no idea why I chose those parameters, I probably just thought that it looked like a nice curve.*"

vus aux sections 1.4.2, 1.4.3 et 1.4.4. Nous verrons également comment les questions de quantification et d'économie de ressources matérielles ont été considérées par les différents auteurs. Finalement, nous terminerons par une revue de l'utilisation des générateurs uniformes dans ces architectures.

2.3.1 Architectures universelles

L'utilisation de générateurs de distributions non-uniformes devenant de plus en plus populaire, la mise au point de méthodes universelles devient un enjeux important pour éviter l'élaboration fastidieuse de l'architecture pour chaque distribution visée. Une approche simple et universelle est d'appliquer la méthode de l'inversion au moyen de LUT conservant l'ensemble des points de l'inverse de la fonction de répartition, mais les auteurs se refusent à une telle solution en raison de la croissance exponentielle de la taille de telles tables lorsque la précision arithmétique recherchée est élevée [34].

Une approche plus réaliste consiste à approximer numériquement l'inverse de la cumulative. Cheung et al. proposent d'évaluer la fonction inverse par une approximation polynomiale par parties, où les polynômes sont de degré supérieur ou égal à 1, suivant la précision recherchée [10]. Il s'avère en effet que la fonction inverse de toutes les distributions présentées à la table 2.1 se prêtent facilement à une telle approximation, comme l'illustre la figure 2.3 pour les distributions normale et exponentielle. En précalculant les intervalles définissant les segments non-linéaires en fonction de la précision recherchée, il devient possible de trouver une segmentation non-linéaire appropriée pour une distribution donnée et d'en déterminer le système de décodage d'adresse correspondant. L'évaluation du polynôme s'effectue alors en cascadant les opérations d'addition et de multiplication, de sorte à offrir une architecture pipelinée et un débit élevé.

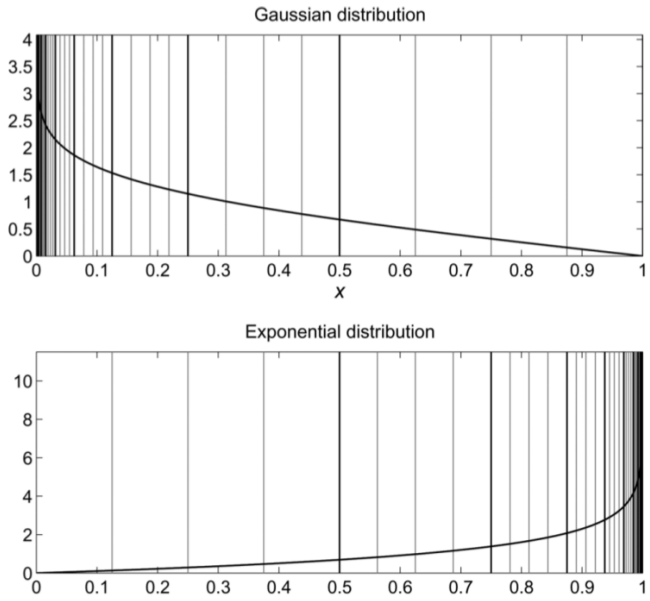


FIG. 2.3 Segmentation non-linéaire de l'inverse de la fonction de répartition telle qu'appliquée par [10] aux distributions normale et exponentielle.

Dans leur article, Cheung et al. affirment avoir réussi à automatiser le processus de conception de tels générateurs à l'aide d'un script MATLAB permettant de générer le code VHDL pour une distribution donnée [10]. Le script prend en compte le nombre de bits pour représenter la variable non-uniforme x, la précision désirée sur la représentation à virgule-fixe, l'erreur de quantification acceptée et la distribution visée. Néanmoins, la qualité du bruit généré se paie au prix d'une avidité matérielle regrettable. Par exemple, pour générer une gaussienne couvrant x (16 bits) sur l'intervalle allant de -8.2σ à $+8.2\sigma$, la surface occupée est pas moins de 550 *slices* en plus de l'emploi de 2 blocs DSP et 2 blocs RAM. Cependant, le générateur atteint une fréquence allant de 360 MHz à 480 MHz en utilisant un pipeline allant croissant de 2 à 20 étages, et une latence maximale de 65 ns (32 coups d'horloge).

David Thomas propose quant à lui de générer les distributions non-uniformes par une combinaison linéaire de distributions plus simples [56, 58], et ce avec différentes performances, tant du point de vue de l'occupation de surface sur puce, de la vitesse d'exécution que de la précision statistique. En notant $0 < w_i < 1$ ($0 < i \leq n$) les n poids utilisés pour la combinaison linéaire des n différentes distributions $g_i(x)$, l'auteur propose d'approximer la distribution $f(x)$ par :

$$f(x) = \sum_{i=1}^{n} w_i g_i(x), \sum_{i=1}^{n} w_i = 1 \tag{2.1}$$

Cela revient à échantillonner une des distributions g_i en la choisissant parmi les n disponibles suivant une probabilité définie par les poids w_i de la combinaison linéaire. Dans [56], l'auteur propose d'utiliser des distributions équiprobables $g_i(x)$ sous trois formes possibles : trapèze, triangle et rectangle. Il parvient ainsi à générer la distribution gaussienne sur 16 bits et d'atteindre une fréquence légèrement inférieure à 200 MHz. En prenant $n = 2048$, l'auteur propose un générateur occupant 411 *slices*, 3 blocs RAM et de générer x jusqu'à 5σ. L'auteur indique que chaque sigma supplémentaire demande un bloc RAM additionnel. La qualité du bruit n'est cependant pas exceptionnelle et ne peut être améliorée qu'en recourant au théorème central limite. De plus, l'auteur indique que, bien que générale, cette technique est très difficile à adapter, demandant un temps de calcul allant de quelques minutes jusqu'à plusieurs heures pour définir les distributions $g_i(x)$ nécessaires [58]. Afin d'améliorer ces performances, l'auteur propose de n'utiliser que la distribution triangulaire avec des poids w_i non égaux [58]. Cela a l'avantage de réduire le temps de calcul en vue d'une conception et améliore l'occupation de surface sur puce. Pour générer la gaussienne sur 16 bits, l'auteur utilise alors 80 bits uniformes, pose $n = 1024$ et s'en tire avec 1 bloc RAM et 137 *slices* (excluant le générateur uniforme) pour une vitesse d'exécution de 249 MHz sur un Virtex-II.

Finalement, les auteurs Cui et al. proposent un générateur qu'ils qualifient d'universel et qui permet de produire en même temps les distributions uniformes, normale, exponentielle et Rayleigh [13]. Les auteurs partent du constat que la fonction inverse $F^{-1}(u) = \sqrt{-2\sigma^2 \ln(u)}$ de la distribution de Rayleigh partage avec les équations (1.9) et (1.12) — respectivement celles de l'inverse de l'exponentielle et de la méthode Box-Muller — l'évaluation du logarithme de u. Pour calculer ce logarithme ainsi que les fonctions racine carrée, $sinus$ et $cosinus$, les auteurs suggèrent d'utiliser une architecture CORDIC optimisée. Leur solution génère les échantillons sur 16 bits et parvient à atteindre 133 MHz. Le générateur utilise deux multiplieurs et 64 bits uniformes. Bien entendu, pour des raisons de corrélation évidentes, un seul échantillon des différentes distributions est accessible à la fois. Notons néanmoins qu'aucune spécification quant à la surface d'utilisation n'est donnée par les auteurs et que la démonstration empirique du bon fonctionnement du générateur suivant les tests statistiques fut réalisée pour une population de 1 000 échantillons.

2.3.2 Génération matérielle de la distribution exponentielle

La distribution exponentielle est très utile à nombre d'applications mais n'a mérité que peu d'attention pour son implémentation matérielle. Les travaux portant sur les générateurs universels cités à la section 2.3.1 donnent uniquement un estimé des ressources nécessaires et presque aucune indication quant à la vitesse d'exécution et au débit du générateur. L'approche adoptée pour la gaussienne étant très similaire à celle proposée pour l'exponentielle dans [10], on peut estimer qu'une approximation polynomiale par parties de la distribution exponentielle utiliserait près de 500 *slices* pour une vitesse d'exécution de 230 MHz sur un Virtex II et de 370 MHz sur un Virtex IV. Les auteurs indiquent qu'il leur faut 32 bits issus de l'uniforme pour générer x dans l'intervalle 0.0 à 22.0 et 48 pour aller de 0.0 à 32.0. Finalement, une telle implémentation nécessite 2 blocs RAM et 2 DSP (ou blocs multiplieur).

Timarchi et al. [59] ont comparé différentes méthodes pouvant servir à implémenter un générateur d'exponentielle sur les FPGA Virtex de Xilinx[4]. Les auteurs ont privilégié trois méthodes de calcul du logarithme de la fonction inverse, nommément 1) une architecture CORDIC similaire à celle utilisée par Cui et al. [13], 2) une approximation polynomiale par parties utilisant une LUT et finalement 3) un algorithme d'interpolation linéaire.

L'architecture CORDIC de Timarchi et al. n'est pas des plus heureuses si on la compare aux performances obtenues par [13] avec le même algorithme (dont la résolution est cependant de moitié inférieure). Une implémentation 32 bits ne dépasse pas les 24 MHz mais donne une très bonne approximation. Les performances fréquentielles de l'approche de l'interpolation linéaire ne sont guère meilleures (25 MHz), mais cette dernière présente l'avantage d'une bonne précision pour des résolutions inférieures à 32 bits. Finalement, l'approximation par parties utilisant une LUT présente le meilleur rapport surface vs. précision pour des implémentations faisant usage de 22 bits pour représenter x et une vitesse de 190 MHz.

Un des résultats intéressants ressortant des travaux de Timarchi et al. est la croissance exponentielle de l'occupation de la surface sur puce suivant le nombre de bits pour représenter x. Alors que les trois méthodes utilisent 1000 *slices* et moins pour une représentation 16 bits, elle atteint, à une résolution de 32 bits, de $2\,000$ *slices* pour la méthode CORDIC à $10\,000$ *slices* pour l'approximation par parties et $60\,000$ *slices* pour l'interpolation[5].

[4] Il n'est nulle part spécifié quel FPGA a été utilisé pour cet article ; probablement un Virtex II ou un Virtex II Pro de forte capacité tel que le XC2V4000 ou le XC2VP100, étant donnés l'année de publication et le nombre de *slices* utilisé sur le FPGA.

[5] Il est important de relever que Timarchi et al. ont utilisé un générateur uniforme à base de LFSR, particulièrement gourmand (1 000 *slices* et plus selon nos propres estimés) pour générer 32 bits avec une période suffisante. Un autre choix eût été de s'inspirer de ce que Cheung et al. [10] ont réussi en implémentant un générateur Tausworthe combiné utilisant 141 *slices*.

Au vu de l'estimation des ressources nécessaires pour la méthode de l'inversion proposée par Cheung et al. [10] et en les comparant avec les résultats obtenus par les travaux de Timarchi et al. [59], il apparaît que :

1. L'utilisation de blocs RAM et DSP accroît considérablement les performances de vitesse du générateur d'exponentielle. Pour le même générateur, Cheung et al. [10] ont montré que la vitesse est réduite de 370 MHz à 200 MHz sur Virtex IV si il n'est pas fait usage des blocs RAM et DSP.
2. L'utilisation de blocs RAM et DSP réduit de beaucoup la surface sur puce occupée. Pour le même générateur, Cheung et al. [10] ont montré que leur design utilise plus de $1\,600$ *slices* sur un Virtex IV si il n'est pas fait usage des blocs RAM et DSP.
3. Les résutats de Timarchi et al. [59] montrent qu'une implémentation de résolution supérieure à 16 bits est très coûteuse en surface et réduit considérablement les performances de vitesse du générateur. L'utilisation de blocs DSP est dans ce cas à exclure puisque les FPGA actuels réalisent difficilement la multiplication à de telles résolutions.

Pour finir, notons que l'approche de combinaison linéaire préconisée par David Thomas dans [58] serait certainement beaucoup plus économe et performante dans les mêmes conditions. Pour générer une gaussienne, Thomas montrait que le générateur n'utilisait que 137 *slices* et 1 bloc RAM. Générer une exponentielle devrait occuper une surface similaire. De plus, cette technique a l'avantage de n'utiliser aucune fonction arithmétique telles que l'addition ou la multiplication. Mais cette approche souffre d'une limitation sur la précision qui écarte son utilisation dans le contexte de la simulation et qui laisse craindre qu'une précision de 32 bits soit tout simplement impossible à atteindre, comme nous le verrons à la section 2.3.4.

2.3.3 Génération matérielle de la distribution normale

La première implémentation sur FPGA d'un générateur de bruit gaussien qui mérite d'être mentionné est due à Boutillon et al. [5]. De ce *design* est issu le *core* actuellement commercialisé par Xilinx sous le nom XAWGN (Xilinx Additive White Gaussian Noise) [66], disponible pour les FPGA Spartan 3, Virtex-II/II Pro et Virtex IV. Le *core* occupe 480 ***slices***, 5 blocs RAM et 5 blocs multiplieur. Il couvre x sur l'intervalle allant de -4.8σ à $+4.8\sigma$, sa période est $\rho = 2^{190}$ et sa fréquence maximale sur un Virtex-II Pro est de 300 MHz [66], offrant 300 millions d'échantillons par seconde.

Dans leur article, Boutillon et al. exploitent la méthode Box-Muller sous sa forme cartésienne, telle que donnée par l'équation (1.12). Afin d'évaluer les fonctions $\sqrt{\ln u_0}$, $\sin(2\pi u_1)$ et $\cos(2\pi u_1)$, les auteurs procèdent à une quantification au moyen de LUT. Une première LUT est adressée par 12 bits uniformes pour évaluer une racine de logarithmique représentée sur 10 bits (dont 7 bits pour la partie fractionnaire) et une seconde LUT adressée par 4 bits uniformes pour générer les fonctions trigonométriques représentées sur 8 bits (dont 7 bits pour la partie fractionnaire). La quantification est ensuite corrigée par l'accumulation de plusieurs échantillons de la normale ainsi générés en utilisant le théorème central limite.

L'implémentation du design de Boutillon par Xilinx a certainement mérité un perfectionnement par le manufacturier eu égard aux performances affichées par ce dernier — de loin meilleures ne fût-ce que du point de vue de la période du générateur. Néanmoins, ce *design* n'a eu cesse d'être décrié par des auteurs tels que Lee et Zhang en raison de ses piètres performances statistiques [33, 34, 35, 69]. Il est vrai en effet que Boutillon et al. ont davantage concentré leurs efforts à évaluer les erreurs de quantifications numériques qu'à valider leur générateur au moyen de tests statistiques. Cette attitude de défiance reflétait surtout un changement de paradigme et de mentalité au sein de la communauté scientifique en ce qui a trait à la génération des nombres aléatoires sur puce.

Lee et al. [33, 34, 36] ont présenté une nouvelle implémentation matérielle de la méthode Box-Muller sur les FPGA de Xilinx. La visée première de ces travaux était l'évaluation des canaux de codage qui nécessitent de couvrir x sur un intervalle plus large que ce que permet le *core* XAWGN. Pour ce faire, l'approche de Boutillon et al. aurait eu un impact significatif sur la consommation des ressources du FPGA. Aussi, Lee et al. ont-ils choisi d'implémenter la méthode Box-Muller et de mettre en application les méthodes d'approximation polynomiale par parties développées par leur équipe dans le cadre de travaux portant sur l'évaluation de fonctions transcendantales sur FPGA [31, 32] que nous avions déjà discutées à la section 2.3.1.

En raison du coût prohibitif associé aux multiplieurs, les auteurs avaient d'abord opté pour une approximation linéaire des différents segments non-linéaires [33, 34]. Dans ces premières versions, où les échantillons de la normale sont représentés sur 32 bits, le *design* consommait 50 bits uniformes issus de LFSR de 60 registres chacun ; 32 bits servaient à évaluer la racine carrée du logarithme et les 18 bits restants servaient aux $sinus$ et $cosinus$. Pour compenser les erreurs de quantification, les auteurs recouraient au théorème central limite tout comme le faisaient Boutillon et al.. Dans sa version finale [36], le design recoure à 64 bits de l'uniforme, 48 bits servant au calcul du logarithme et les 16 bits restants aux fonctions trigonométriques. La variable x est cependant représentée sur 16 bits seulement. La première version du *design* [33] parvenait à générer x jusqu'à 4.8σ et utilisait 10% du FPGA Virtex II XC2V4000 (environ $5\ 000$ *slices*[6]), 2 blocs RAM et 8 multiplieurs, pour une fréquence de fonctionnement de 133 MHz (133 millions d'échantillons par seconde). La seconde version [34] quant à elle accroissait la précision des fonctions logarithmiques et trigonométriques en augmentant le nombre de segments non-linéaires, sans affecter ni la consommation des ressources sur FPGA ni la fréquence de fonctionnement.

[6]En soustrayant la place occupée par le générateur uniforme, il reste près de $4\ 000$ *slices* pour le générateur de gaussienne à lui seul.

Plus récemment encore, Lee et al. déclinent leur *design* sous une forme supérieure couvrant x jusqu'à 8.2σ [36]. Implémenté sur un Virtex II, c'est-à-dire le même FPGA que celui utilisé pour [33, 34], le générateur fonctionne à une cadence de 233 MHz (466 millions d'échantillons par seconde). Pour ce faire, les auteurs recourent à deux générateurs Tausworthe combiné (2×32 bits) pour les échantillons uniformes et utilisent une approximation quadratique sur les segments des fonctions transcendantales. De plus, ce *design* ne recoure plus au théorème central limite grâce à l'extrême précision des opérations arithmétiques. Ce faisant, Lee et al. occupent $1\,528$ *slices* du FPGA et utilisent 3 blocs RAM et 12 blocs multiplieur.

Fan et al. ont proposé une implémentation matérielle de le forme polaire de la méthode Box-Muller [16, 17]. Par souci d'économie de ressources, les auteurs ont tenté de ne recourir qu'à 4 bits uniformes pour générer les variables uniformément réparties X et Y de l'équation (1.14). Mais cette tentative louable fut malheureusement accompagnée de performances fréquentielles médiocres, puisque pour une cadence de 78 MHz, le générateur ne produit que 24 millions d'échantillons par seconde.

Nous ne manquerons pas de citer les travaux de Alimohammad et al. qui tentaient une implémentation compacte de la forme cartésienne de Box-Muller dans [1, 2]. Ces auteurs parvenaient à générer x jusqu'à $\pm 5\sigma$ en recourant à 54 bits uniformes issus de LFSR, 36 pour le logarithme et 18 pour les fonctions trigonométriques. La nouveauté du travail de Alimohammad et al. réside dans les algorithmes itératifs utilisés pour évaluer les fonctions transcendantales. Ce faisant, les auteurs parviennent à une fréquence de fonctionnement de 165 MHz sur un Virtex-II de Xilinx, dont ils utilisent 3% des ressources logiques (environ $1\,500$ *slices*) et 2 blocs RAM. Alimohammad et al. ont récemment publié une version améliorée de leur générateur de gaussienne [3]. Le nouveau design a été perfectionné notamment en abandonnant le choix d'utiliser les LFSR. Le nouveau *design* occupe 576 *slices*, 2 blocs mémoire, 3 multiplieurs et atteint une fréquence maximale de 269 MHz sur un Virtex II Pro.

Relevons également l'implémentation matérielle par Zhang et al. de l'algorithme Ziggurat [69]. Ce *design* utilise 891 *slices* sur un Virtex-II, 4 blocs RAM et 2 multiplieurs, pour une cadence de 170 MHz, la plus haute fréquence jamais atteinte au moment de la publication de l'article. La méthode Ziggurat faisant partie de la famille des méthodes de rejet, le débit obtenu était légèrement inférieur à 170 millions d'échantillons par seconde. La sortie x y est représentée sur 32 bits et exhibe une forte qualité statistique, mais son intervalle de couverture demeure inconnu.

Finalement, citons l'implémentation matérielle par Lee et al. de l'algorithme de Wallace [35]. Tout comme le suggérait Wallace [63], les auteurs recourent à un bassin de $4 \times 256 = 1\ 024$ échantillons conservés dans une RAM à double accès (*dual port*), de sorte que le bassin puisse être continuellement renouvelé sans en empêcher l'accès. Les auteurs évitent la corrélation que laissait craindre Wallace (voir la section 1.4.3 pour un rappel) en recourant à un générateur uniforme pour adresser les échantillons à extraire du bassin. Ce design utilise 770 *slices*, 6 blocs RAM et 4 multiplieurs sur un Virtex-II, pour une cadence de 155 MHz. La variable x est représentée sur 24 bits mais l'intervalle de couverture de x et la période du générateur demeurent inconnus.

La table 2.2 récapitule les caractéristiques des principales implémentations de gaussienne discutées jusqu'ici :

TAB. 2.2 Récapitulatif des principaux générateurs de gaussienne matériels.

Référence	[69]	[35]	[10]	[36]	[3]
Résolution (bits)	32	24	16	16	32
Fréquence max (MHz)	170	155	232	233	269
Nb de *slices*	891	770	548	1528	576
Nb de blocs RAM	4	6	2	3	2
Nb de multiplieurs 18×18	2	4	2	12	3

2.3.4 Note sur l'utilisation des bits uniformes

Maintenant que les différentes architectures matérielles ont été vues, il est intéressant de relever la grande disparité qui caractérise les différents travaux dans leur recours aux échantillons uniformes, plus exactement le nombre de bits nécessaires pour générer un échantillon non-uniforme. Rappelons que Knuth et Yao avaient démontré que le nombre d'échantillons uniformes nécessaires à la génération d'une variable non-uniforme avait une borne supérieure fixée à $\rho_a = 2$ [25]. Quand on songe que Lee et al. recourent à 64 bits uniforme pour générer une variable normale sur 16 bits [36], doit-on croire que le ratio $\rho_a = 4$ ainsi obtenu est sous-optimal ?

La réponse se trouve dans une analyse proposée indépendamment par Chen et al. [9] et Chul et al. [11]. L'analyse diffère de l'approche de Knuth et Yao en ce qu'elles n'observent pas le nombre de bits uniformes nécessaires pour générer chaque bit de la variable non-uniforme de x, mais plutôt la segmentation de l'intervalle $[0, 1)$ en zones non-linéaires.

Par exemple, supposons que x soit une variable aléatoire discrète pouvant prendre 4 valeurs différentes, tel que l'exprime l'équation (2.2) :

$$P_X(x) = \begin{cases} 2^{-32} & \text{si } x = 0 \\ 2^{-32} & \text{si } x = 1 \\ 2^{-31} & \text{si } x = 2 \\ 1 - 2^{-30} & \text{si } x = 3 \end{cases} \tag{2.2}$$

Il est alors possible de générer x en générant une variable aléatoire uniforme u dans l'intervalle $[0, 1)$ et de comparer sa valeur aux constants $c_1 = 2^{-32}$, $c_2 = 2^{-31}$ et $c_3 = 2^{-29}$.

Le générateur sera ainsi formé par trois comparateurs et respectera l'équation (2.3) :

$$x = \begin{cases} 0 & \text{si} \quad u < c_1 \\ 1 & \text{si} \quad c_1 \leq u < c_2 \\ 2 & \text{si} \quad c_2 \leq u < c_3 \\ 3 & \quad \text{autrement} \end{cases} \tag{2.3}$$

D'un point de vue numérique, l'équation (2.3) ne peut être respectée qu'en générant u avec une résolution suffisante, cette dernière étant définie par la plus petite des résolutions requises par les constantes c_i utilisées. Dans le cas de l'exemple que nous considérons, cette résolution doit être au moins de 32 bits pour que la comparaison avec c_1 soit réalisable.

Cette expression n'est qu'une traduction discrète de la méthode de l'inverse vue à la section 1.4.1. Mais en considérant que toute distribution continue est équivalente à une distribution discrète une fois transposée au domaine numérique [15], il en ressort que toutes les manipulations arithmétiques des implémentations matérielles des générateurs de distributions non-uniformes vues s'inscrivent dans ce paradigme. Aussi, plus les concepteurs tentent d'atteindre une précision élevée dans la génération de x, plus le nombre de bits uniformes requis est élevé. C'est notamment ce qui explique que dans toute implémentation matérielle de la distribution normale faisant usage de la méthode Box-Muller, le nombre de bits uniformes utilisés pour générer le logarithme est toujours plus élevé que celui nécessaire pour les fonctions trigonométriques puisque sa pente est très forte dans l'intervalle $[0, 1]$.

Ces remarques nous semblent nécessaires pour comprendre la limitation inhérente aux approches couramment adoptées dans l'implémentation matérielle des générateurs de distributions non-uniformes, notamment les approches universelles telles que celles proposées par Cheung et al. [10] ou Thomas et Luk [58].

2.4 Qualification d'un générateur non-uniforme

Dans le domaine de la statistique, il existe différentes méthodes permettant de vérifier qu'une population issue d'une expérience aléatoire respecte une loi de probabilité donnée. Généralement, on tente de classer les échantillons suivant des intervalles contigus, et on vérifie que l'histogramme empirique ainsi formé respecte celui attendu, la distribution de probabilité visée étant connue. Mais il n'existe pas de consensus absolu sur la façon dont ces mesures quantitatives sont effectuées. Les points de débat touchent notamment le nombre d'échantillons requis, la taille des intervalles, le nombre des intervalles et les quantificateurs qui en sont issus. De plus, on ne recommandera pas la même approche de mesure pour qualifier une distribution continue et une distribution discrète [14]. Dans cette section, nous verrons d'une part quelles sont les techniques reconnues et celles que l'on rencontre dans l'application. Cette revue nous permettra de définir le protocole expérimental que nous nous imposerons pour qualifier nos générateurs.

2.4.1 Technique reconnue

D'Agostino et Stephens ont consacré un ouvrage à la qualification des distributions empiriques [14] — devenu l'ouvrage de référence dans le domaine. Nombre de chapitres de ce livre sont consacrés au test du χ^2, méthode universelle s'il en est une, de la comparaison d'une distribution empirique avec l'expression mathématique de la distribution de probabilité attendue. Le test du χ^2 est effectué en évaluant le quantificateur :

$$\chi^2 = \sum_{i=1}^{k} \frac{(n_i - Np_i)^2}{Np_i} \tag{2.4}$$

où n_i est le nombre d'éléments dans la classe i, p_i la probabilité d'appartenance à cette classe telle que définie par l'expression mathématique de la distribution, N est la taille

de la population empirique et k le nombre de classes utilisées. Il est d'usage de choisir k de sorte qu'aucune classe ne possède moins de 5 éléments. D'Agostino et Stephens préconisent plutôt de diviser l'axe des x suivant $k = 2N^{\frac{2}{5}}$ intervalles équiprobables.

Le quantificateur χ^2 variant suivant la taille N de la population aléatoire, il est suggéré de le ramener à une mesure de probabilité de confiance (*p-value*) donnée par la distribution du χ^2 de $k - 1$ degrés de liberté. Là où il y a divergence, c'est dans l'interprétation de cette valeur. Il est d'usage courant de rejeter l'hypothèse $\mathcal{H}_0$ (voulant que la population suit la distribution attendue) si la *p-value* est inférieure à 5%. D'Agostino et Stephens soulignent que cette valeur est quelque peu arbitraire et qu'elle est davantage issue d'une sorte de consensus général que d'une évaluation mathématique rigoureuse. L'Écuyer quant à lui va plus loin en notant que, pour des populations très larges (quelques millions), la *p-value* est uniformément répartie sur l'intervalle $(0, 1)$ et l'hypothèse $\mathcal{H}_0$ doit être rejetée si la *p-value* est proche de 0.0 (ou proche de 1.0) à 1% près [30].

2.4.2 Techniques couramment appliquées

Les concepteurs de générateurs matériels de nombres aléatoires s'appliquent à valider leur *design* de différentes façons. Il y a d'une part ceux qui s'astreignent simplement à une validation de la qualité numérique de l'histogramme empirique vs. la courbe mathématique, parmi lesquels [5], [10] et [36]. Des auteurs qui appliquent un test statistique (celui du χ^2 généralement), il y a ceux qui n'observent que peu d'échantillons [13] et ceux qui considèrent un grand nombre d'échantillons, mais ne s'acquittent pas des recommandations de D'Agostino et Stephens qu'ils citent pourtant [33, 34]. Ces derniers réfèrent souvent aux fortes valeurs des *p-value* obtenues, soit-disant garantes de la grande qualité de leur générateur.

Une approche tout à fait originale et rencontrée dans les travaux de D. Thomas [56, 58] consiste à non pas donner la *p-value*, mais plutôt la taille de la population au-delà de laquelle le générateur échoue le test du χ^2. Il est alors considéré qu'un générateur est meilleur qu'un autre si la taille de cette population est très grande.

En ce qui a trait à la corrélation des générateurs, les auteurs se contentent, quand il le font, de vérifier que deux échantillons consécutifs ne sont pas corrélés [3, 33, 34]. Il est étonnant de constater que seul ce test est effectué quand on connaît la grande rigueur avec laquelle les générateurs uniformes sont considérés (voir la section 1.3.2). Nous verrons plus loin que ce critère est souvent insuffisant et qu'une inspection plus poussée est davantage conseillée.

2.5 Conclusion

Ce chapitre a couvert la génération matérielle de nombres aléatoires suivant une distribution non-uniforme. Nous avons pu constater que les architectures dites universelles sont encore à un stade embryonnaire. L'avenue la plus prometteuse est celle de l'inverse utilisant l'approximation polynomiale par parties et proposée par Cheung et al. [10]. Mais cette méthode n'offre malheureusement pas la possibilité d'une reconfiguration en temps d'exécution, ni la garantie d'une généralité d'application. La méthode de combinaison linéaire proposée par D. Thomas [56, 58] est la seule qui puisse prétendre à l'universalité, mais les limitations qui la caractérise quant à la précision demeurent préoccupantes. Finalement, notre revue des générateurs d'exponentielle et de gaussienne nous ont permis non pas tant de connaître les approches utilisées, mais plutôt les critères qui les entourent, tant du point de vue de la résolution des échantillons, du nombre de bits uniformes nécessaires et de la taille relative de leur implémentation. Ces critères nous permettent de mieux situer nos propres travaux et de dessiner la frontière entre ce qui est acceptable et ce qui ne l'est pas.

CHAPITRE 3

MODÈLE MATHÉMATIQUE

3.1 Introduction

Le chapitre qui débute ici couvre la partie théorique de notre travail. Nous présentons l'algorithme spécialement conçu pour la génération de nombres aléatoires sur puce. Dans un premier temps, nous nous intéressons à son expression générale et la rattachons à la théorie des réseaux bayésiens et des arbres de décisions. Nous tentons ensuite de couvrir les considérations propres à l'implémentation matérielle de l'algorithme et en dégageons les bénéfices et les limitations théoriques. Nous verrons par la suite le développement mathématique à même d'ouvrir une brèche dans les écueils mis en évidence et présentons les conclusions à tirer en ce qui a trait aux distributions exponentielle et normale. Le chapitre est achevé par l'énonciation de notre méthodologie de travail et les hypothèses admissibles qui s'y rattachent.

3.2 Principe de base du modèle

Notre travail s'inscrit dans l'héritage scientifique des méthodes stochastiques vues à la section 2.2.1. Rappelons brièvement qu'il s'agit de techniques matérielles associées aux signaux probabilistes. Aussi, notre modèle de calcul se fonde sur la génération des bits d'une variable aléatoire issue d'une distribution non-uniforme. Considérant qu'une variable aléatoire puisse être générée un bit à la fois, il est évident qu'il existe un lien intime qui rattache le modèle de calcul à la représentation binaire des nombres à générer ; aussi la connaissance préalable de cette dernière est primordiale.

La section 3.2.1 présente sommairement notre algorithme afin d'en éclairer les enjeux. La section 3.2.2 fait référence aux réseaux bayésiens auxquels notre modèle peut être rattaché. La section 3.2.3 replace le modèle dans le domaine des arbres de décision binaires afin de mieux en expliciter le fonctionnement et le positionner par rapport au modèle de Knuth et Yao [25]. Finalement, la section 3.2.4 replace le modèle dans un contexte d'exécution matérielle.

3.2.1 Formulation générale

La figure 3.1 illustre graphiquement l'exécution de l'algorithme que nous proposons. Nous considérons ici une distribution arbitraire qui se trouve être la distribution normale de paramètres $\mu = 512$ et $\sigma = 128$ définie sur l'intervalle $\pm 4\sigma$.

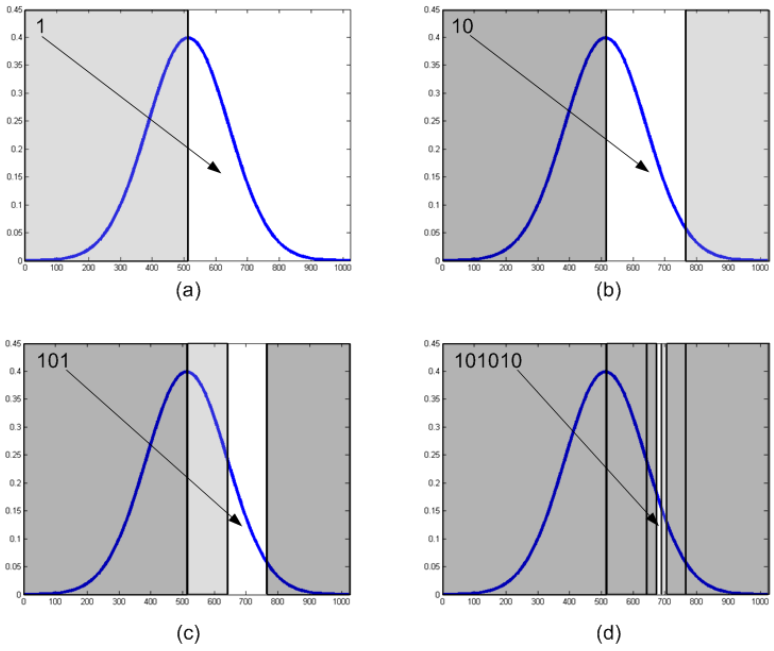


FIG. 3.1 Étapes de l'approche dichotomique. (a) Génération du bit le plus significatif. (b) Génération du second bit. (c) Génération du troisième bit. (d) Étapes subséquentes jusqu'à la génération du sixième bit.

La génération de la variable aléatoire x s'effectue suivant un processus dichotomique permettant de générer x un bit à la fois. À chaque étape de l'algorithme, l'intervalle des valeurs possibles est divisé en deux parties égales, puis l'une des deux parties est choisie de manière stochastique. Cette procédure permet la génération de x un bit à la fois, en commençant par le bit le plus significatif, tel que l'illustre la figure 3.1.a. Le processus se poursuit en divisant en deux parties égales le segment choisi, dont l'une des deux est sélectionnée de manière similaire aux bits précédents, tel que l'illustrent les figures 3.1.b et 3.1.c. Pour des raisons de lisibilité, le processus est continué jusqu'au sixième bit (figure 3.1.d). Il peut néanmoins l'être indéfiniment, suivant la précision recherchée.

L'intérêt de cette méthode est d'abord qu'elle peut-être exécutée en parallèle dans un contexte matériel. En supposant que l'on dispose d'unités chargées de générer un bit de x, on peut imaginer une structure collaborative où les unités se transmettent les mots partiels de x en générant continuellement le bit dont elles sont chargées. Le second intérêt est que plus on progresse, plus la probabilité associée à un bit tend vers 0.5, et la précision requise devient unitaire (un seul bit uniforme suffit). Finalement, cette approche est suffisamment générale pour être considérée universelle, comme nous le démontrons à la section 3.2.2.

3.2.2 Mise en contexte appliquée aux réseaux Bayésiens

Nous allons tenter ici de modéliser l'algorithme au moyen des réseaux bayésiens (RB) dont la paternité revient à Judea Pearl [50]. Pearl a concentré ses efforts durant les années 1980 à asseoir les réseaux bayésiens sur un formalisme on ne peut plus rigoureux qui ne peut que nous être profitable.

Suivant la définition qu'en donnent Russel et Norving [51], un RB est défini comme :

1. Un ensemble de N variables aléatoires formant les nœuds A_i ($1 \leq i \leq N$) du réseau. Ces variables peuvent tout autant être discrètes que continues ;
2. Un ensemble d'arcs dirigés joignant des paires de nœuds du réseau. L'ensemble des nœuds parents du nœud A_i est l'ensemble des nœuds A_j ($j < i$) qui pointent directement vers A_i ;
3. Chaque nœud A_i possède une distribution de probabilité conditionnelle notée $\mathrm{P}(A_i \mid \mathrm{Parents}(A_i))$ qui quantifie sa dépendance à l'état de ses parents.

Si un RB est constitué de nœuds binaires (c'est-à-dire que chaque nœud A_i du réseau ne peut prendre que deux valeurs possibles — 0/1 pour nous —, dont la réalisation est notée a_i), nous l'appellons réseau bayésien binaire (RBB). Soit $\mathcal{S}$ l'état d'un RBB à N nœuds, alors $\mathcal{S}$ peut atteindre au plus 2^N états dont le vecteur $\boldsymbol{a} = a_1 a_2 a \ldots a_N$ est la réalisation. La probabilité jointe[1] $\mathrm{P}(\boldsymbol{a})$ est donnée par la probabilité conjonctive sur l'état des nœuds A_i : $\mathrm{P}(\boldsymbol{a}) = \mathrm{P}(a_1, a_2 a, \ldots, a_N)$. Pearl a fait la démonstration que la distribution jointe est donnée par le produit des probabilités conditionnelles associées aux nœuds [50] :

$$\mathrm{P}(\boldsymbol{a}) = \prod_{i=1}^{N} \mathrm{P}(A_i = a_i \mid \mathrm{Parents}(A_i)) \tag{3.1}$$

Si nous regardons l'état $\boldsymbol{a}$ comme la représentation binaire d'un entier $0 \leq \boldsymbol{a} \leq 2^N$, alors $\boldsymbol{a}$ suit une distribution de probabilité $f(\boldsymbol{a})$ définie par l'équation (3.1). Pour des raisons de lisibilité, nous notons $\hat{\boldsymbol{a}}_i$ la réalisation des parents du nœuds A_i. Nous relevons alors que $\mathrm{P}(A_i = 0 \mid \hat{\boldsymbol{a}}_i) = 1 - \mathrm{P}(A_i = 1 \mid \hat{\boldsymbol{a}}_i)$. Il en ressort que la distribution de probabilité $f(\boldsymbol{a})$ est entièrement décrite par ses paramètres $\mathrm{P}(A_i = 1 \mid \hat{\boldsymbol{a}}_i)$.

[1] Le vocabulaire et les notations sont empruntés à la théorie des RB.

Il est d'usage de décrire les paramètres du RB sous la forme d'un tableau de probabilités conditionnelles [51]. Nous les décrivons plutôt comme des fonctions nodales que nous notons $\varphi_i(\hat{\boldsymbol{a}}_i)$, où $\hat{\boldsymbol{a}}_i$ sera pour nous la représentation binaire d'un entier $0 \leq \hat{\boldsymbol{a}}_i < 2^{i-1}$.

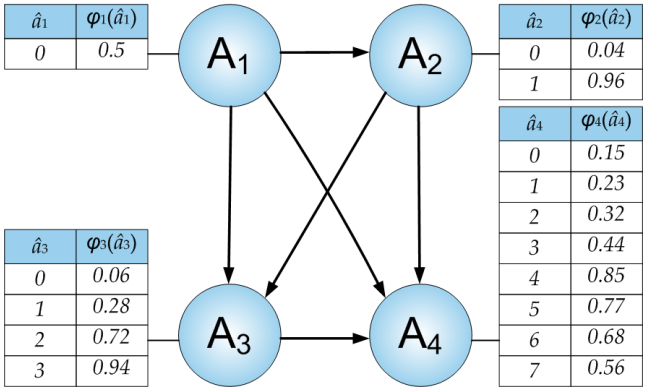


FIG. 3.2 RBB totalement connecté de la distribution vue à la section 3.2.1 et tables des distributions conditionnelles (ou fonctions nodales) associées.

Si l'ensemble des nœuds parents de tout nœud A_i est l'ensemble de tous les nœuds A_j ($j < i$) du RBB, alors ce RBB est dit totalement connecté. Un RBB totalement connecté peut suivre toute distribution discrète. En définissant une fonction bijective $x_b = T(\boldsymbol{a})$ associant à chaque état $\boldsymbol{a}$ une représentation binaire à virgule fixe x_b de la variable aléatoire x, il devient possible de générer n'importe quelle distribution de probabilité $f(x)$ en simulant le RBB totalement connecté. Cette simulation n'est que la traduction dans le domaine des RB de notre algorithme dichotomique et il a de fait été proposé par Henrion [22]. Pour ce faire, il suffit de calculer les paramètres de ce RBB : on définit pour chaque nœud A_i les trois constantes $\boldsymbol{a}_{\mathrm{A},\,i} = 2^{N-i+1}\hat{\boldsymbol{a}}_i$, $\boldsymbol{a}_{\mathrm{B},\,i} = \boldsymbol{a}_{\mathrm{A},\,i} + 2^{N-i}$ et $\boldsymbol{a}_{\mathrm{C},\,i} = \boldsymbol{a}_{\mathrm{B},\,i} + 2^{N-i}$ et on exprime la fonction nodale par :

$$\varphi_i(\hat{\boldsymbol{a}}_i) = \frac{F[T(\boldsymbol{a}_{\mathrm{C},\,i})] - F[T(\boldsymbol{a}_{\mathrm{B},\,i})]}{F[T(\boldsymbol{a}_{\mathrm{C},\,i})] - F[T(\boldsymbol{a}_{\mathrm{A},\,i})]} \tag{3.2}$$

où $F(x)$ est la fonction de répartition de la variabe non-uniforme x.

3.2.3 Mise en contexte appliquée aux arbres de décision binaire

La section 3.2.2 nous a permis d'articuler notre algorithme dans le cadre des RB et d'exprimer ainsi les fonctions nodales comme des distributions de probabilité conditionnelle. Les arbres de décision binaire sont un outil graphique puissant pour illustrer de manière compacte les relations conditionnelles. Outre cette caractéristique illustrative, cet outil va nous permettre de comparer notre algorithme au modèle de génération de variables aléatoires proposé par Knuth et Yao [25] et de mieux apprécier l'originalité de notre approche.

Considérons pour ce faire l'exemple de génération d'une variable aléatoire x dont la représentation binaire tient sur 4 bits[2]. Supposons maintenant que x soit la valeur d'une des six faces d'un dé à jouer. La fonction de répartition de x est simplement exprimée par $\mathrm{P}(X = x) = \mathrm{P}(a_1, a_2, a_3, a_4) = 1/6$ si $x \in \{1, 2, 3, 4, 5, 6\}$ et 0 autrement. Nous obtenons ainsi les fonctions nodales données par la table 3.1, définies pour l'arbre de décision binaire de la figure 3.3.

TAB. 3.1 Valeurs des fonctions nodales pour le problème du dé à six faces. Les valeurs accompagnées d'une astérisque sont arbitrairement fixées à 1.

$\hat{\boldsymbol{a}}_i$	$\varphi_1(\hat{\boldsymbol{a}}_1)$	$\varphi_2(\hat{\boldsymbol{a}}_2)$	$\varphi_3(\hat{\boldsymbol{a}}_3)$	$\varphi_4(\hat{\boldsymbol{a}}_4)$
0	0	$1/2$	$2/3$	1
1	N/A	1^*	$1/3$	$1/2$
2	N/A	N/A	1^*	$1/2$
3	N/A	N/A	1^*	0
4	N/A	N/A	N/A	1^*
5	N/A	N/A	N/A	1^*
6	N/A	N/A	N/A	1^*
7	N/A	N/A	N/A	1^*

[2]Afin de respecter la notation des RB, nous notons $x = a_1a_2a_3a_4$, contrairement à la nomenclature habituelle de la représentation binaire des entiers.

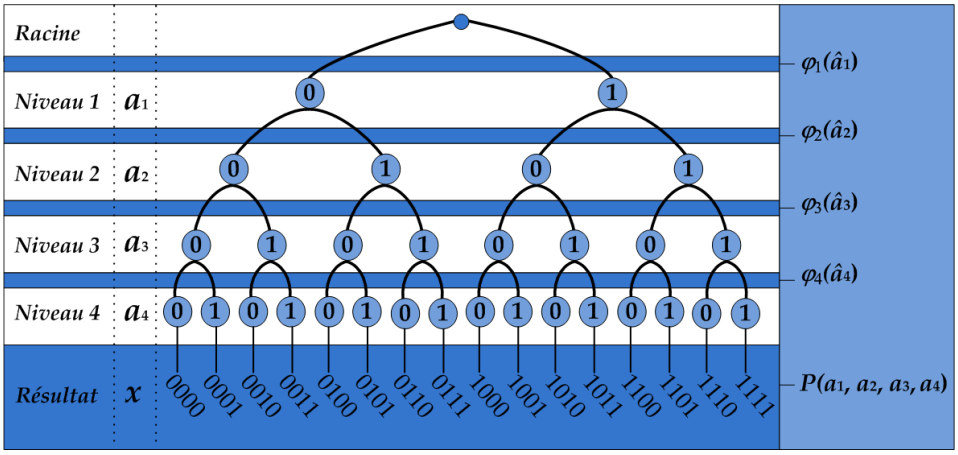


FIG. 3.3 Arbre de décision binaire de quatre niveaux et les fonctions nodales associées pour le problème du dé à six faces.

Considérons maintenant comment le même problème est traité par Knuth et Yao [25]. Le bit de signe a_1 valant toujours 0, ce dernier peut être retiré de l'arbre. Nous aurons donc à générer aléatoirement les bits a_2, a_3 et a_4. La méthode de Knuth consiste à jouer à pile ou face jusqu'à ce que le résultat recherché arrive. Il y a donc une probabilité de 0.5 d'aller à droite (respectivement à gauche) depuis n'importe quel nœud de l'arbre de décision binaire de la figure 3.4. Dans le cas où l'expérience aléatoire atterrit sur $a_2a_3a_4 = 000$ ou $a_2a_3a_4 = 111$, l'algorithme se déplace en amont de l'arbre pour reprendre la procédure. Knuth et Yao ont montré qu'il n'était pas nécessaire de remonter jusqu'à la racine dans ce cas et qu'un retour au niveau 1 était optimal. Il a également fait la démonstration qu'en moyenne, sur un grand nombre d'essais, il faudrait au plus 6 bits uniformes pour obtenir un résultat valide. Généralisant cette conclusion, il parvenait à faire la démonstration que toute génération aléatoire de variable non-uniforme de ce type pouvait être réalisée pour un ratio maximal de $\rho_a = 2$. Nous avons en effet $\rho_a = 6/3 = 2$.

Remarquons pour commencer qu'il est possible de « dérouler » à l'infini l'arbre de décision binaire de la figure 3.4 en s'appuyant sur les chemins de rétroaction. Ce déroulement transforme l'arbre balancé en un arbre binaire totalement débalancé et possédant une in-

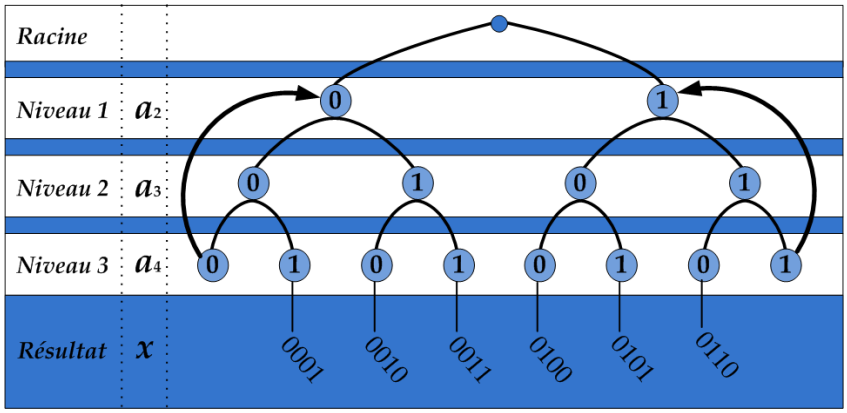


FIG. 3.4 Arbre de décision binaire de trois niveaux dû à Knuth pour le problème du dé à six faces.

finité de niveaux. On dira que le fait de fixer la probabilité de choisir la branche de droite à une probabilité de 0.5 déforme l'arbre pour respecter la distribution de probabilité visée. *A contrario*, notre approche part d'un arbre de décision binaire totalement balancé puis contraint les probabilités conditionnelles sur les nœuds (fonctions nodales φ_i) pour respecter la distribution de probabilité.

Notons que l'implémentation matérielle de l'algorithme de Knuth aurait l'avantage de ne requérir qu'un seul bit uniforme par unité nodale, alors que la nôtre en requiert non seulement plusieurs, mais nécessite également la mémorisation d'un nombre exponentiellement croissant de probabilités conditionnelles. Cependant, l'avantage qu'offre notre approche est que les probabilités conditionnelles au niveau des nœuds sont toujours de loin supérieures à la plus petite des probabilités jointes de la fonction de répartition visée, et ce en raison de la forme de l'équation (3.1). C'est cette expression sous forme de produit des probabilités conditionnelles qui donne l'avantage selon nous à notre méthode en comparaison des approches matérielles adoptées jusqu'ici, notamment la méthode de combinaison linéaire proposée par Thomas [56, 58].

3.2.4 Difficultés dans la mise en œuvre matérielle

Les sections qui précèdent nous ont permis de mieux situer notre approche et d'en éclairer les avantages sur les méthodes matérielles classiques, qui consistent pour la plupart à « câbler » les algorithmes connus. Néanmoins, elle nous a également permis de mesurer les difficultés que pourrait poser son implémentation matérielle. En supposant que nous voulions générer une variable aléatoire non-uniforme x décrite par un nombre raisonnable de bits, 16 par exemple, il apparaît évident qu'un premier problème réside dans l'utilisation des nombres uniformes. En effet, en admettant que 20 bits uniformes soient suffisants pour la génération d'une expérience de Bernoulli (section 2.2.1), il ne faudrait pas moins de 320 bits uniformes pour générer x, soit dix générateurs uniformes tels que le Tausworthe combiné. Ceci est bien entendu inacceptable en regard de ce qui se fait couramment et des critères usuels de disposer d'un générateur compact. De plus, pour générer 16 bits, il faudrait mémoriser pas moins de $2^{16} - 1 = 65\,535$ probabilités conditionnelles, exprimées sur 20 bits chacune.

3.3 Développement du modèle mathématique

Nous allons tenter ici de développer les équations vues à la section 3.2.2 pour voir comment la contrainte de mémoire peut être contournée. Nous nous intéresserons ensuite à l'application de ces équations aux distributions usuelles que nous décrivions à la section 2.2.3 et que résume le tableau 2.1. Finalement, nous nous intéresserons plus attentivement aux distributions qui nous concernent dans ce mémoire, à savoir l'exponentielle et la normale.

3.3.1 Expression compacte de la fonction nodale

La figure 3.5 exprime graphiquement le calcul de la fonction nodale donnée par l'équation (3.2). Le calcul est effectué pour la fonction φ_5 suivant la distribution normale de paramètres $\mu = 512$ et $\sigma = 128$ sur l'intervalle $\pm 4\sigma$. L'axe des $\hat{\boldsymbol{a}}_5$ est superposé aux axes usuels de la courbe pour expliciter le mécanisme de calcul. L'axe des x est également discrétisé de sorte à refléter la subdivision de l'abscisse de la distribution normale $\mathcal{N}(512, 128)$ en 16 zones équidistantes — elles-mêmes subdivisées en deux parties égales. Les 16 zones de l'axe des x correspondent à une valeur de $\hat{\boldsymbol{a}}_5$, ces valeurs vont de 0 à $2^{i-1} - 1 = 2^4 - 1 = 15$. La probabilité conditionnelle φ_5 est donnée par le ratio de la surface de la zone sur l'ensemble de la surface sous la courbe. Joindre les points de ces ratios permet de tracer la fonction nodale, qui dans le cas de la normale est une droite.

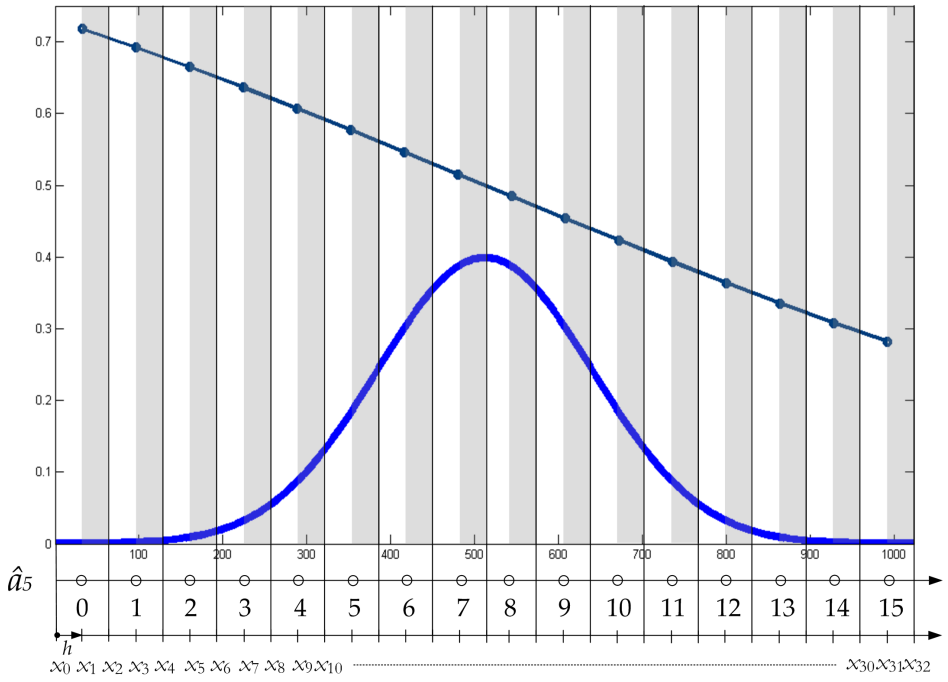


FIG. 3.5 Illustration du calcul de la fonction nodale φ_5 pour $\hat{\boldsymbol{a}}_5$.

La fonction nodale φ_i prendra différentes formes suivant la distribution de probabilité considérée. Dans le cas de la distribution normale, nous aurons toujours une droite pour certaines conditions sur le rang i associé au nœud. Afin de mieux comprendre le comportement de ces fonctions, nous allons les étudier ici. Dans un premier temps, nous allons définir les points x_k ($0 \leq k \leq 2^i$) utiles pour calculer la fonction nodale φ_i, telle que la donne l'équation (3.2), et que nous retrouvons à la figure 3.5. Nous réécrivons alors l'équation (3.2) comme suit :

$$\varphi_i(\hat{\boldsymbol{a}}_i) = 1 - \frac{F(x_{k+1}) - F(x_k)}{F(x_{k+2}) - F(x_k)}, k = 2\hat{\boldsymbol{a}}_i \tag{3.3}$$

Nous notons h le pas entre les x_k (h=$x_{k+1} - x_k$, $0 \leq k < 2^i$). Nous procédons alors à un changement de variables $x_k = kh + x_0$ qui nous permet d'écrire :

$$\varphi_i(\hat{\boldsymbol{a}}_i) = 1 - \frac{F([k+1]h + x_0) - F(kh + x_0)}{F([k+2]h + x_0) - F(kh + x_0)}, k = 2\hat{\boldsymbol{a}}_i \tag{3.4}$$

En supposant h suffisamment petit, il est possible de multiplier le nominateur et le dénominateur par h^{-1} et d'approximer $\varphi_i(\hat{\boldsymbol{a}}_i)$:

$$\varphi_i(\hat{\boldsymbol{a}}_i) \simeq 1 - \frac{f(kh + x_0)}{f([k+1]h + x_0) + f(kh + x_0)}, k = 2\hat{\boldsymbol{a}}_i \tag{3.5}$$

Le développement de Taylor de $f([k+1]h + x_0) \simeq f(kh + x_0) + hf'(kh + x_0)$ de degré 1 nous permet d'écrire :

$$\varphi_i(\hat{\boldsymbol{a}}_i) \simeq 1 - \frac{f(kh + x_0)}{2f(kh + x_0) + hf'(kh + x_0)}, k = 2\hat{\boldsymbol{a}}_i \tag{3.6}$$

Dans le cas de la grande majorité des distributions, $f(kh + x_0) \neq 0$. Nous aurons donc :

$$\varphi_i(\hat{\boldsymbol{a}}_i) \simeq 1 - \frac{1}{2 + h\frac{f'(kh+x_0)}{f(kh+x_0)}}, k = 2\hat{\boldsymbol{a}}_i \tag{3.7}$$

Tout l'intérêt de ce développement tient dans le fait que $\ln'(f(x)) = \frac{f'(x)}{f(x)}$, comme nous le verrons à la section 3.3.2. D'ici là, terminons ce développement mathématique en relevant qu'un développement limité sur l'équation (3.7) nous donne :

$$\varphi_i(\hat{\boldsymbol{a}}_i) \simeq \frac{1}{2} + \frac{h}{4}\ln'(f(x)), x = kh + x_0 \text{ et } k = 2\hat{\boldsymbol{a}}_i \tag{3.8}$$

3.3.2 Applications aux distributions usuelles

TAB. 3.2 Application de l'équation (3.8) aux distributions usuelles

Distribution	Fonction de densité	Fonction de répartition	Approximation de $\varphi_i(\hat{\boldsymbol{a}}_i)$ $x = kh + x_0$ et $k = 2\hat{\boldsymbol{a}}_i$
Normale	$\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{(x-\mu)^2}{2\sigma^2}}$	N/A	$\frac{1}{2} - \frac{h}{4\sigma^2}(x-\mu)$
Exponentielle	$\lambda e^{-\lambda x}$	$1 - e^{-\lambda x}$	$\frac{1}{2} - \frac{h\lambda}{4}$
Log-normale	$\frac{1}{x\sigma\sqrt{2\pi}}e^{-\frac{\ln^2(x-\mu)}{2\sigma^2}}$	N/A	$\frac{1}{2} - \frac{h}{4x} - \frac{h}{4\sigma^2}\frac{\ln(x-\mu)}{x-\mu}$
Rayleigh	$xe^{-\frac{x^2}{2\sigma^2}}$	$1 - e^{-\frac{x^2}{2\sigma^2}}$	$\frac{1}{2} + \frac{h}{4x} - \frac{hx}{4\sigma^2}$
Weibull	$\frac{c}{\lambda}(\frac{x}{\lambda})^{c-1}e^{-(\frac{x}{\lambda})^c}$	$1 - e^{-(\frac{x}{\lambda})^c}$	$\frac{1}{2} + \frac{(c-1)h}{4x} - \frac{ch}{\lambda^c}x^{c-1}$

La table 3.2 présente l'application de l'équation (3.8) aux distributions que nous considérions déjà au chapitre 2. L'intérêt de cette revue est l'étude des expressions analytiques des fonctions nodales. Relevons qu'il est remarquable que la fonction logarithmique

de l'équation (3.8) permette d'exprimer toute multiplication présente dans la fonction de densité $f(x)$ sous la forme d'une addition, et toute puissance sous la forme d'une constante multiplicative. C'est un constat d'autant plus remarquable que la grande majorité des distributions exploitent la fonction exponentielle. Il est tout aussi appréciable de relever que l'équation (3.8) s'appuie sur la fonction de densité $f(x)$ et non la fonction de répartition $F(x)$, dont l'expression analytique n'est pas toujours connue.

Alors que les fonctions nodales des distributions normale et exponentielle exhibent un comportement linéaire, les autres distributions semblent contraindre à une étude plus approfondie que nous nous sommes épargné. Nous allons donc concentrer nos efforts sur les distributions où la méthode semble s'appliquer avec aisance.

3.3.3 Spécificités des distributions exponentielle et normale

Les expressions analytiques des fonctions nodales données au tableau 3.2 sont des approximations qui se fondent sur certaines hypothèses : un pas h suffisamment petit, applicabilité de développements de Taylor de degré 1, etc. L'équation (3.8) offre donc un bon aperçu du comportement attendu de la fonction nodale, mais aucune garantie sur son exactitude numérique. Nous allons considérer dans le détail ce que vaut la fonction nodale pour les distributions exponentielle et normale.

Pour la distribution exponentielle, la fonction nodale devrait se ramener à une constante selon les résultats de la table 3.2. Considérons la génération de x dans l'intervalle $[0, 32\lambda]$ où la distribution est entièrement couverte à $1,2657 \cdot 10^{-12}$ % près. Pour chaque nœud i, nous avons $x_0 = 0$ et $h = 32\lambda/2^i$. Nous réécrivons l'équation (3.4) comme suit :

$$\varphi_i(\hat{\boldsymbol{a}}_i) = 1 - \frac{e^{-\lambda k \frac{32\lambda}{2^i}} - e^{-\lambda(k+1)\frac{32\lambda}{2^i}}}{e^{-\lambda k \frac{32\lambda}{2^i}} - e^{-\lambda(k+2)\frac{32\lambda}{2^i}}}, k = 2\hat{\boldsymbol{a}}_i \tag{3.9}$$

En divisant le nominateur et le dominateur par $e^{-k\frac{32\lambda^2}{2^i}}$, on trouve :

$$\varphi_i(\hat{\boldsymbol{a}}_i) = 1 - \frac{1 - e^{-\frac{32\lambda^2}{2^i}}}{1 - e^{\frac{-64\lambda^2}{2^i}}} \tag{3.10}$$

Ce qui indique que la fonction nodale $\varphi_i(\hat{\boldsymbol{a}}_i)$ ne dépend pas de $\hat{\boldsymbol{a}}_i$ et qu'elle se résume à une constante pour chaque nœud i. Autrement dit, le résultat de la table 3.2 illustre très exactement le comportement de la fonction nodale. Cependant, elle n'en donne pas l'expression analytique exacte mais une approximation qui tend à l'être quand le pas h diminue (quand l'indice du nœud augmente). Cette approximation tend à devenir exact pour les nœuds 5 à 32. Afin d'estimer l'exactitude de cette approximation, la figure 3.6 trace l'erreur relative obtenue pour ces nœuds. L'échelle des ordonnées est logarithmique afin de refléter la précision machine requise.

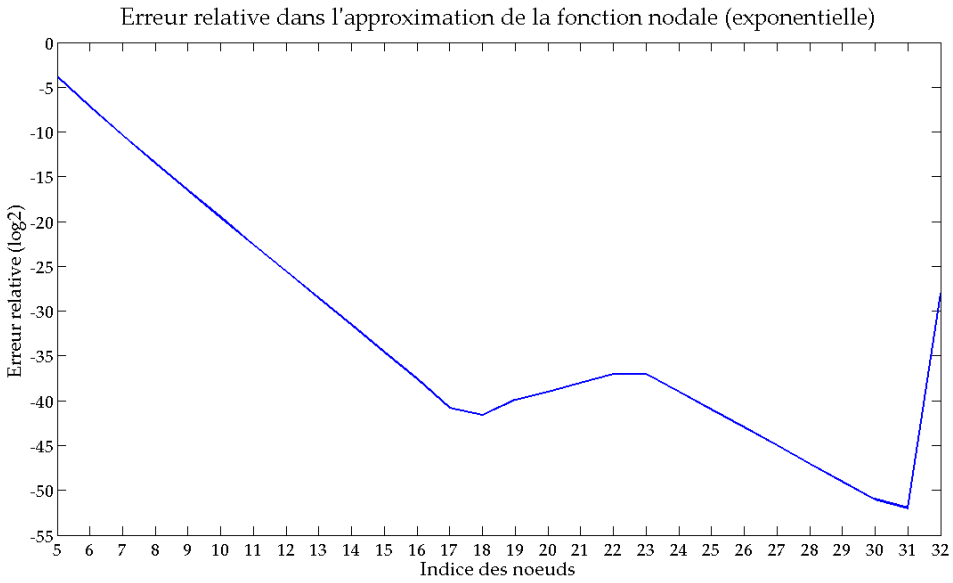


FIG. 3.6 Erreur d'approximation de la fonction nodale pour la distribution exponentielle.

La distribution exponentielle présente un très bon enseignement de la façon dont il faut recourir à l'équation (3.8). Cette dernière doit être utilisée comme estimateur analytique de l'allure de la fonction nodale en vue de l'élaboration d'une stratégie d'approximation. Si la fonction nodale peut être exprimée analytiquement, il est préférable de s'y tenir. Autrement, il est suggéré de rechercher les paramètres permettant de retrouver une expression ayant la forme de celle donnée par l'équation (3.8).

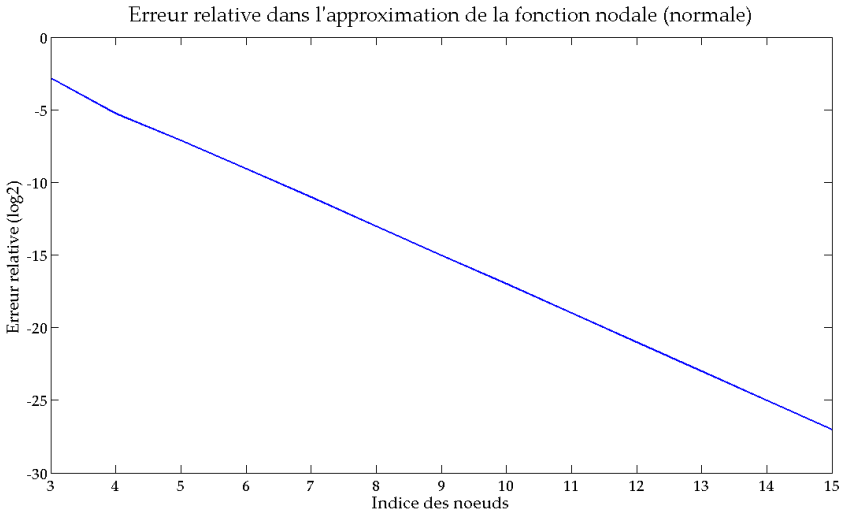


FIG. 3.7 Erreur d'approximation de la fonction nodale pour la distribution normale.

Pour nous en convaincre, considérons la loi normale. Celle-ci ne connaît pas d'expression analytique de sa fonction de répartition mais l'équation (3.8) indique que la fonction nodale devrait avoir un comportement linéaire. Considérons pour fins de vérification la génération de la variable x issue de la normale sur la moitié de l'intervalle $[-4\sigma, +4\sigma]$ (l'autre moitié étant générée par symétrie)[3]. Pour chaque nœud i, nous posons $x_0 = 0$ et $h = 4\sigma/2^i$. Comme on peut s'en apercevoir, l'approximation gagne en précision avec l'ordre des nœuds. Une approximation linéaire faisant fi de l'expression analytique de

[3]Nous nous contentons de 4σ car les outils logiciels tels que MATLAB causent des problèmes d'arrondi dans l'évaluation de la fonction de répartition dans l'intervalle $[4\sigma, 8\sigma]$.

l'approximation est plus appropriée pour les nœuds de premier ordre. Pour les nœuds d'ordre supérieur, l'approximation convient parfaitement comme l'illustre la figure 3.7.

La figure 3.7 présente l'erreur relative (moyenne) de l'approximation donnée par l'équation (3.8). Nous n'avons considéré ici que les nœuds 3 à 15 à cause de la trop grande quantité de mémoire requise pour l'évaluation numérique des fonctions nodales d'ordre supérieur. Il est intéressant de remarquer la décroissance exponentielle de l'erreur. La tendance de cette décroissance est justifiée par le fait que le pas h diminue de moitié à chaque fois que l'ordre du nœud croît d'une unité. Si nous comparons cette courbe avec celle de la figure 3.6, il est curieux de constater la forme particulièrement cahoteuse de l'erreur relative pour les nœuds 17 à 32 dans le cas de l'exponentielle. En vérité, la précision des librairies mathématiques (MATLAB en l'occurrence ici) est particulièrement compromise pour des résolutions aussi extrêmes.

3.4 Hypothèses et méthodologie

Maintenant que notre modèle mathématique est suffisamment élaboré et que notre revue de littérature est suffisamment complète, nous allons énoncer les hypothèses de travail que nous admettrons pour le restant du mémoire et la méthodologie que nous avons suivie pour la réalisation du projet. Nous considérons d'une part la représentation des nombres, c'est-à-dire tout ce qui a trait au format et à la résolution des variables aléatoires. Nous voyons également comment nous qualifions les générateurs aléatoires produits à la lumière des méthodes vues à la section 2.4. Nous énonçons finalement le processus de conception que nous avons mis en application pour la bonne conduite du projet.

3.4.1 Représentation des nombres

Comme nous avons pu le voir au chapitre 2, tous les générateurs matériels de nombres aléatoires suivant une distribution non-uniforme recourent à une représentation à virgule fixe. Lorsque la distribution le requiert, les nombres signés sont représentés en complément à 2. Cette représentation peut causer certains effets indésirables lors d'une conversion vers une représentation à virgule flottante (section 1.3.5), plus répandue dans un environnement logiciel. Nous considérons donc comme une hypothèse raisonnable le recours exclusif à la représentation à virgule fixe. De ce fait, il sera tout aussi raisonnable de considérer toute variable aléatoire continue telle une variable discrète [15].

Le travail de Timarchi et al. [59], revu à la section 2.3.2, nous a permis d'évaluer l'appétit grandissant des architectures recourant à plus de 16 bits. La limitation de la représentation des nombres à 16 bits est également appliquée par Cheung et al. [10], Lee et al. [33, 34, 36] et Thomas [56, 58]. Aussi, nos architectures finales considèrent la variable aléatoire x représentée sur 16 et 32 bits.

Pour ce qui est de la représentation des nombres pour nos générateurs, nos choix sont dictés par la couverture recherchée pour x. Ainsi, la génération de x dans l'intervalle $[0, 16)$ est suffisante pour la distribution exponentielle de paramètre $\lambda = 1$; l'intervalle $[0, 32)$ est considéré exhaustif. Nous aurons donc une représentation non signée à virgule fixe comptant 4 à 5 bits pour la partie entière et les bits restants pour la partie fractionnaire. Dans le cas de la distribution normale centrée réduite, nous considérons la génération de x dans l'intervalle $[-4, +4)$ comme suffisante ; elle sera pour nous exhaustive dans l'intervalle $[-8, +8)$. Nous aurons une représentation signée comportant 1 bit de signe, 3 à 4 bits pour la partie entière et les bits restants pour la partie fractionnaire.

3.4.2 Utilisation des bits uniformes

Nous avons pu voir au chapitre 2 que les auteurs fixaient le nombre de bits uniformes nécessaires à leur générateur en fonction de la précision recherchée. Ainsi, les architectures visant une très grande précision utilisent entre 48 et 80 bits uniformes. Nous avons noté à la section 2.3.4 que cette approche est caractéristique du paradigme de division non-linéaire de l'intervalle $[0, 1]$, qui n'est pas le nôtre. Notre approche consiste plutôt à répéter plusieurs expériences aléatoires de Bernouilli, ce qui nous permet de fixer l'intervalle de génération de x sans connaissance préalable du nombre de bits uniformes nécessaires. Nous avons considéré que 32 bits uniformes étaient largement suffisant pour mener une expérience de Bernoulli avec une bonne précision (en deçà du milliardième). Nous verrons à la section 4.2.2 de quelle façon il est également possible de minimiser le nombre de générateurs uniformes pour l'ensemble du générateur non-uniforme partageant une ressource commune pour l'ensemble des nœuds au moyen d'un chemin de données approprié.

3.4.3 Qualification des générateurs proposés

Nous avons vu à la section 2.4 que le test du χ^2 est le test de choix pour qualifier les générateurs non-uniformes. Contrairement à ce que pratiquent différents auteurs, nous avons choisi de suivre les consignes d'Agostino et Stephens et de diviser l'intervalle de génération en $k = 2N^{\frac{2}{5}}$ intervalles équiprobables pour une population de $N = 20 \cdot 10^6$ échantillons. La *p-value* n'ayant pas grande signification dans ces conditions [30], nous nous contenterons de la mentionner pour confirmer que le générateur passe ou non le test du χ^2. Néanmoins, pour que nos *designs* puissent être comparés à ceux de la littérature, nous procéderons comme le firent Lee et al. [33, 34] en divisant l'intervalle de génération en $k = 100$ intervalles équidistants pour une population de $N = 4 \cdot 10^6$ et comparerons la *p-value* ainsi obtenue à celle donnée dans les articles susmentionnés.

3.4.4 Processus de conception et de validation

Notre travail de conception a suivi un processus de descente hiérarchique d'abstraction. Aussi, chaque implémentation est initialement validée par un équivalent de prototypage logiciel implémenté dans MATLAB, permettant d'en explorer l'efficacité. Une fois le modèle validé à ce niveau d'abstraction par le test du χ^2 pour une population raisonnable ($N = 150\ 000$), Le générateur est implémenté en matériel au moyen d'une description VHDL ou un modèle Simulink équivalent au moyen de l'outil System Generator de Xilinx. Le *design* ainsi produit est alors validé à son tour par le même test du χ^2, pour des populations de $N = 4 \cdot 10^6$ et $N = 20 \cdot 10^6$. Le recours renouvelé au test du χ^2 est justifié par le fait que l'implémentation MATLAB ne fait pas usage du même générateur uniforme que celui implémenté en matériel (section 4.2.4). Toutes les sources utiles sont données en annexes et clairement identifiées dans le texte.

Nous considérons dans notre travail la corrélation sérielle. Le générateur d'exponentielle ayant l'expression mathématique la plus simple (puisque la génération de chaque bit est indépendante de celle des autres), nous y recourrons comme modèle d'investigation empirique. Nous aurons à cœur de produire des générateurs assurant aucune corrélation entre deux échantillons consécutifs. Néanmoins, au contraire de ce qu'ont pu faire les auteurs cités dans ce mémoire, nous nous attarderons également à une étude plus approfondie en considérant davantage de k-tuplets et énoncerons les moyens de garantir la non corrélation en ayant soin de viser des implémentations les plus compactes possibles.

Nous comparons nos générateurs avec ceux cités dans la littérature, tant du point de vue de la vitesse d'exécution, de la surface d'utilisation, de l'intervalle de génération et de la qualité du bruit produit. L'implémentation matérielle étant réalisée sur les FPGA de Xilinx, la surface d'utilisation est comparée en se référant simplement au nombre de *slices* nécessaires et à la fréquence maximale d'opération. Nous basons nos comparaisons sur les estimations apportées par le logiciel ISE 10.1 après placement et routage.

Finalement, nous aurons à cœur de présenter un exemple d'accélération matérielle d'une application Monte Carlo tirée de la pratique des physiciens en optique et qui a le mérite de faire intervenir les générateurs présentés ici. Le code source de cette application est librement disponible sur le net et sera discuté au chapitre 5.

3.5 Conclusion

Ce chapitre nous a permis d'exposer notre algorithme dichotomique ainsi que le modèle mathématique s'y rattachant. Le rapprochement avec le domaine des réseaux bayésiens nous a aidés à mettre en équation la relation des probabilités conditionnelles exprimées par notre approche dichotomique. La représentation sous forme d'arbre de décision binaire a révélé son fonctionnement intuitif et l'originalité qui le distingue du modèle de Knuth et Yao.

Notre développement entourant la fonction nodale nous a permis de considérer les avantages et les limites de notre méthode. Ainsi, nous savons que les distributions exponentielle et normale se prêtent bien à une implémentation matérielle éventuelle puisque leurs fonctions nodales se résument respectivement à une constante et une droite. Le chapitre 4 nous permettra de mieux apprécier ces conditions favorables.

CHAPITRE 4

ARCHITECTURES DES GÉNÉRATEURS ALÉATOIRES

4.1 Introduction

Le chapitre qui débute ici couvre les différentes architectures matérielles que nous proposons et explorons pour générer une variable aléatoire non-uniforme en exploitant l'algorithme dichotomique présenté au chapitre 3. Cette présentation comprend l'architecture *universelle* qui en est issue et sa déclinaison aux distributions exponentielle et normale. L'architecture que nous nommons « universelle » sera pour nous une sorte de patron de conception très général sans véritable visée d'application matérielle. Ce patron nous permettra d'exprimer clairement les difficultés que pose la mise en œuvre matérielle de ces générateurs non-uniformes, que cela soit en terme d'utilisation des bits uniformes, de résolution de la fonction nodale etc.... Nous nous concentrons ensuite à détailler les architectures matérielles possiblement applicables aux distributions exponentielle et normale. Ce recensement permettra de définir les expérimentations auxquelles nous nous livrerons au chapitre 5.

4.2 Architecture universelle

L'algorithme dichotomique du chapitre 3 est basé sur la génération de la variable aléatoire non-uniforme x, un bit à la fois. Nous considérons dans un premier temps les générateurs de Bernoulli existants et y proposons une modification qui nous sera utile. Nous survolons ensuite l'architecture collaborative que nous proposons et les considérations pratiques entourant les générateurs uniformes.

4.2.1 Générateur d'une distribution de Bernoulli

Nous avons vu à la section 2.2.1 l'architecture du générateur de Bernoulli proposée par Gaines [8, 18] (figure 2.1). Celle-ci souffre malheureusement d'un chemin critique préoccupant qui poussa van Daalen et al. [61] d'en proposer une version modifiée (figure 2.2). Si on y regarde de plus près, il devient clair que la partie combinatoire de la cellule de la figure 2.2 est stochastiquement identique à celle de chacun des multiplexeurs de la figure 2.1. Aussi, l'option de pipeliner l'architecture de Gaines est également suggérée par Brown et Card en registrant la sortie des multiplexeurs [8] . Cependant, dans les deux cas, le générateur de Bernouilli ne pourra changer de paramètre (probabilité que la sortie vale 1) d'un coup d'horloge à l'autre. Ce critère est fondamental pour notre générateur puisque chaque nœud produit un bit stochastique dont la probabilité varie avec l'état des nœuds parents.

Analysons le fonctionnement de l'architecture de Gaines. Nous avons volontairement choisi de la représenter comme une combinaison de 2^m multiplexeurs cascadés. Il n'est aucunement nécessaire de disposer d'un nombre de multiplexeurs qui soit une puissance de 2. Supposons cependant que ce soit le cas ; alors il est intéressant de noter qu'il est possible de prévoir la valeur de la sortie Y en parcourant les bits de commande u_i, et ce du bit u_1 au bit u_{2^m} : la sortie Y prendra la valeur du bit b_i dont l'indice est celui du premier bit u_i non nul ; Y vaudra 0 si tous les bits u_i sont nuls. Cette simple analyse nous permet de diviser le générateur de Bernoulli de Gaines en deux parties distinctes. La première partie trouve l'indice du premier bit non nul parmi les 2^m bits uniformes u_i. La seconde partie choisit quant à elle le bit correspondant parmi les bits de probabilité b_i. La sortie est forcée à 0 dans le cas où tous les bits uniformes sont nuls. La première partie est implémentée en utilisant un encodeur de priorité ; la seconde partie est réalisée au moyen d'un multiplexeur ; la sortie est forcée à 0 au moyen d'une porte ET, comme l'illustre la figure 4.1.a.

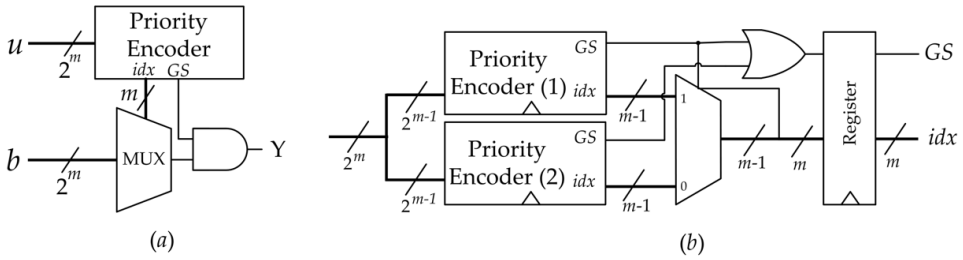


FIG. 4.1 Générateur de Bernoulli proposé. (a) Générateur de Gaines séparé en deux parties. (b) Architecture itérative d'un encodeur de priorité pipeliné.

Les encodeurs de priorité sont moins populaires dans les systèmes numériques que ne le sont les multiplexeurs par exemple. Xilinx préconise d'ailleurs de forcer explicitement l'inférence d'un encodeur de priorité en ajoutant la contrainte PRIORITY_EXTRACT. Si l'encodeur de priorité est trop gros (si m est de l'ordre de 5 et plus), nous proposons d'utiliser une version itérative de l'encodeur de priorité de notre cru que présente la figure 4.1.b. Un tel encodeur de priorité peut être pipeliné au besoin en registrant les sorties.

L'encodeur de priorité 2^m à m itératif fonctionne comme suit. L'entrée est divisée en deux parties confiées à deux encodeurs de priorités 2^{m-1} à $m-1$ notés (1) et (2) (ces derniers peuvent être registrés au besoin) ; le MSB de l'entrée est confié à l'encodeur de priorité (1), le LSB est confié à l'encodeur de priorité (2). La sortie GS (*Group Signal*) d'un encodeur de priorité vaut 1 à moins que l'entrée ne soit nulle. La sortie GS de l'encodeur de priorité itératif est obtenue par une porte OU sur les signaux GS des deux encodeurs de priorité (1) et (2). Si le MSB n'est pas nul, le multiplexeur conduit la sortie de l'encodeur de priorité (1), autrement, il conduit celle de l'encodeur de priorité (2). La sortie de l'encodeur de priorité itératif (représentée sur m bits) est obtenue en concaténant la sortie du multiplexeur et le signal GS de l'encodeur de priorité (1) (ce dernier est placé en MSB).

4.2.2 Architecture collaborative des noeuds

Quand on cherche à concevoir une machine mimiquant le comportement de l'algorithme dichotomique, on en vient à se demander : 1) Comment faire pour que la génération soit la plus efficace ? 2) Comment faire pour que les nœuds communiquent entrent eux ?

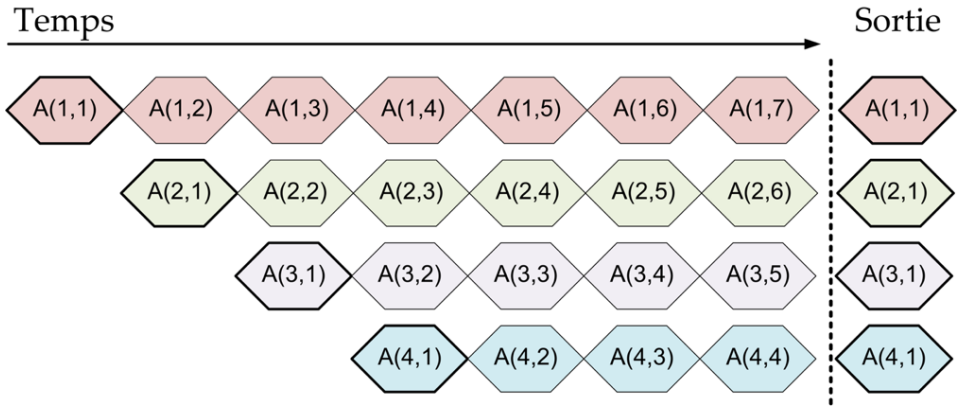


FIG. 4.2 Génération pipelinée des états des nœuds pour un RBB à quatre nœuds. La sortie est validée une fois que le dernier état est généré.

À la première question, il est possible de répondre aisément en imaginant un système pipeliné dont le fonctionnement est schématisé à la figure 4.2. Chaque nœud doit connaître l'état de ses parents. Les parents sont donc générés un à la fois en fonction des nœuds précédents. Le premier nœud est indépendant des suivants et ses états $A(1, j)$ peuvent être générés continuellement. Le second nœud génère son état en fonction du nœud précédent avec une période de retard. La relation de dépendance est indiquée à la figure 4.2 par le second indice j. Le processus est ainsi continué pour l'ensemble des nœuds jusqu'à ce que le dernier nœud soit atteint. À ce moment, la sortie est mise à jour. Il devient possible dans ces conditions de générer un nouvel état du réseau à chaque période à la condition que le pipe soit rempli.

Le schéma de la figure 4.3 répond à la seconde des deux questions. Il s'agit là de notre structure collaborative des nœuds. Chaque unité nodale possède une sortie registrée. À chaque coup d'horloge, elle concatène son entrée (état des parents) au bit qu'elle a elle-même généré. Ce bit est fonction de l'entrée et se place en LSB. À la fin de la structure, l'état $\boldsymbol{a}$ du RBB est ainsi généré à chaque coup d'horloge.

Pour que chaque cellule puisse générer un bit stochastique, une seconde entrée doit être acheminée : le nombre aléatoire uniforme nécéssaire à la génération du bit aléatoire. Plutôt que d'acheminer un nombre aléatoire dans son intégralité, nous proposons de profiter de la structure de la figure 4.1.a pour n'acheminer que *l'indice* associé au nombre uniforme (la sortie d'un encodeur de priorité), de sorte à faciliter le routage. Il est également possible de se dispenser du routage du signal GS, ce dernier n'ayant d'utilité que très exceptionnellement (la probabilité de son utilisation est de 2^{-2^m}).

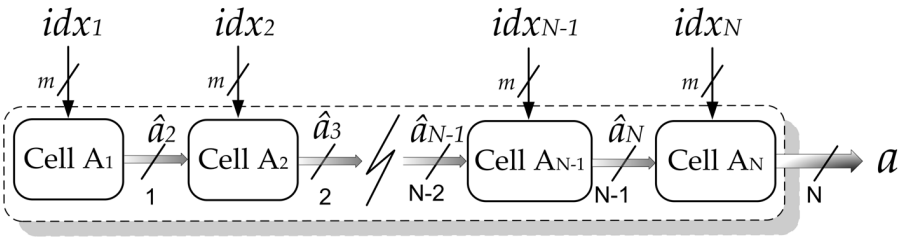


FIG. 4.3 Structure collaborative des nœuds. Les sorties des cellules sont registrées.

Songeons à fournir à la structure un seul générateur aléatoire sans compromettre pour autant la qualité du bruit généré. Cette idée part de l'hypothèse qu'un chemin de données adéquat inhiberait les contrefaits éventuels de ce choix. La figure 4.4 suggère trois scénarios : le premier scénario (figure 4.4.a) achemine le nombre aléatoire[1] en supposant que le registrement des unités nodales permet la génération de chaque bit de $\boldsymbol{a}$ indépendemment des autres ; le second scénario (figure 4.4.b) part du même principe mais ajoute

[1]Nous dirons nombre aléatoire plutôt que l'indice obtenu par l'encodeur de priorité pour alléger l'écriture.

des registres entre chaque nœud ; le troisième scénario (figure 4.4.b) procède de la même façon (générateur unique plus registrement entre les nœuds) mais achemine les données à contre-courant du chemin de génération, comme le faisait van Daalen [61] pour son générateur de Bernoulli (section 2.2.1).

Afin de bien comprendre le fonctionnement de ces architectures, considérons l'exemple de la génération de quatre bits suivant les trois scénarios de la figure 4.4. Pour ce faire, la figure 4.5 reprend le chronogramme de la figure 4.2 en y superposant temporellement la génération des nombres aléatoires uniformes. La table 4.1 part de ce graphique pour identifier les nombres aléatoires impliqués dans la génération des bits des sept échantillons $A(1, j)$ à $A(7, j)$.

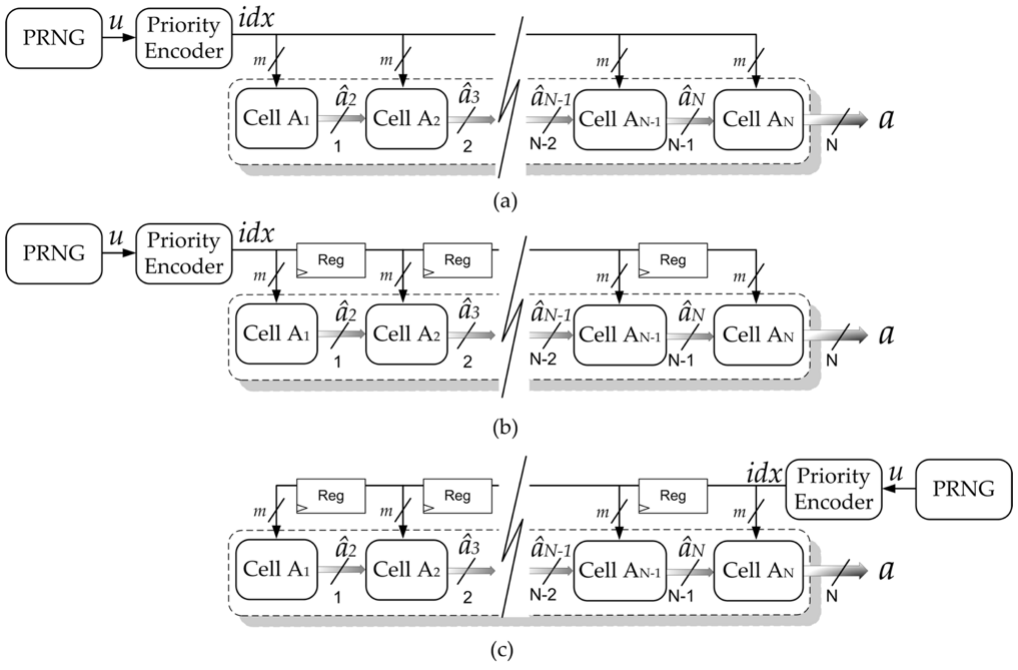


FIG. 4.4 Scénarios d'alimentation en nombre aléatoires. (a) Un générateur unique pour tous les nœuds. (b) Un générateur unique avec registrement entre les nœuds. (c) Générateur unique, registrement entre les nœuds et propagation à contre-courant.

Au scénario #1 (figure 4.4.a), les bits de tous les échantillons considérés sont générés indépendamment les uns des autres. Le scénario #2 (figure 4.4.b) ne garantit pas cette condition et doit par conséquent être rejeté. Le scénario #3 (figure 4.4.c) exhibe un comportement bien particulier : le chemin de propagation des nombres aléatoires allant à contre-courant de celui de la génération des bits stochastiques, les premiers échantillons générés sont invalides ; dès après, les échantillons sont constamment générés et les bits sont générés indépendamment les uns des autres.

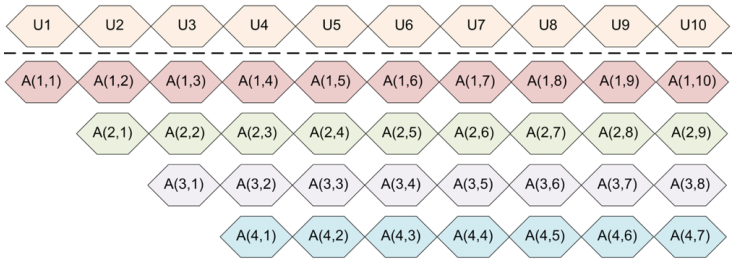


FIG. 4.5 Superposition temporelle des données et des uniformes.

TAB. 4.1 Trois scénarios touchant les bits uniformes.

Échantillon	Scénario #1	Scénario #2	Scénario #3
$A(1,j)$	$U(1,2,3,4)$	$U(1,1,1,1)$	Invalide
$A(2,j)$	$U(2,3,4,5)$	$U(2,2,2,2)$	Invalide
$A(3,j)$	$U(3,4,5,6)$	$U(3,3,3,3)$	Invalide
$A(4,j)$	$U(4,5,6,7)$	$U(4,4,4,4)$	$U(1,3,5,7)$
$A(5,j)$	$U(5,6,7,8)$	$U(5,5,5,5)$	$U(2,4,6,8)$
$A(6,j)$	$U(6,7,8,9)$	$U(6,6,6,6)$	$U(3,5,7,9)$
$A(7,j)$	$U(7,8,9,10)$	$U(7,7,7,7)$	$U(4,6,8,10)$

Il n'est pas évident à cette étape de mesurer l'impact du compromis que nous avons accepté. Ainsi, les échantillons $A(1,j)$ et $A(2,j)$ du scénario #1 partagent un certain nombre d'échantillons uniformes. Ce n'est pas le cas du scénario #3 (voir les échantillons $A(4,j)$ et $A(5,j)$), mais ce même patron est rencontré entre deux échantillons générés à deux temps d'intervalle au scénario #3 ($A(4,j)$ et $A(6,j)$; $A(5,j)$ et $A(7,j)$).

4.2.3 Architecture des unités nodales

Maintenant que nous avons une idée de la façon dont la structure collaborative des nœuds peut être réalisée, considérons l'architecture des unités nodales. Chacune de ces cellules reçoit les bits précédemment générés par les nœuds hiérarchiquement supérieurs et un nombre aléatoire uniforme (son indice représenté sur m bits). L'unité nodale comprend donc le multiplexeur issu de la seconde partie du générateur de Bernoulli modifié que nous avons proposé. Ce multiplexeur est commandé par les m bits de l'encodeur et reçoit 2^m bits représentant la fonction nodale φ_i. L'unité nodale possède une sortie registrée qui est simplement la concaténation de l'entrée et du bit généré. Le bit généré va en LSB.

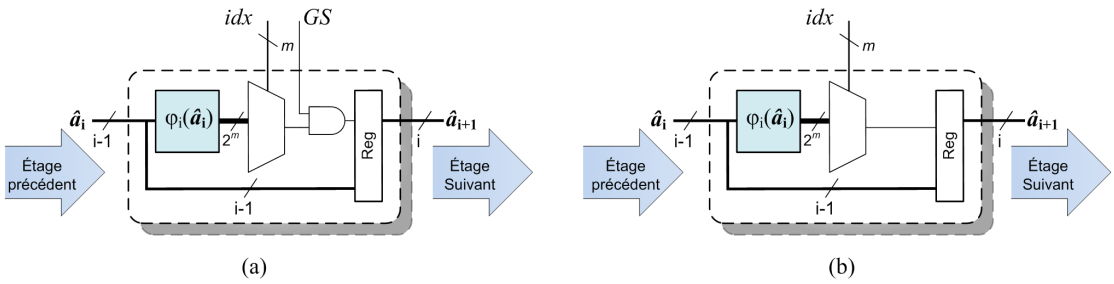


FIG. 4.6 Architecture générique d'une unité nodale.

La figure 4.6 présente les deux façons possibles d'implémenter cette unité nodale : la figure 4.6.a traduit exactement la séparation en deux du générateur de Bernoulli modifié ; la figure 4.6.b présente le cas où le signal GS est ignoré, ce qui épargne le routage d'un signal le long de la structure collaborative et évite l'utilisation d'une porte ET.

Finalement, considérons l'implémentation de la fonction nodale φ_i. Celle-ci peut soit être réalisée au moyen d'une mémoire de taille $2^{i-1} \times 2^m$, d'une fonction arithmétique correspondant à l'équation 3.8 ou encore d'une fonction stochastique équivalente. Nous verrons dans ce qui suit quelle implémentation choisir.

4.2.4 Choix d'un générateur uniforme

Nous avons vu à la section 1.3.4 les différentes stratégies permettant d'implémenter des GRLB. Nous y mentionnions que Xilinx mettait à la disposition des concepteurs de circuits numériques des exemples de code et des tables de polynômes irréductibles pour les LFSR. Malheureusement, nous mentionnions également qu'il a été démontré que les LFSR possédaient de faibles propriétés aléatoires en raison de la très forte corrélation entre les nombres générés. Plusieurs solutions s'offrent au concepteur pour utiliser les LFSR dans un contexte de simulation stochastique. La première de ces solutions est de réduire le débit du LFSR en prenant un échantillon uniforme tous les N coups d'horloge, où N est le nombre de registres du LFSR. Cette approche est à rejeter en raison des besoins de rapidité d'exécution de la structure collaborative sus-décrite. Une autre solution consiste à utiliser M LFSR en parallèle. Il a été prouvé que cette solution exhibait de très bonnes propriétés aléatoires [43], mais cette assertion n'est vraie que pour certaines conditions sur les graines (critère de non-corrélation des graines), ce qui en fait une contrainte d'autant plus disgracieuse que les générateurs uniformes ainsi obtenus sont coûteux en matériel (environ 1000 *slices*). Ainsi, la piste traditionnelle des LFSR est à écarter dans la conception d'un générateur aléatoire de qualité.

Cheung et al. [10] ont rapporté une implémention de générateur de type Taussworthe combiné [27] n'utilisait que 141 *slices* tout en affichant une période de 2^{88}. Le générateur Taussworth combiné a le désavantage d'imposer la représentation 32 bits du nombre aléatoire uniforme[2]. Nous accepterons avec indulgence certaines sous-optimisations des conceptions à venir dues à cette contrainte, comme le firent avant nous les auteurs susnommés [10].

[2]Il est en vérité possible de reprendre les équations de L'Écuyer et de retrouver des configurations permettant une représentation sur n bits quelconques, mais rares sont ceux qui se sont lancés dans l'aventure, et nous n'avons pas échappé à la règle.

4.3 Application à la distribution exponentielle

La distribution exponentielle a des propriétés qui la distingue singulièrement des autres distributions. Notre développement de l'équation (3.10) montrait que la fonction nodale associée à la distribution exponentielle se réduisait à une constante. Ce résultat signifie, à toute fin utile, que les nœuds du réseau sont indépendants les uns des autres. Autrement dit, il n'est pas nécessaire de générer les nœuds parents préalablement à la génération des nœuds hiérarchiquement inférieurs. La conclusion à tirer de cela est que dans le cas de la distribution exponentielle, la structure collaborative des nœuds n'est pas nécessaire. Dans ce qui suit, nous allons considérer deux façons permettant de tirer parti de cette indépendance pour optimiser l'architecture. Nous pourrons ainsi établir un protocole pour l'étude du comportement des différents scénarios d'alimentation en nombres aléatoires de la figure 4.4.

4.3.1 Vue globale de l'architecture

La figure 4.7.a reprend l'architecture collaborative de la figure 4.3 en brisant les liens de dépendance (rendus inutiles) entre les nœuds. La figure 4.7.b inscrit cette version modifiée dans le cadre du scénario #1 de la figure 4.4a.

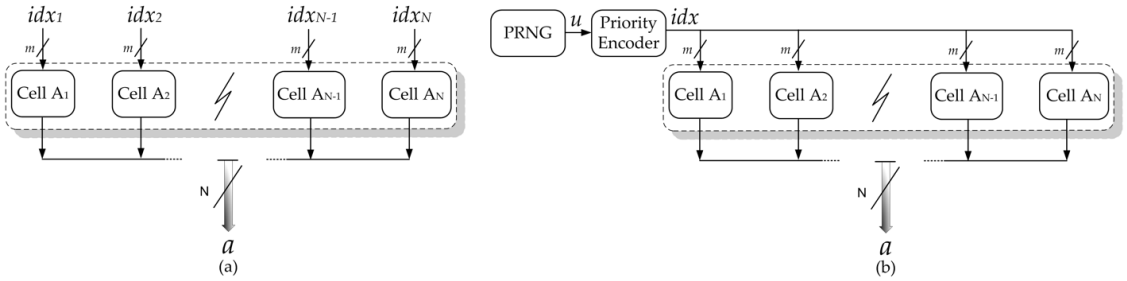


FIG. 4.7 Esquisse de l'architecture d'un générateur d'exponentielle.

En se référant à l'architecture de l'unité nodale présentée à la figure 4.6, on s'aperçoit que l'exponentielle est implémentable à l'aide d'une simple mémoire de $N \times 2^m$, où 2^m renvoie à la longueur du nombre aléatoire uniforme et N au nombre de nœuds dans le RBB. Néanmoins, la rupture des liens de dépendance modifie quelque peu le comportement de la structure collaborative. Ainsi, au lieu de se comporter comme le prévoyait le scénario #1 décrit à la table 4.1, la structure exhibe le comportement propre au scénario #2. Nous allons considérer à la section 4.3.2 qui suit comment il est possible de modifier la structure pour qu'elle affiche différents comportements des scénarios précédemment présentés.

4.3.2 Étage de sortie

En s'en tenant à l'esquisse de la figure 4.7.b, il apparaît que la sortie $\boldsymbol{a}$ ne peut prendre qu'une des 2^m valeurs présentes dans la ROM du générateur d'exponentielle. Or le générateur d'exponentielle est supposé pouvoir générer jusqu'à 2^N valeurs différentes ($2^N \gg 2^m$) de $\boldsymbol{a}$ suivant la distribution visée. On peut alors corriger ce comportement en ajoutant un étage de sortie constitués de chemins de délais pour chaque bit, comme l'illustre la figure 4.8, où l'évaluation préliminaire de $\boldsymbol{a}$ est notée $\hat{\boldsymbol{a}}$.

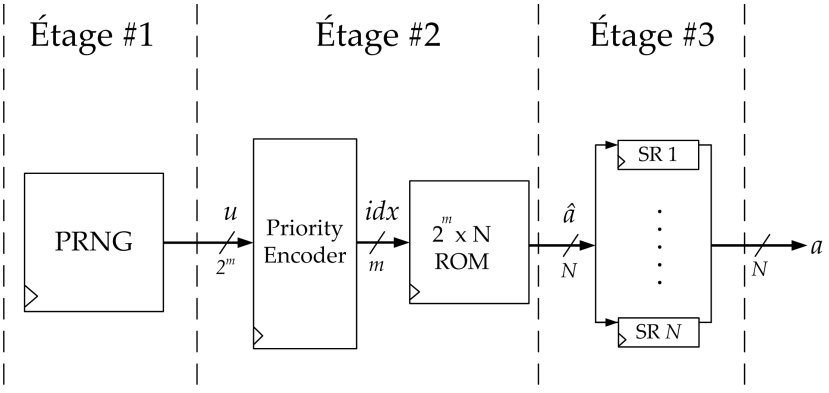


FIG. 4.8 Architecture d'un générateur d'exponentielle modifiée.

Si tous les registres à décalage sont de longueur différente et que cette différence est exactement de $\mathrm{diff} = 1$, alors le générateur d'exponentielle se comporte suivant le scénario #1. Si cette différence est exactement de 2, le générateur se comporte suivant le scénario #3. En variant la différence de la longueur des registres à décalage, il est possible d'aboutir à d'autres scénarios. Ainsi, outre le fait d'avoir entre nos mains une nouvelle architecture de générateur d'exponentielle, nous voilà en possession d'un outil matériel permettant l'exploration de la corrélation sérielle de l'architecture collaborative alimentée par un seul PRNG que nous évoquions à la section 4.2.2.

4.3.3 Architecture alternative

Repartons maintenant du principe que les bits d'un générateur d'exponentielle sont indépendants les uns des autres suivant le résultat de l'équation (3.10). À la section 4.2.1, nous rejetions le générateur de Bernoulli de van Daalen et al. [61] sous prétexte qu'il ne permettait pas le changement rapide du paramètre de la distribution de Bernoulli.

Cet argument n'a plus lieu d'être dans le cas de la distribution exponentielle puisque ce paramètre (fonction nodale) ne varie plus. Aussi devient-il possible d'implémenter un générateur d'exponentielle en utilisant en parallèle N générateurs de Bernoulli de van Daalen. Cette approche permet d'éviter les registres à décalage de l'étage de sortie présenté à la figure 4.8. Ainsi aurons-nous au moins trois différentes architectures d'exponentielle à évaluer, nommément :

1. Générateur d'exponentielle à N Bernoulli de van Daalen ;
2. Générateur d'exponentielle à ROM compacte avec étage de sortie à $\mathrm{diff} = 1$;
3. Générateur d'exponentielle à ROM compacte avec étage de sortie à $\mathrm{diff} = 2$;
4. Générateur d'exponentielle à ROM compacte avec toute autre configuration de l'étage de sortie.

4.4 Application à la distribution normale

Au contraire de ce que nous avons pu constater avec la distribution exponentielle, la fonction nodale de la distribution normale n'exhibe pas toujours un comportement linéaire. Aussi, l'implémentation matérielle de l'unité nodale du générateur de la distribution normale aura différentes formes selon la hiérarchie à laquelle elle appartient. La section qui suit permet de donner une vue d'ensemble de ces architectures avant que leur mise en œuvre matérielle ne soit présentée au chapitre 5.

4.4.1 Implémentation de la fonction nodale

Nous avons vu à la section 4.2.3 que le bloc associé à la fonction nodale pouvait être implémenté de trois façons possibles. La première et la plus simple de ces approches consiste à utiliser une mémoire de type RAM ou ROM dans laquelle les valeurs de la fonction nodale $\varphi_i(\hat{\boldsymbol{a}}_i)$ sont enregistrées. Cette solution est viable en autant que la taille de la mémoire requise soit raisonnable, ce qui n'est vrai que pour les nœuds de premier ordre. Pour les nœuds d'ordre supérieur, nous devons recourir à des solutions alternatives.

La première de ces approches alternatives se fonde sur l'expression analytique de la fonction nodale donnée par l'équation (3.8). Cela peut être fait en recourant aux circuits arithmétiques usuels tels que les additionneurs, les multiplieurs, etc ; mais il est également possible d'avoir recours à des blocs spécialisés tels que les architectures CORDIC. Néanmoins, vu le nombre de nœuds impliqués, cette solution n'est pas réaliste eu égard à nos critères de tailles sur puce. La seconde méthode se base sur l'exploitation de l'arithmétique stochastique pour réaliser la même tâche. L'arithmétique stochastique dispose à l'heure actuelle d'un nombre restreint d'opérateurs [8] ; mais ces derniers suffisent dans le cas de la distribution normale.

4.4.2 Implémentation arithmétique des noeuds linéaires

Reprenons ici l'expression analytique de la fonction nodale que nous donnait l'équation (3.8) pour la loi gaussienne de paramètres μ et σ :

$$\varphi_i(\hat{\boldsymbol{a}}_i) = \frac{1}{2} - \frac{h}{4\sigma^2}(x-\mu), x = kh + x_0, k = 2\hat{\boldsymbol{a}}_i \tag{4.1}$$

Cette expression n'est pas exacte pour les nœuds de premier ordre. La figure 3.7 nous donnait le comportement de son erreur relative, laissant présumer son applicabilité aux nœuds d'indice 5 et plus et pour une couverture symétrique de $\pm 4\sigma$. La fonction nodale peut néanmoins exhiber un comportement linéaire sans que ce ne soit exactement celui de l'équation (4.1). Il est alors possible d'approximer la fonction nodale en effectuant une régression linéaire sur ses points. En notant α_i et β_i les paramètres de la fonction nodale $\varphi_i(\hat{\boldsymbol{a}}_i) = \alpha_i \hat{\boldsymbol{a}}_i + \beta_i$, on retrouve à la table 4.2 l'ensemble des paramètres obtenus par régression linéaire pour une couverture de $\pm 8\sigma$.

TAB. 4.2 Paramètres des nœuds linéaires de la gaussienne.

i	β_i	α_i	i	β_i	α_i
4	0.37409	-0.057993	10	0.49998	-0.000030
5	0.46425	-0.024042	11	0.5	$-7.6292\cdot10^{-6}$
6	0.49397	-0.007401	12	0.5	$-1.9073\cdot10^{-6}$
7	0.49896	-0.001943	13	0.5	$-4.7684\cdot10^{-7}$
8	0.49975	-0.000487	14	0.5	$-1.1921\cdot10^{-7}$
9	0.49994	-0.000122	15	0.5	$-2.9802\cdot10^{-8}$

Plutôt que de considérer la probabilité conditionnelle que le nœud génère un 1, il est plus avantageux de considérer la probabilité qu'il génère un 0 (le 0 est alors obtenu en inversant la sortie) puisque $P(A_i = 0 \mid \hat{\boldsymbol{a}}_i) = 1 - \varphi_i(\hat{\boldsymbol{a}}_i)$. Or $1 - \varphi_i(\hat{\boldsymbol{a}}_i) = -\alpha_i \hat{\boldsymbol{a}}_i + (1 - \beta_i)$, ce qui permet d'aboutir à une équation ne faisant intervenir que des opérandes positifs.

Pour des raisons purement technologiques concernant la multiplication et tel que discuté à la section 4.4.1, l'implémentation arithmétique des nœuds n'est pas réalisée matériellement. Cependant, elle est considérée pour une évaluation logicielle pour valider les paramètres de la table 4.2.

4.4.3 Implémentation stochastique des noeuds linéaires

L'implémentation stochastique des nœuds linéaires sera pour nous un sujet d'investigation ayant pour finalité l'optimisation de la consommation des ressources logiques. Rappelons que l'arithmétique stochastique constitue un paradigme où les nombres sont représentés par des pulses aléatoires dont la probabilité de valoir 1 exprime une valeur réelle située entre 0.0 et 1.0. En considérant deux signaux stochastiques indépendants de probabilité p_1 et p_2, opérer un ET logique sur les deux signaux permet d'obtenir un troisième signal stochastique de probabilité $p_c = p1 \cdot p2$. Ainsi, une multiplication est réalisée stochastiquement au moyen d'une simple porte ET. L'addition dans le domaine de l'arithmétique stochastique est cependant moins simple à exprimer puisque le résultat peut dépasser l'intervalle des valeurs permises si il est supérieur à 1.0. Il est néanmoins possible de disposer d'une addition pondérée en recourant à un multiplexeur dont les signaux de commande sont aléatoires [8, 18]. Prenons pour exemple le cas d'un multiplexeur 2 à 1 recevant deux signaux stochastiques indépendants de probabilité p_1 et p_2 et dont le signal de commande a une probabilité de 0.5 de valoir 1. Alors le signal de sortie du multiplexeur sera un signal stochastique dont la probabilité de valoir 1 est $p_a = 0.5p_1 + 0.5p_2$. En fixant la probabilité sur le signal de commande à w, nous aurons $p_a = (1 - w)p_1 + wp_2$. Aussi, si nous voulons effectuer une addition sur les nombres x_1 et x_2, il suffit d'effectuer une addition pondérée sur deux signaux stochastiques dont la probabilité est respectivement $x_1/(1 - w)$ et x_2/w en utilisant un multiplexeur 2 à 1 commandé par un signal stochastique de probabilité w, en autant que ces valeurs ne dépassent pas la valeur limite 1.0.

Si nous nous référons aux paramètres β_i de la table 4.2, on s'aperçoit que :

$$0.5 \leq 1 - \beta_i < 0.75 \tag{4.2}$$

Ainsi, l'opération arithmétique à effectuer pour obtenir le complément de la fonction nodale $1 - \varphi_i(\hat{\boldsymbol{a}}_i) = -\alpha_i \hat{\boldsymbol{a}}_i + (1 - \beta_i)$ est réalisable dans le cadre de l'arithmétique stochastique en considérant un multiplexeur 2 à 1, deux signaux stochastiques de probabilité p_1 et p_2 et un signal de commande stochastique de probabilité w tels que :

$$\begin{cases} p_1 = \hat{\boldsymbol{a}}_i \cdot (\frac{-\alpha_i}{w}) \\ p_2 = \frac{1-\beta_i}{1-w} \\ w \leq 0.25 \end{cases} \tag{4.3}$$

Pour des considérations pratiques, on fixe w à 0.25 puisqu'un signal généré avec une telle probabilité s'obtient facilement en réalisant un ET logique sur deux bits aléatoires de probabilité 0.5.

Pour chaque nœud i, la valeur p_2 est constante alors que celle de p_1 varie dans le temps avec la variation de la valeur $\hat{\boldsymbol{a}}_i$. Aussi, une question demeure en suspens : Comment générer le signal p_1 sachant que $\hat{\boldsymbol{a}}_i$ est un entier ($0 \leq \hat{\boldsymbol{a}}_i < 2^{i-1}$) ? La réponse est fort simple : p_1 peut être vu comme un signal aléatoire obtenu par la multiplication stochastiques de deux signaux aléatoires de probabilité $p_{1.1} = \hat{\boldsymbol{a}}_i \cdot 2^{1-i}$ et $p_{1.2} = -2^{i-1} \cdot \alpha_i / w$ respectivement. La table 4.3 confirme la faisabilité de la chose pour les nœuds A_6 à A_{15} (pour les nœuds 4 et 5, la valeur $p_{1.2}$ est supérieure à 1.0).

TAB. 4.3 Paramètres des nœuds stochastiques de la gaussienne.

i	$p_2 = (1-\beta_i)/(1-w)$	$p_{1.2} = -2^{i-1}\alpha_i/w$	i	p_2	$p_{1.2}$
4	0.76099	1.8558	10	0.66668	0.062500
5	0.69095	1.5387	11	0.66667	0.031250
6	0.67244	0.9473	12	0.66666	0.015625
7	0.66804	0.4975	13	0.66666	0.007812
8	0.66700	0.2497	14	0.66666	0.003906
9	0.66674	0.1249	15	0.66666	0.001953

La figure 4.9 résume sous forme schématisée la construction d'une telle cellule nodale :

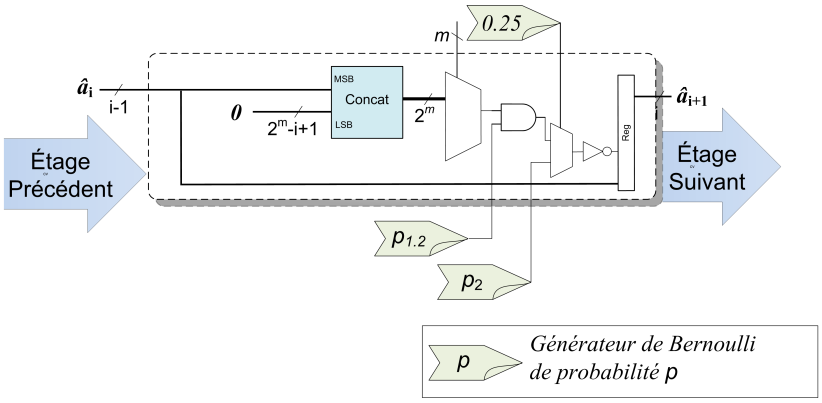


FIG. 4.9 Architecture d'un nœud stochastique pour le générateur de la normale avec en légende le symbole employé pour la représentation d'un générateur de Bernoulli.

4.4.4 Implémentation alternative des noeuds linéaires

Considérons une autre implémentation des nœuds linéaires du générateur de gaussienne.

Le complément de la fonction nodale exprimé par l'équation (4.1) est donné par :

$$1 - \varphi_i(\hat{\boldsymbol{a}}_i) = \frac{1}{2} + \frac{h}{4\sigma^2}(x-\mu), x = kh + x_0, k = 2\hat{\boldsymbol{a}}_i \tag{4.4}$$

En remplaçant les paramètres de la gaussienne par ceux de la normale centrée réduite, on obtient les paramètres pour une génération symétrique ($x_0 = 0$) :

$$1 - \varphi_i(\hat{\boldsymbol{a}}_i) = \frac{1}{2} + \frac{h^2}{2}\hat{\boldsymbol{a}}_i \tag{4.5}$$

Dans le cas d'une génération symétrique de la normale centrée réduite sur l'intervalle $\pm 8\sigma$, nous pourrons exprimer h comme une puissance de 2 : $h = 2^{3-i}$, sachant que $0 \leq \hat{\boldsymbol{a}}_i < 2^{i-1}$, Ainsi, pour $i \geq 5$, la fonction nodale s'exprime en virgule fixe par la simple concaténation de $0.1_{(2)}$ et des bits de $\hat{\boldsymbol{a}}_i$ suivant un décalage approprié. Plus exactement, elle est donnée par :

$$1 - \varphi_i(\hat{\boldsymbol{a}}_i) = 2^{-1} + 2^{5-2i}\hat{\boldsymbol{a}}_i, i \geq 5 \tag{4.6}$$

La figure 4.10 résume schématiquement cette deuxième possible implémentation de la cellule nodale du générateur de la normale. Au chapitre 5, nous étudierons les performances des deux implémentations basées respectivement sur 1) l'architecture d'un nœud stochastique et 2) l'architecture d'un nœud à concaténation de l'entrée.

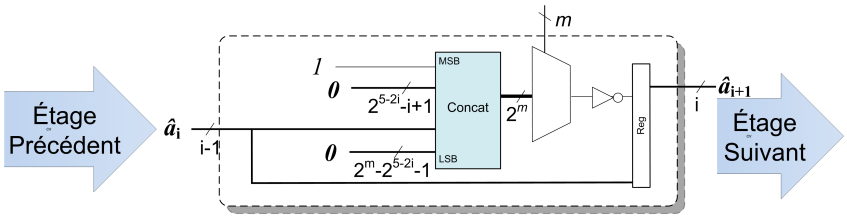


FIG. 4.10 Architecture alternative d'un nœud pour le générateur de la normale.

4.5 Conclusion

Ce chapitre a couvert les architectures matérielles des générateurs réalisant l'algorithme dichotomique couvert au chapitre 3 pour l'exponentielle de paramètre $\lambda = 1$ et la normale centrée réduite. Partant d'une architecture universelle bâtie sur une structure collaborative des nœuds, nous avons illustré comment il était possible de réduire le nombre de générateurs uniformes nécessaires au générateur non-uniforme et avons par là-même exprimé un modèle matériel d'exploration de la corrélation sérielle au travers du modèle de générateur d'exponentielle. Pour ce dernier, nous avons proposé non pas une architecture mais plusieurs dont le chapitre 5 explore l'étude empirique. Nous avons également procédé à la définition de trois modèles de générateurs de gaussienne dont deux ne requierent aucune fonction arithmétique matérielle explicite.

CHAPITRE 5

IMPLÉMENTATIONS ET RÉSULTATS EXPÉRIMENTAUX

5.1 Introduction

Maintenant que notre modèle mathématique est posé et que nous avons défini un nombre d'architectures matérielles à étudier, nous allons procéder à la présentation des résultats expérimentaux obtenus lors de la mise en pratique du bagage théorique couvert. Dans un premier temps, nous allons procéder au survol des environnements de développements que nous avons adoptés, tant logiciels que matériels. Tel que l'exposait la section 3.4.4, nos modèles d'architecture sont dans un premier temps validés dans l'environnement logiciel de prototypage rapide Matlab. Nous procédons ensuite à l'étude du comportement des architectures dans leur contexte matériel avant de conclure par la discussion des résultats obtenus à la lumière d'une application de type Monte Carlo que nous aurons soin de présenter.

5.2 Simulation algorithmique

Il est une bonne pratique de concepteur FPGA que de valider toute architecture matérielle dans un environnement de simulation logiciel. On préférera dans le cadre du travail de prototypage des langages de programmation scriptés, précompilés, disposant de librairies à la fois riches et éprouvées. Ce sont ces critères qui favorisent et popularisent l'utilisation des scripts Matlab dont nous ferons usage ici.

Il est en effet possible de décrire en quelques lignes de code Matlab des comportement algorithmiques très complexes. De plus, MATLAB peut aisément être couplé au logiciel Simulink et à la boîte à outils (*toolbox*) System Generator de Xilinx, avec pour résultat une proximité encore plus tangible de la simulation d'avec le comportement réel du circuit puisque la boîte à outils de Xilinx permet une simulation exacte au coup d'horloge près.

Le travail que nous avons entrepris consiste à valider au moyen des tests statistiques les algorithmes de génération basés sur les RBB que nous avons discutés au chapitre 3. Nous avons également considéré l'impact qu'avait notre choix de n'utiliser qu'un seul générateur uniforme sur la corrélation sérielle. Ce faisant, nous avons exploré les différents scénarios de configuration présentés à la section 4.2.2. Pour cette partie du travail, l'architecture extrêmement épurée du générateur d'exponentielle fut exploitée, comme nous le mentionnions à la section 4.3.1.

5.2.1 Sommaire des sources Matlab

L'annexe I liste intégralement les sources Matlab utilisées pour le présent travail. Nous les avons répertoriées en quatre catégories principales que nous énumérons ici :

Architecture universelle : Nous avons dans un premier temps décrit en Matlab un RBB à 16 nœuds générique permettant de simuler toute les distributions de probabilité représentée sur 16 bits. La section I.1 donnée à l'annexe I liste l'ensemble des fichiers décrivant ce générateur. Il s'agit davantage d'un prototype générique que d'une architecture autonome. Ce patron sert à l'écriture des codes simulant les générateurs d'exponentielle et de normale.

Test du χ^2 : Le test du χ^2 est une mesure permettant de qualifier la similarité d'une courbe mathématique d'une loi de probabilité et un histogramme empirique. La

théorie entourant le test du χ^2 fut discutée à la section 2.4. Le protocole que nous avons suivi pour la mesure de la *p-value* est décrit à la section 3.4.4 et l'ensemble des sources utilisées pour y parvenir sont listés en annexe à la section I.2.

Le générateur de l'exponentielle : Les sources du générateur d'exponentielle, listées en annexes à la section I.3, sont une modification des sources de la section I.1 avec un changement apporté à la fonction *mtx*() qui reflète l'emploi d'une ROM compacte de 32×32. On y retrouve également une paramétrisation de l'étage de sortie permettant l'étude de l'effet de l'emploi d'un seul générateur uniforme.

Le générateur de la normale : Les sources du générateur de la normale, listées en annexes à la section I.4 sont une modification des sources de la section I.1 avec un changement apporté à la fonction *mtx*() qui reflète l'étude de l'implémentation arithmétique des nœuds, discutée à la section 4.4.2.

5.2.2 Corrélation sérielle

Nous considérons trois scénarios pour l'étage de sortie de générateur de l'exponentielle :

Scénario A : Les longueurs des registres à décalage sont différentes et diffèrent de 1.

Scénario B : Les longueurs des registres à décalage sont différentes et diffèrent de 2.

Scénario C : Les longueurs des registres à décalage sont différentes et diffèrent d'une quantité qui croît de 1 (suite algébrique : 1, 2, 3, 4, etc).

Du point de vue du strict respect de la loi de probabilité, les trois scénarios semblent au-dessus de tout soupçon comme le suggère trompeusement la figure 5.1. Cette figure illustre comme nous alors le voir la coexistence d'une part du strict respect de la loi de probabilité et de la corrélation sérielle. Nous avions déjà observé la chose au chapitre 1 lorsque nous discutions du RANDU et des LFSR. De la même façon, la sortie d'un

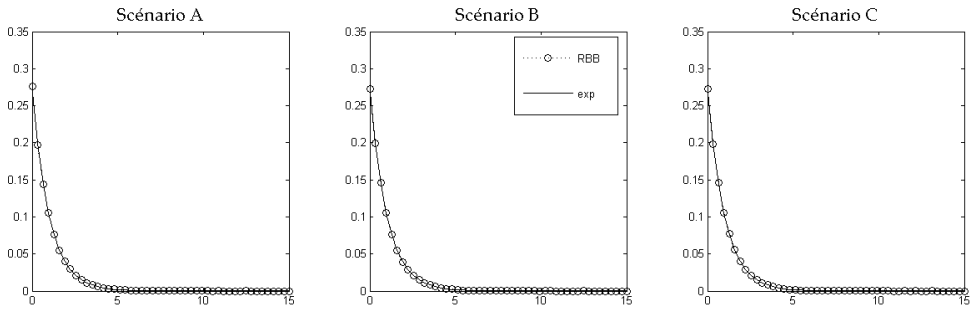


FIG. 5.1 Exponentielle obtenue par RBB suivant les scénarios A, B et C.

compteur binaire tournant en boucle est uniformément répartie et respecte donc la loi de probabilité uniforme, mais *la séquence* de ces nombres peut difficilement être qualifiée d'aléatoire. C'est là un paradoxe inhérent au domaine des nombres pseudo-aléatoires que nous n'exhumerons pas davantage.

Considérons la figure 5.2 où sont tracés les nuages de points correspondant aux séquences $(i, i+1)$, $(i, i+2)$ et $(i, i+3)$ du *Scénario A* — équivalent du scénario #1 décrit à la section 4.2.2. On y constate un patron dans le cas des séquences $(i, i+1)$ et $(i, i+2)$. Ce scénario de configuration de l'étage de sortie ne peut clairement pas être adopté, tout comme ne peut l'être le chemin de donnée non registré de la figure 4.4.a.

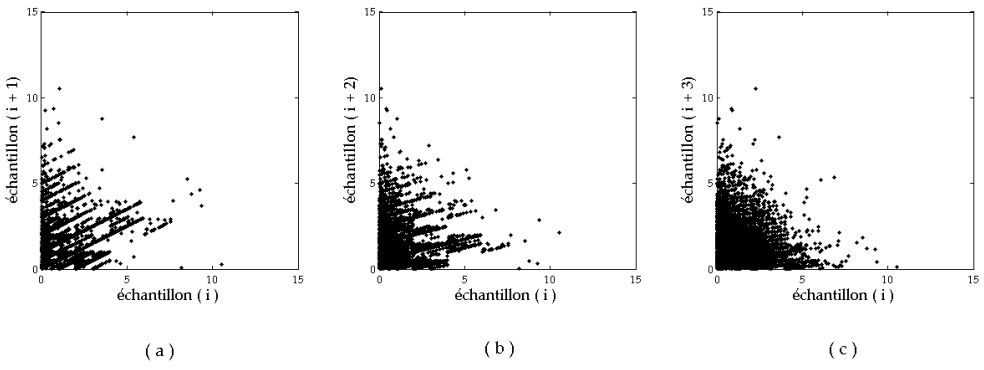


FIG. 5.2 Corrélation sérielle : Scénario A.

Nous appliquons le test de corrélation sérielle au *Scénario B* — équivalent au scénario #3 de la section 4.2.2 — dont les résultats sont donnés à la figure 5.3. La séquence $(i, i+1)$ est correcte, de même que la séquence $(i, i+3)$, tandis que la séquence $(i, i+2)$ est fautive comme nous le laissait soupçonner les résultats de la table 4.1.

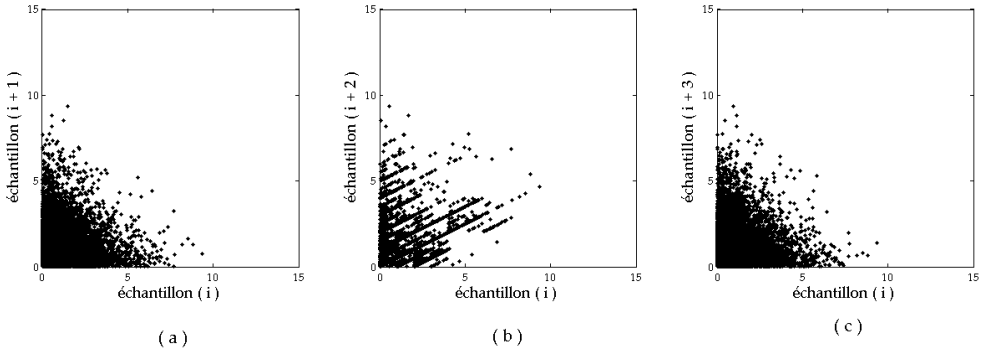


FIG. 5.3 Corrélation sérielle : Scénario B.

Selon les résultats de la table 4.1, la corrélation sérielle est induite par le partage des nombres aléatoires entre les bits des échantillons successifs. Pour atténuer cet effet, nous recourant à une différenciation en série algébrique des registres de sortie, de sorte que le chemin du bit k ait une longueur de $k(k-1)/2$. La figure 5.4 illustre le succès de cette telle approche.

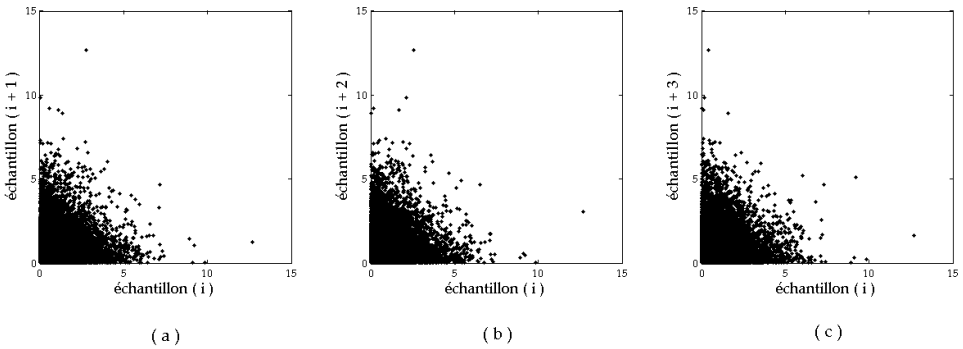


FIG. 5.4 Corrélation sérielle : Scénario C.

Afin de qualifier le générateur, nous avons exécuté une quarantaine de tours de génération de $N = 150\ 000$ échantillons et évalué la *p-value* en segmentant l'abscisse de manière équidistribuée tel que préconisé par D'Agostino et Stephens [14]. La figure 5.5 présente les *p-value* obtenues triées par ordre de grandeur et montre clairement que le générateur RBB passe correctement le test. De plus, on voit bien que la *p-value* couvre le spectre des valeurs allant de 0.0 à 1.0 (-1% aux extrémités) comme le soulignait très justement L'Écuyer [30].

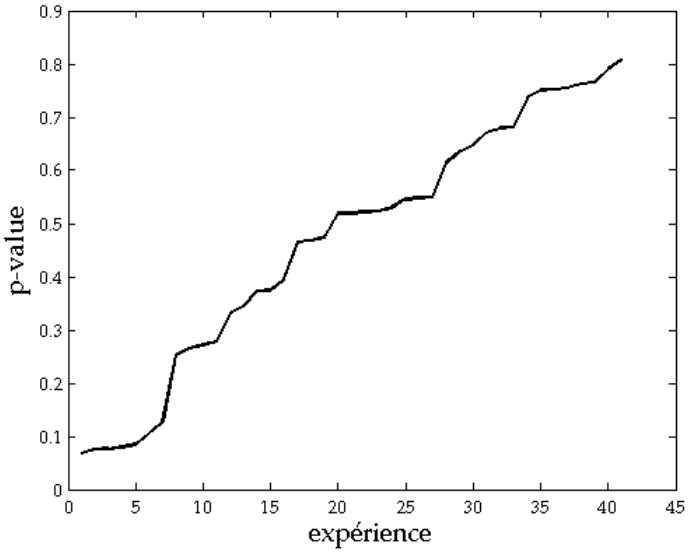


FIG. 5.5 Résultats du test du χ^2 pour le générateur d'exponentielle.

5.2.3 Implémentation arithmétique des nœuds

Nous le mentionnions à la section 4.4.2, la fonction nodale du générateur de la normale de type RBB est linéaire sauf pour les nœuds de premier ordre. La table 4.2 nous donnait les paramètres α_i et β_i obtenus par régression linéaire pour une couverture symétrique de $\pm 8\sigma$. Nous avons employé les scripts de la section I.4 pour évaluer la qualité statistique

des générateurs de gaussienne de type RBB de 16 nœuds et couvrant x symétriquement sur $\pm 4\sigma$.

La figure 5.6 présente les résultats du test du χ^2 effectué pour l'ensemble de ces implémentations. L'indice donné en abscisse renvoie à celui du premier nœud linéarisé. Le test du chi^2 a été effectué en utilisant $N = 150\ 000$ échantillons par tour et en exécutant une quarantaine de tours par implémentation. La courbe de la figure 5.6 parcoure la moyenne des *p-value* obtenues sur les tours tandis que la zone ombragée trace le spectre des valeurs obtenues (retranché du tiers aux extrémités).

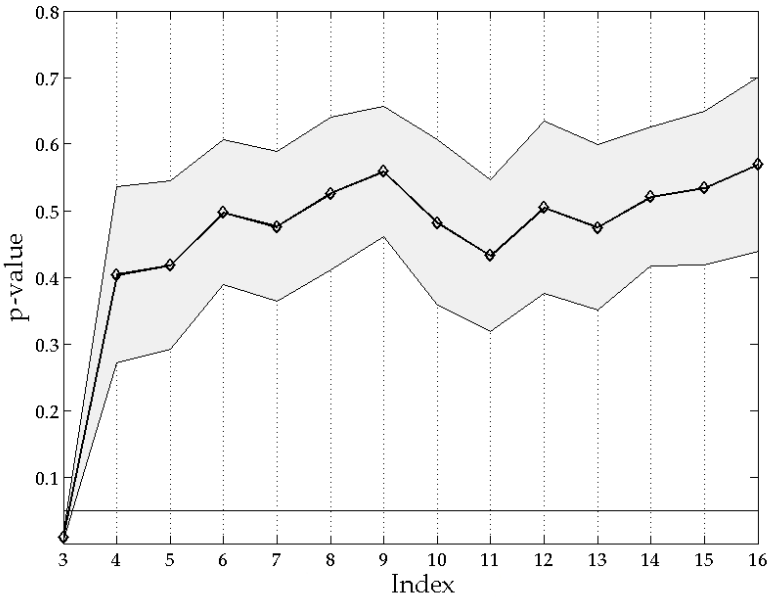


FIG. 5.6 Résultats du test du χ^2 pour l'implémentation arithmétique de la normale.

On voit clairement que l'implémentation arithmétique passe le test du χ^2 sauf dans le cas de l'indice 3. Mentionnons pour les besoins de la précision que ces indices sont décalés d'une unité par rapport à ceux de la table 4.2. Ainsi l'indice 3 de la figure 5.6 est donné comme 4 à la table 4.2.

5.3 Implémentation matérielle

Les implémentations matérielles présentées ont été testées sur le FPGA Virtex II Pro de Xilinx. Les temps de fréquences maximales sont ceux donnés par le logiciel ISE 10.1 de Xilinx après placement et routage. Lorsque la source VHDL est disponible, elle est donnée en annexe et le listing correctement référée dans le texte. Les générateurs exploitent le génréteur uniforme Tausworthe (annexe II.1) combiné dont la période est de 2^{88} et qui occupe 141 *slices*.

5.3.1 Générateur de l'exponentielle

Nous avons implémenté le générateur d'exponentielle de deux manières différentes. Les descriptions matérielles sont données en VHDL à la section II.2 :

Implémentation à ROM compact : Où nous avons utilisé une différenciation en série algébrique des registres de sortie afin de refléter le résultat de la figure 5.4. Le code de description matérielle écrit en VHDL de l'entité est donné au listing II.5.

Implémentation à multiple Bernoulli : Où nous avons mis en parallèle 32 générateurs de Bernoulli ayant une implémentation de van Daalen. Le code de description matérielle écrit en VHDL de l'entité est donné au listing II.11.

La ROM du premier générateur est inférée par le synthétiseur de ISE et convertie en Block ROM, laissant une occupation de 336 *slices* pour l'ensemble du design (ce qui inclut le générateur Tausworthe) et une fréquence maximale annoncée de 225 MHz. Le second générateur a une occupation de 906 *slices* pour une fréquence de fonctionnement maximale de 271 MHz. La différence de taille entre les deux générateurs peut s'expliquer par le fait que dans le second cas, nous avons affaire à une implémentation précise au bit près, de type *fine grained* [12], beaucoup plus gourmande en ressources reconfigurables.

Néanmoins, nous avons également synthétisé un générateur d'exponentielle de ce type avec une résolution de 16 bits. La fréquence de fonctionnement maximale est demeurée identique tandis que la consommation de ressources reconfigurables a baissé à 281.

En ce qui a trait aux performances statistiques. Nous avons recueilli $20 \cdot 10^6$ d'échantillons et effectué le test du χ^2 sur cette population en segmentant l'axe des abscisses en $1\ 665$ régions équidistribuées. Le générateur (implémentation à ROM compacte, et 32 bits de résolution) a passé le test avec une *p-value* de 0.6548. Nous avons également extrait $4 \cdot 10^6$ d'échantillons et effectué le test du χ^2 sur cette nouvelle population en segmentant l'axe des abscisses en 100 régions équidistantes. Le générateur a passé le test avec une *p-value* de 0.9323.

5.3.2 Résultats obtenus pour la distribution normale

Nous avons implémenté le générateur de la normale de deux manières différentes. Les descriptions matérielles sont réalisées en schéma Simulink au moyen de la librairie *System Generator* de Xilinx.

Implémentation stochastique : Où nous avons utilisé des opérateurs stochastiques pour la multiplication et l'addition comme nous le discutions à la section 5.4.

Implémentation à décalage des parents : Où nous avons utilisé l'approximation obtenue par le décalage de l'état des parents.

Les multiples générateurs de Bernoulli utilisés dans l'implémentation stochastique sont réalisés à l'aide d'une ROM, comme nous le faisions pour le générateur d'exponentielle. La linéarisation est réalisée à compter du nœud 7 pour minimiser les erreurs d'approximation. Le générateur occupe la même surface pour une résolution de 16 ou de 32 bits, les 16 bits supplémentaires nécessaires pour passer du premier au second sont extraits

du générateur uniforme. Le synthétiseur de ISE réalise l'implémentation en utilisant 540 *slices* et un bloc RAM pour l'ensemble du design (incluant les deux générateurs Tausworthe). La fréquence maximale annoncée de 225 MHz.

Le second générateur a une occupation de 347 *slices* pour une fréquence de fonctionnement maximale de 225 MHz. Ici encore, le passage d'une implémentation 16 bits à 32 bits ne nécessite aucune occupation de surface supplémentaire.

En ce qui a trait aux performances statistiques. Nous avons recueilli $20 \cdot 10^6$ d'échantillons et effectué le test du χ^2 sur cette population en segmentant l'axe des abscisses en $1\,665$ régions équidistribuées. Les deux générateurs échouent le test, visiblement à cause de l'inexactitude de l'approximation stochastique des fonctions nodales. Cependant, il est intéressant de relever que les deux générateurs ont passé le test du chi^2 pour une population de $4 \cdot 10^6$ d'échantillons et en segmentant l'axe des abscisses en 100 régions équidistantes (comme le faisaient les auteurs Lee et al. [36]) avec un *p-value* de 0,3517 et 0,8029 respectivement. Aussi peut-on considérer que ces générateurs sont qualifiés même si ils ne réussissent pas aussi bien que nos architectures de l'exponentielle.

Nous reprenons ici la table 2.2 en incluant nos propres résultats (l'implémentation stochastique des nœuds est référée comme (1), la seconde par (2)) :

TAB. 5.1 Récapitulatif des principaux générateurs de gaussienne matériels.

Référence	[69]	[35]	[10]	[36]	[3]	(1)	(2)
Résolution (bits)	32	24	16	16	32	16/32	16/32
Fréquence max (MHz)	170	155	232	233	269	225	225
Nb de *slices*	891	770	548	1528	576	540	347
Nb de bloc RAM	4	6	2	3	2	1	0
Nb de multiplieur 18×18	2	4	2	12	3	0	0

5.4 Accélération matérielle

Afin d'illustrer l'intérêt d'accélérer matériellement les méthodes Monte Carlo et de mesurer la qualité des générateurs autrement que par les tests statistiques, nous nous sommes intéressé à un algorithme Monte Carlo simple dont la source est librement distribuée sur l'Internet par le centre médical de l'Oregon (c.f. annexe III).

L'algorithme simule la propagation de photons dans un milieu isotropique. On dispose d'un ensemble de photons (ligne 31 du listing III.1) et on associe à chaque photon un poids unitaire (ligne 35). Les photons se déplacent dans l'espace à partir de l'origine (ligne 33). Chaque déplacement est dicté par une loi exponentielle (ligne 40) suivant une orientation aléatoire en choisissant un point au hasard sur la sphère unitaire (lignes 61-75, voir Marsaglia pour une description du problème [38]). À chaque fois que le photon se déplace, sa masse est réduite et convertie en chaleur qu'absorbe le milieu isotropique (lignes 53 et 56). Le programme observe le réchauffement radial du milieu (ligne 53).

L'algorithme prend $1\ 050\ ms$ pour s'exécuter sur un Core 2 Duo cadencé à 2.0 GHz. L'accélération matérielle est atteinte en recherchant une rentabilisation de chaque coup de l'horloge. Nous visons la carte Amirix AP1100 munie d'un Virtex II Pro VP 100 de Xilinx et cadencée à 100 MHz. La première amélioration réalisée consistait à améliorer la génération uniforme d'un point sur la sphère unitaire (lignes 61-75). L'algorithme utilisé là est une méthode de rejet générant $\pi/4 \simeq 78.5\%$ points valides. Afin d'augmenter ce ratio, nous utilisons trois générateurs en parallèles ce qui donne un taux de rejet de $(1 - \pi/4)^3 \simeq 0.99\%$.

Le calcul du poids et celui de la position du photon se font indépendamment et des *fifos* sont utilisées pour synchroniser les processus. La table *heat[]* est implémentée à l'aide d'une RAM à double accès pour permettre l'incrémentation itérative (ligne 53). Cependant, pour éviter des conflits d'adressage, on utilise à tour de rôle trois de ces

RAM, et les résultats des trois RAM sont réunis à la fin de l'algorithme. La figure 5.7 compare le résultat obtenu par le FPGA à celui donné par l'algorithme du listing I.14. Comme on peut s'en rendre compte, il existe de légères variations dues notamment au calcul des racines carrées que nous avons implémentées au moyen de cellules CORDIC.

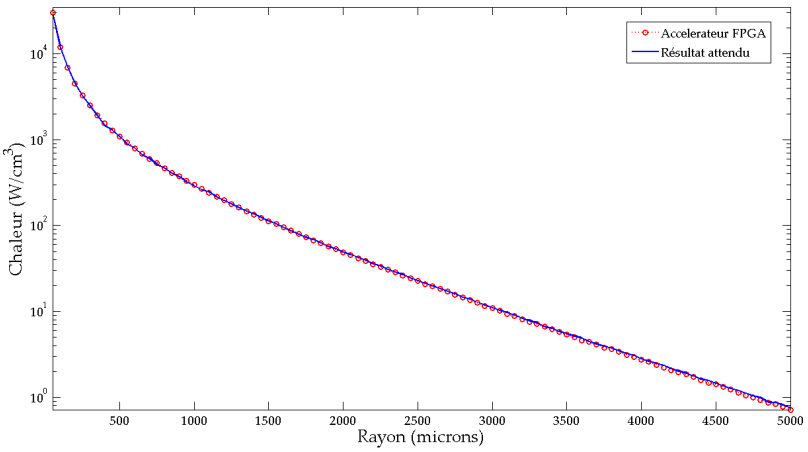


FIG. 5.7 Résultats de l'algorithme Monte Carlo comparé à celui obtenu grâce au FPGA.

L'algorithme prend environ 82 ms pour s'exécuter, ce qui correspond à un facteur d'accélération de l'ordre de 12.9. Le système occupe 5 427 *slices* (sur les 44 096 disponibles), 10 blocs RAM et 56 multiplieurs (sur les 444 disponibles). Il est donc envisageable de mettre de multiples instances du système sur la même puce et d'augmenter d'autant le facteur d'accélération.

5.5 Conclusion

Les résultats présentés dans ce chapitre sont plutôt concluants. D'une part, la simulation algorithmique montre que l'algorithme dichotomique que nous avons présenté fonctionne correctement pour l'exponentielle et la gaussienne. Les implémentations ma-

térielles qui en sont issues sont plus compactes que toutes celles rapportées par la littérature, et cela sans compromettre les qualités statistiques ou fréquentielles du générateur.

Plus remarquable encore, le modèle mathématique de l'algorithme dichotomique que nous avons présenté a permis des implémentations ne recourant à aucune fonction arithmétique. Ainsi, les générateurs réalisés ne recourent à aucun bloc spécialisé (le bloc RAM compact mis à part).

Les résultats fréquentiels présentés (limite à 225 Mhz) sont légèrement plus bas que ceux des architectures les plus avancées de la littérature. Cette différence est mineure et s'explique par deux facteurs : 1) les auteurs cités recourent à Synplify pour obtenir les estimations annoncées ; 2) l'encodeur de priorité limite la fréquence de fonctionnement de nos générateurs et une optimisation est à observer à ce niveau.

Des deux générateurs étudiés, le générateur d'exponentielle est celui qui offre le meilleur comportement sur puce, puisque sa fonction nodale ne requiert aucune opération arithmétique. Néanmoins, le générateur de gaussienne proposé est tout fait valable, eu égard aux qualités des générateurs publiés dans la littérature, puisque notre architecture passe le teste du χ^2 lorsqu'on l'effectue de la même façon que le firent les auteurs cités.

Finalement, nous avons illustré l'intérêt de disposer d'un générateur matériel puisque nous avons réussi à accélérer d'un facteur de 13 l'algorithme Monte Carlo du centre médical de l'Oregon.

CONCLUSION GÉNÉRALE

Le travail présenté ici a réuni nombre d'innovations dans le domaine de la génération de distributions non-uniformes sur puce. On citera d'abord l'algorithme dichotomique et les modélisations dans le domaine des réseaux bayésiens et des arbres de décision binaires. Ces modélisations permirent d'exprimer de manière compacte les fonctions nodales décrivant les probabilités conditionnelles aux nœuds et d'offrir un outil analytique important.

Le modèle et l'algorithme furent appliqués aux distributions normale et exponentielle avec succès. Les simulations empiriques ont par ailleurs montré la grande qualité du comportement des générateurs. Cette qualité peut être créditée au compte du changement de paradigme entrepris par rapport à ce qui se fait généralement dans la littérature. De plus, notre méthode permit une économie importante sur le plan du nombre de générateurs uniformes nécessaires.

Les implémentations matérielles des générateurs de l'exponentielle et de la normale que nous avons proposées méritent une analyse approfondie. D'une part, ils offrent des implémentations compactes pour des réalisations à 16 bits et à 32 bits. De plus, les implémentations ne recourent à aucun opérateur arithmétique grâce à l'analyse adéquate des fonctions nodales qui leur sont associées.

Ainsi, le générateur d'exponentielle offre un comportement admirable à tout point de vue : consommation des ressources, fréquence d'opération, résolution de la variable aléatoire, qualité statistique et corrélation. Mentionnons à ce sujet l'importance du générateur de Bernoulli que nous avons proposé et qui a permis une importante économie de surface sur puce qui devint à nos yeux criarde lorsque la cellule de van Daalen fut utilisée pour le générateur d'exponentielle.

Le générateur de la normale quant à lui connaît une faiblesse du point de vue de la qualité statistique même si toutefois il n'est pas moins bon qu'aucun des générateurs présentés dans la littérature.

Finalement, une application de type Monte Carlo fut accélérée avec succès au moyen d'un FPGA en utilisant notre générateur d'exponentielle, cela avec pour résultat un facteur d'accélération de l'ordre de 13 par rapport à une exécution logicielle sur un processeur Intel à usage général cadencé à 2.0 GHz.

RÉFÉRENCES

[1] ALIMOHAMMAD, A., COCKBURN, B., AND SCHLEGEL, C. An iterative hardware gaussian noise generator. In *IEEE Pacific Rim Conference on Communications, Computers and signal Processing* (August 2005), pp. 649–652.

[2] ALIMOHAMMAD, A., COCKBURN, B., AND SCHLEGEL, C. Area-efficient parallel white gaussian noise generator. In *IEEE Canadian Conference on Electrical and Computer Engineering* (May 2005), pp. 1872–1875.

[3] ALIMOHAMMAD, A., FARD, S., COCKBURN, B., AND SCHLEGEL, C. A compact and accurate gaussian variate generator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 16*, 5 (May 2008), 517–527.

[4] BI, Y., PETERSON, G. D., WARREN, G. L., AND HARRISON, R. J. Hardware acceleration of parallel lagged-fibonacci pseudo random number generation. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms* (2006).

[5] BOUTILLON, E., DANGER, J., AND GAZEL, A. Design of high speed awgn communication channel emulator. *Analog Integrated Circuits and Signal Processing 34*, 2 (2003), 133–142.

[6] BOWER, J. A., THOMAS, D. B., LUK, W., AND MENCER, O. A reconfigurable simulation framework for financial computation. In *IEEE International Conference on Reconfigurable Computing and FPGA's* (September 2006), pp. 1–9.

[7] BOX, G., AND MULLER, M. A note on the generation of random normal deviates. *The Annals of Mathematical Statistics 29* (1958), 610–611.

[8] BROWN, B., AND CARD, H. Stochastic neural computation I : Computational elements. *IEEE Transactions on Computers 50*, 9 (Sep 2001), 891–905.

[9] CHEN, J., MOON, J., AND BAZARGAN, K. Reconfigurable readback-signal generator based on a field-programmable gate array. *IEEE Transactions on Magnetics 40*, 3 (March 2004), 1744–1750.

[10] CHEUNG, R. C. C., LEE, D.-U., LUK, W., AND VILLASENOR, J. Hardware generation of arbitrary random number distributions from uniform distributions via the inversion method. *IEEE Transactions on Very Large Scale Integration Systems 15*, 8 (August 2007), 952–962.

[11] CHUL, P., AND FRANTZ, B. A reprogrammable fpga-based atm traffic generator. In *Proceedings of the Sixth Great Lakes Symposium on VLSI* (March 1996), pp. 35–38.

[12] COWEN, C., AND MONAGHAN, S. A reconfigurable Monte-Carlo clustering processor (MCCP). In *IEEE Workshop on FPGAs for Custom Computing Machines* (1994), pp. 59–65.

[13] CUI, W., LI, C., AND SUN, X. FPGA implementation of universal random number generator. In *Proceedings of the 7th International Conference on Signal Processing* (2004), vol. 1, pp. 495–498.

[14] D'AGOSTINO, R. B., AND STEPHENS, M. A. *Goodness-of-Fit Techniques*. Marcel Dekker, 1986.

[15] DEVROYE, L. *Non-Uniform Random Variate Generation*. Springer-Verlag, New York, 1986.

[16] FAN, Y., AND ZILIC, Z. A novel scheme of implementing high speed awgn communication channel emulators in fpgas. In *Proceedings of the IEEE International Symposium on Circuits and Systems* (May 2004), vol. 2.

[17] FAN, Y., ZILIC, Z., AND CHIANG, M. A versatile high speed bit error rate testing scheme. In *Proceedings of the IEEE International Symposium on Quality Electronic Design* (2004), pp. 395–400.

[18] GAINES, B. Stochastic computing systems. In *Advances in Information Systems Science* (1969), vol. 2, pp. 37–172.

[19] GALASSI, M., DAVIES, J., THEILER, J., GOUGH, B., JUNGMAN, G., BOOTH, M., AND ROSSI, F. *GNU Scientific Library Reference Manual*. Pub-Network-Theory, 2006.

[20] GOLOMB, S. *Shift Register Sequences, Revised Edition*. Ae-gean Park Press, 1982.

[21] GOTHANDARAMAN, A., WARREN, G., PETERSON, G., AND HARRISON, R. Hardware acceleration of a quantum Monte Carlo application. In *Proceedings of the Third Annual Reconfigurable Systems Summer Institute* (July 2007).

[22] HENRION, M. Propagating uncertainty in bayesian networks by probabilistic logic sampling. In *Uncertainty in Artificial Intelligence, second edition* (New York, 1988), J. Lemmer and L. Kanal, Eds., Elsevier Science, pp. 149–163.

[23] JEAVONS, P., AND SHAWE-TAYLOR, D. C. J. Generating binary sequences for stochastic computing. *IEEE Transactions on Information Theory 40*, 3 (1994), 716–720.

[24] KNUTH, D. E. *Art of Computer Programming, Volume 2 : Seminumerical Algorithms (3rd Edition)*. Addison-Wesley Professional, 1997.

[25] KNUTH, D. E., AND YAO, A. C. The complexity of non-uniform random number generation. In *Algorithms and Complexity* (New York, 1976), Academic Press, pp. 357–428.

[26] L'ECUYER, P. Uniform random number generation. *Annals of Operations Research 53* (1994), 77–120.

[27] L'ECUYER, P. Maximally equidistributed combined tausworthe generators. *Mathematics and Computation 65*, 213 (1996), 203–213.

[28] L'ECUYER, P. Random number generation. In *Elsevier Handbooks in Operations Research and Management Science : Simulation* (Amsterdam, 2006), Elsevier Science, pp. 55–81.

[29] L'ECUYER, P., AND PANNETON, F. Fast random number generators based on linear recurrences modulo 2 : Overview and comparison. In *Proceedings of the 2005 Winter Simulation Conference* (2005), pp. 110–119.

[30] L'ECUYER, P., AND SIMARD, R. TestU01 : A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software 33*, 4 (August 2007).

[31] LEE, D.-U., LUK, W., VILLASENOR, J., AND CHEUNG, P. Hierarchical segmentation schemes for function evaluation. In *Proceedings of the IEEE International conference on Field-Programmable Technology* (2003), pp. 92–99.

[32] LEE, D.-U., LUK, W., VILLASENOR, J., AND CHEUNG, P. Y. K. Hardware function evaluation using non-linear segments. In *Proceedings of International conference on Field Programmable Logic and Applications* (2003), pp. 796–807.

[33] LEE, D.-U., LUK, W., VILLASENOR, J. D., AND CHEUNG, P. A hardware gaussian noise generator for channel code evaluation. In *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* (2003), pp. 69–78.

[34] LEE, D.-U., LUK, W., VILLASENOR, J. D., AND CHEUNG, P. A gaussian noise generator for hardware-based simulations. *IEEE Transactions on Computers 53*, 12 (December 2004), 1523–1534.

[35] LEE, D.-U., LUK, W., VILLASENOR, J. D., ZHANG, G., AND LEONG, P. A hardware gaussian noise generator using the wallace method. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 13*, 8 (Aug. 2005), 911–920.

[36] LEE, D.-U., VILLASENOR, J., LUK, W., AND LEONG, P. A hardware gaussian noise generator using the box-muller method and its error analysis. *IEEE Transactions on Computers 55*, 6 (June 2006), 659–671.

[37] MARSAGLIA, G. Random numbers fall mainly in the planes. In *Proceedings of the national academy of science* (1968), vol. 61, pp. 25–28.

[38] MARSAGLIA, G. Choosing a point from the surface of a sphere. *Annals of Mathematical Statistics 43*, 2 (1972), 645–646.

[39] MARSAGLIA, G. The structure of linear concruential sequences. In *Applications of number theory to numerical analysis* (London, 1972), S. K. Zaremba, Ed., Ed. Academic Press, pp. 249–285.

[40] MARSAGLIA, G. DIEHARD : a battery of tests of randomness. Tech. rep., State University of Florida, 1996.

[41] MARSAGLIA, G., AND TSANG, W. W. The ziggurat method for generating random variables. *Journal of Statistical Software 5*, 8 (2000).

[42] MARSAGLIA, G., AND TSAY, L. Matrices and the structure of random number sequences. *Linear Algebra and its Applications 67* (1985), 147–156.

[43] MARTIN, P. An analysis of random number generators for a hardware implementation of genetic programming using FPGAs and Handel-C. In *Proceedings of the Genetic and Evolutionary Computation Conference* (San Francisco, CA, USA, 2002), Morgan Kaufmann Publishers Inc., pp. 837–844.

[44] MATSUMOTO, M., AND NISHIMURA, T. Mersenne twister : A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation 8*, 1 (January 1998), 3–30.

[45] MCCOLLUM, J., LANCASTER, J., BOULDIN, D., AND PETERSON, G. Hardware acceleration of pseudo-random number generation for simulation applications. In *Proceedings of the 35th Southeastern Symposium on System Theory* (March 2003), pp. 299–303.

[46] MOLER, C. *Numerical Computing with MATLAB*. SIAM, 2004.

[47] MONAGHAN, S. A gate-level reconfigurable Monte Carlo processor. *J. VLSI Signal Process. Syst. 6*, 2 (1993), 139–153.

[48] NEGOI, A., AND ZIMMERMANN, J. Monte Carlo hardware simulator for electron dynamics in semiconductors. In *Proceedings of the International Annual Semiconductor Conference* (1996), pp. 557–560.

[49] OSANA, Y., FUKUSHIMA, T., YOSHIMI, M., IWAOKA, Y., FUNAHASHI, A., HIROI, N., SHIBATA, Y., KITANO, H., AND AMANO, H. An FPGA-based, multi-model simulation method for biochemical systems. In *Proceedings of the IEEE Transactions on International Parallel and Distributed Processing Symposium* (April 2005).

[50] PEARL, J. *Probabilistic Reasoning in Intelligence Systems, , Second edition*. Morgan Kaufmann, 1988.

[51] RUSSEL, S. J., AND NORVING, P. *Artificial Intelligence : A Modern Approach, Second edition*. Prentice Hall, 2003.

[52] SADASIVAM, M. M., WAGHOLIKAR, J., AND HONG, S. A look-up based low-complexity parallel noise generator for particle filter processing. *International Symposium on Signals, Circuits and Systems 2* (Juy 2003), 621–624.

[53] SHAWE-TAYLOR, J., JEAVONS, P., AND DAALEN, M. Probabilistic bit stream neural chip : Theory. *Connection Science 3*, 3 (1991), 317–328.

[54] TAUSWORTHE, R. C. Random numbers generated by linear recurrence modulo two. *Mathematics and Computation 19*, 90 (1965), 201–209.

[55] THOMAS, D. B., BOWER, J. A., AND LUK, W. Hardware architectures for Monte Carlo based financial simulations. In *IEEE International Conference on Field Programmable Technology* (December 2006), pp. 377–380.

[56] THOMAS, D. B., AND LUK, W. Efficient hardware generation of random variates with arbitrary distributions. In *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* (2006), pp. 57–66.

[57] THOMAS, D. B., AND LUK, W. High quality uniform random number generation using LUT optimised state-transition matrices. *The Journal of VLSI Signal Processing 47*, 1 (April, 2007), 77–92.

[58] THOMAS, D. B., AND LUK, W. Non-uniform random number generation through piecewise linear approximations. In *International Conference on Field Programmable Logic and Applications* (August 2006), pp. 1–6.

[59] TIMARCHI, S., MIREMADI, S., AND EJLALI, A. Evaluation of some exponential random number generators implemented by FPGA. In *Proceedings of the International Multi-Conference on Parrallel and Distributed Computing and networks* (Innsbruck, Austria, February 2005), pp. 578–583.

[60] VAN DAALEN, M., JEAVONS, P., AND SHAWE-TAYLOR, J. Probabilistic bit stream neural chip : Implementation. *Oxford Workshop on VLSI for Artificial Intelligence and Neural Networks* (September 1990), 285–294.

[61] VAN DAALEN, M., JEAVONS, P., SHAWE-TAYLOR, J., AND COHEN, D. Device for generating binary sequences for stochastic computing. *Electronics Letters 29*, 1 (1993), 80–81.

[62] VON NEUMANN, J. Various techniques used in connection with random digits. *Applied Mathematics Serires 12* (1951), 36–38.

[63] WALLACE, C. Fast pseudorandom generators for normal and exponential variates. *ACM Transactions on Mathematical Software 22*, 1 (1999), 119–127.

[64] XILINX. *XAPP052 : Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators*, 1996. Xilinx Application Note.

[65] XILINX. *XAPP210 : LFSRs in Virtex Devices*, 1999. Xilinx Application Note.

[66] XILINX. *DS210 : Additive White Gaussian Noise (AWGN) Core v1.0*, 2002. Xilinx Data Sheet.

[67] YOSHIMI, M., OSANA, Y., FUKUSHIMA, T., AND AMANO, H. Stochastic simulation for biochemical reactions on FPGA. In *International Conference on Field Programmable Logic and Applications* (2004), pp. 105–114.

[68] ZHANG, G., LEONG, P., HO, C., TSOI, K., CHEUNG, C., LEE, D.-U., CHEUNG, R., AND LUK, W. Reconfigurable acceleration for Monte Carlo based financial simulation. In *IEEE International Conference on Field-Programmable Technology* (December 2005), pp. 215–222.

[69] ZHANG, G., LEONG, P., LEE, D.-U., VILLASENOR, J., CHEUNG, R., AND LUK, W. Ziggurat-based hardware gaussian random number generator. In *International Conference on Field Programmable Logic and Applications* (Auguste 2005), pp. 275–280.

ANNEXE I

SOURCES MATLAB

L'annexe I comporte l'ensemble des scripts MATLAB utilisés dans ce travail et discutés au chapitre 5. Un descriptif rapide précède script et le code a été commenté au mieux.

I.1 Générateur universel

Suivent ici les *listing* des scripts et fonctions MATLAB d'un générateur universel de type RBB à 16 nœuds. Ce code est ensuite remanié pour les besoins des distributions exponentielle et normale.

Le prototype d'un appel à un générateur de type RBB à 16 nœuds :

Listing I.1 Appel générique d'un générateur de type RBB à 16 noeuds

```
%-------------------------------------------------------------
% FUNCTION : VALUES = UNIV_GEN(NB_ECHANTILLONS)
% Fonction générique qui simule un RBB à 16 noeuds
%  et génère NB_ECHANTILLONS échantillons qui en
%  sont issus
%
% NOM DE FICHIER: UNIV_GEN.M
%
% INPUT:
% NB_ECHANTILLONS: Nombre d'échantillons à générer
%
% OUTPUT:
% VALUES : Vecteur comportant les NB_ECHANTILLONS
%          échantillons aléatoires générés.
%
% NOTE:
% La fonction params doit être configuré correctement
% pour chaque nouvelle distribution.
%-------------------------------------------------------------

function values = univ_gen(nb_echantillons)

% Variables globales;
global m;
global nombres;

% lire les CPT et convertir la valeur de probabilité
% de [0.0 1.0] en un entier 32 bits
m = uint32(2^32*mtx());

%nombres de noeuds
niveaux = 16;

% pipe des valeurs (sortie de chaque noeuds)
nombres = uint16(zeros(niveaux+1,1));
```

```
36
   % pipe des (indices) nomrbres aléatoires
   idx = zeros(niveaux,1);

   % remplir le pipe des (indices) nombres aléatoires
41 for i=1:length(idx)
       idx(i) = index( uint32(2^32*rand())) ;
   end

   % Nombre d'itérations
46 % (on ajoute le nb de noeuds pour remplir le pipe)
   nb_loop = nb_echantillons + niveaux + 1;

   % cree la variable
   values = 0;
51
   % boucle sur les echantillons
   for i = 1:nb_loop

       % Récupérer la valeur à la sortie
56   % du pipe si il est plein
       if( i > niveaux+1)
           u = nombres(niveaux);
           values(i-niveaux-1) = u;
       end
61
       % chaque noeud génère un bit
     % en fonction des nombres aléatoires générés
     % et on concatène le tout avec el valeurs
     % existantes
66     nombres = gennombre(nombres, m, niveaux, idx);

       % decaler les chiffres (shift register)
       nombres(2:niveaux) = nombres(1:niveaux-1);
       nombres(1) = uint16(0);
71
       % Decaler les (indices) nombres aléatoires
     % mais a contre courant (shift register)
       idx(1:niveaux-1) = idx(2:niveaux);
       idx(niveaux) = index( uint32(2^32*rand()) );
76
   end;
```

La fonction qui retourne la matrice des fonctions nodales du RBB :

Listing I.2 Fonction retournant la matrice des phi functions

```
   %-------------------------------------------------------------
2  % FUNCTION : M = MTX()
   % Fonction générique qui crée et retourne l'ensemble des
   %  fonctions nodales (phi function) d'un RBB à 16 noeuds
   %
   % NOM DE FICHIER: MTX.M
7  %
   % OUTPUT:
   % M : Matrice des phi functions (CPT) d'un RBB de 16
   %     noeuds.
   %
12 % NOTE:
   % La fonction params doit être configuré correctement
   % pour chaque nouvelle distribution.
   %-------------------------------------------------------------

17 function m = mtx()

   % Allocation de memoire pour un RBB de 1 noeuds
   m = zeros(2^(16-1),16);

22 % Parcours des 16 noeuds
   for i = 1:16

      % La fonction params donne la phi function (CPT)
      % correspondant au noeud i et doit être adapté
27    p = params( i );

      % Chaque noeud dispose d'une CPT de taille 2^(i-1)
      j=length(p);
      m( 1:j, i ) = p( 1:j, i );
32
   end
```

La fonction qui imite le comportement d'un encodeur de priorité :

Listing I.3 Fonction encodeur de priorité

```
%-------------------------------------------------------------
% FUNCTION : INDEX = INDEX(UI)
% Fonction generique qui simule le comprotement d'un encodeur
%  de priorite
%
% NOM DE FICHIER: INDEX.M
%
% INPUT:
% UI : Nombre represente en UINT32
%
% OUTPUT:
% INDEX : Une valeur entre 0 et 32 correspondant a
%         a l'indice du bit non nul de UI. 0 pour UI=0
%-------------------------------------------------------------

function index = index(ui)

%initialisation
index=32;

% boucle tant que le bit vaut 0
while(~bitget(ui,index))
  % decremente l'indice
    index=index-1;

  %sort si tous le bits sont a 0
    if(index==0)
        break;
    end;
end;
```

La fonction qui s'occupe de générer les bits de tous les noeuds étant donnés les parents et qui place les bits aux bons endroits :

Listing I.4 Fonction générant les bits de chaque noeud et decalages

```
%-------------------------------------------------------------------
% FUNCTION : NOMBRES = GENNOMBRE(NOMBRES, M, NIVEAUX, IDX)
% Fonction générique qui génère les bits des 16 noeuds
% d'un RBB à 16 noeuds
%
% NOM DE FICHIER: GENNOMBRE.M
%
% INPUT:
% NOMBRES: Pipe des nombres d'échantillons à générer
% M      : Matrice des phi functions
% NIVEAUX: 16 en l'occurence
% IDX    : Pipe des nombres aléatoires (indices des)
%
% OUTPUT:
% NOMBRES : Pipe modifié.
%
% NOTE:
% La fonction params doit être configuré correctement
% pour chaque nouvelle distribution.
%-------------------------------------------------------------------

function nombres = gennombre(nombres, m, niveaux, idx);

for i=1:niveaux
  % cas particulier 0
  if(idx == 0) % (très rarement)
    % decaler a gauche chaque nombre et ajouter un 0
    nombres(1:niveaux-1) = bitshift(nombres(1:niveaux-1), 1);

  % autres cas
  else
    % retrouver la probabilite pour chaque bit
    % (tous les nombres, sauf resultat)
    p = uint32(2^32*parametre_parbit(nombres, m, niveaux));

```

```
    % decaler a gauche de 1 bit chaque nombre (sauf resultat)
    nombres(1:niveaux) = bitshift(nombres(1:niveaux), 1);

    % changer le bit le cas échéant, selon p et idx
    nombres(1:niveaux) = bitset(nombres(1:niveaux), 1, bitgen(p,idx));
  end
end
```

Et finalement la lecture des probabilitées conditionnelles suivant les nœuds parents :

Listing I.5 Fonction générant les probabiites de chaque noeud

```
%-------------------------------------------------------------
% FUNCTION : NOMBRES = PARAMETRE_PARBIT(NIVEAUX)
% Fonction generique qui genere les bits des 16 noeuds
% d'un RBB a 16 noeuds
%
% NOM DE FICHIER: PARAMETRE_PARBIT.M
%
% INPUT:
% NOMBRES: Pipe des nombres d'echantillons a generer
% M      : Matrice des phi functions
% NIVEAUX: 16 en l'occurence
%
% OUTPUT:
% NOMBRES : Les valeurs de probabilite recherchees
%
%-------------------------------------------------------------

function p = parametre_parbit(NOMBRES, M, NIVEAUX)

% Lire les prob conditionnelle fonction de l'etat des parents
p = m(nombres(1:level)+1,1:level);

% Ne garder que les valeurs recherches (twik)
p = diag(p,0);
```

I.2 Test du χ^2

La *p-value* est calculée de différentes manières (types de distribution, méthode de segmentation, etc.), ce qui impose le recours à une fonction paramétrisable :

Listing I.6 Appel paramétrisable au calcul de la *p-value*

```
%-------------------------------------------------------------
% FUNCTION : PVAL = EVALCHISQ(V, BINS, SEG, DIST)
% Fonction qui evalue a p-value obtenue par la methode du chi2
% d'un RBB a 16 noeuds
%
% NOM DE FICHIER: EVALQUISQ.M
%
% INPUT:
% V    : Vecteur d'echantilons aleatoires
% BINS   : Nombres de segments sur l'axe des X
% SEG    : Methode de segmentation -
%                0 : equidistribue
%            Autre : equidistant
% DIST   : Type de distribution -
%                0 : normale
%            Autre : exponentiele
% OUTPUT:
% PVAL   : P-value obtenue par la methode du Chi2
%
%-------------------------------------------------------------
```

```
function pval = evalchisq(v, bins, seg, dist)

[x, fc] = segm( length(v), bins, seg, dist);

% calculer l'histogramme
f = histc(v,x);

% rearranger (twik anti histc)
f = f';
f = f(1:bins);

% calculer la valeur de chi2
chi2 = sum((f - fc).^2./fc);

% calculer la p-value en recourant
% a la cumulative de la distribution chi2
pval = 1-chi2cdf(chi2, bins-1);
```

La segmentation de l'abcisse est effectuée par a fonction *segm* :

Listing I.7 Appel paramétrisable aux segments pour le calcul de la *p-value*

```
% ------------------------------------------------------------
% FUNCTION : [X, FC] = SEGM(NB, K, SEG, DIST)
% Fonction qui definit la segmentation de l'axe des X
% et le nombre d'echantilons par segments attendu
%
% NOM DE FICHIER: SEGM.M
%
% INPUT:
% NB    : Taille du vecteur d'echantilons aleatoires
% K     : Nombres de segments sur l'axe des X
% SEG   : Methode de segmentation -
%                 0 : equidistribue
%             Autre : equidistant
% DIST  : Type de distribution -
%                 0 : normale
%             Autre : exponentiele
%
% OUTPUT:
% X     : P-value obtenue par la methode du Chi2
% FC    : Vecteur du nombre d'echantillons attendu
%
% ------------------------------------------------------------

function [x, fc] = segm(nb, k, seg, dist)

% la distribution est la distribution normale
if( dist == 0 )

  % la segmentation est equidistribuee
  if(seg == 0)
    total = normcdf(8,0,1) - normcdf(-8,0,1);
    x=norminv(normcdf(-8,0,1):total/k:normcdf(8,0,1),0,1);

  % la segmentation est equidistante
  else
    x=-8:16/k:8;
  end

  % surface sous la courbe dans les segments
  p = normcdf(x(2:k+1),0,1) - normcdf(x(1:k),0,1);

% la distribution est l'exponentielle
else
  % la segmentation est equidistribuee
  if(seg == 0)
    total = expcdf(16,1);
    x=expinv(0:total/k:expcdf(16),expcdf(16));

  % la segmentation est equidistante
  else
    x=0:16/k:16;
  end

  % surface sous la courbe dans les segments
  p = expcdf(x(2:k+1),1) - expcdf(x(1:k),1);

end
```

```
fc = p*nb;
```

I.3 Générateur de l'exponentielle

Le générateur de l'exponentielle est modifié pour supporter les différents scénarios de l'étage de sortie ainsi que l'utilisation d'une ROM compacte.

Listing I.8 Générateur d'exponentielle type RBB

```
%-------------------------------------------------------------
% FUNCTION : VALUES = EXP_GEN(ECHANTILLON, SCENARIO)
% Fonction qui genere une exponentielle suivant
% la methode RBB
%
% NOM DE FICHIER: EXP_GEN.M
%
% INPUT:
% ECHANTILLON  : Taille du vecteur d'echantilons aleatoires
% SCENARIO       : Longueur de l'etage de sortie
%                  1 : difference de 1
%                  2 : difference de 2
%              Autre : difference de croissante
% OUTPUT:
% VALUES       : Vecteurs des echantillons generes
%
%-------------------------------------------------------------
function values = exp_gen(echantillon, scenario)

% appel a la fonction mtx() de l'exponentielle (ROM)
m = mtx();

% taille fonction du scenario
if( scenario == 1)
    lgth = 32;
elseif( scenario == 2)
    lgth = 32*2 - 1;
else
    lgth = 32*31/2 + 1;
end

% pipe (etage de sortie)
nombres = uint32(zeros(lgth,1));

% on remplit le pipe
for i=1:lgth
    nombres(i,1) = m( index( uint32(2^32*rand())) );
end

% initialisation
values = uint32(zeros(echantillon,1));

% generation
for i = 1:echantillon

    % lecture des bits pour la sortie
  values(i,1) = gennombre(nombres, scenario);

    % decaler les chiffres (shift register)
    for j = lgth:-1:2
        nombres(j,1) = nombres(j-1,1);
    end;
    nombres(1,1) = m( index( uint32(2^32*rand()) ) );

end

% de uint vers double (virgule a la position 27)
values = double(values)/2^27;
```

Le générateur d'exponentielle fait appel à sa propre fonction *mtx*() :

Listing I.9 ROM du générateur d'exponentielle

```
%-------------------------------------------------------------
% FUNCTION : M = MTX()
% Fonction qui cree et retourne la ROM 32 x 32 du
% generateur d'exponentielle
%
% NOM DE FICHIER: MTX.M
%
% OUTPUT:
% M : ROM de l'exponentielle
%
%-------------------------------------------------------------
function m = mtx()

% CPT
ml = uint32(zeros(32,1));
% transposee de la CPT bit a bit
m  = uint32(zeros(32,1));

% on genere la CPT
for indice = 1:32
   % lire les prob
   p =1-((exp(-2.^(5-indice))-1)./(exp(-2.^(6-indice))-1));

   % memorise les parametres
   ml(32-indice+1,1) = uint32(2^32*p);
end

for i=1:32 % Chaque nouveau nombre de m
    for j=1:32% est constitue des bit ml
       % memorise les parametres
       m(i,1) = bitset( m(i,1), j, bitget(ml(j,1),i) );
    end
end
```

Le générateur d'exponentielle fait appel à sa propre fonction *gennombre*() qui émule le comportement de l'étage de sortie :

Listing I.10 Étage de sortie du générateur d'exponentielle

```
%-------------------------------------------------------------
% FUNCTION : NOMBRES = GENNOMBRE(NOMBRES, SCENARIO)
% Fonction emulant l'etage de sortie
%
% NOM DE FICHIER: GENNOMBRE.M
%
% INPUT:
% NOMBRES: Pipe des nombres d'echantillons a generer
% SCENARIO       : Longueur de l'etage de sortie
%                  1 : difference de 1
%                  2 : difference de 2
%              Autre : difference de croissante
%
% OUTPUT:
% NOMBRES : Pipe modifie.
%
%-------------------------------------------------------------

function uvalue = gennombre(nombres, scenario);

% initialise
uvalue = uint32( 0 );

i=1;j=1; %j parcourt les bits de uvalue, i dans le pipe

while( j<=32)

    % lit les bons bits dans le pipe
    uvalue = bitset(uvalue, 32-j +1, bitget(nombres(i,1),32-j +1));

    % position des bits fonction du scenario
    if( scenario == 1)
```

```
        i = i+1;
    elseif( scenario == 2)
        i = i+2;
    else
37      i = i+(32-j);
    end

    j = j+1;
end
```

I.4 Générateur de la distribution normale

Le générateur de la normale est modifié pour supporter les différents scénarios de linéarisation.

Listing I.11 Générateur de normale type RBB

```
% ------------------------------------------------------------
% FUNCTION : VALUES = NORM_GEN(INDICE, NB_ECHANTILLONS)
% Fonction qui simule un RBB a 16 noeuds de la normale
4 %  et genere NB_ECHANTILLONS echantillons qui en
% sont issus
%
% NOM DE FICHIER: NORM_GEN.M
%
9 % INPUT:
% INDICE : Premier noeud linearise
% NB_ECHANTILLONS: Nombre d'echantillons a generer
%
% OUTPUT:
14 % VALUES : Vecteur comportant les NB_ECHANTILLONS
%          echantillons aleatoires gEneres.
%
% ------------------------------------------------------------

19 function values = norm_gen(indice, echantillon)

m = mtx(indice);
niveaux = 15; %16e noeud symetrique
nombres = uint16(zeros(niveaux+1,1));
24 idx = zeros(niveaux+1,1);

for i=1:length(idx)
    idx(i) = index( uint32(2^32*rand())) ;
end
29
echantillons = echantillon + niveaux + 1;
values = zeros(echantillon,1);
for i = 1:echantillons

34      % generer tous les niveaux
        nombres = gennombre(m, nombres, niveaux, idx);

        %memoriser si pipe plein
        if( i > niveaux+1)
39
            u = double(nombres(niveaux+1));

            % rÈpertorier les nombres (pos/neg)
            if( idx(niveaux+1) == 32 )
44              u = -u -1;
            end

            u = u/2^(niveaux-2);

49          %et hop
            values(i-niveaux-1) = u;
        end

        %declaer les chiffres
```

```
54      for j = niveaux+1:-1:2
            nombres(j) = nombres(j-1);
        end;
        nombres(1) = uint16(0);
        nombres;
59
        %declaer les chiffres
        for j = 1:niveaux
            idx(j) = idx(j+1);
        end;
64      idx(niveaux + 1) = index( uint32(2^32*rand()) );

end;
```

Le générateur de la normale fait appel à sa propre fonction *mtx*() :

Listing I.12 Matrice du générateur de normale

```
%-------------------------------------------------------------
% FUNCTION : M = MTX(LIMIT)
3 % Fonction qui cree et retourne l'ensemble des
% fonctions nodales (phi function) d'un RBB a 16 noeuds
% generant une normale par symetrie
%
% NOM DE FICHIER: MTX.M
8 %
% INPUT:
% LIMIT: INDICE DU PREMIER NOEUD LINEARISE (-13 a 1)
%
% OUTPUT:
13 % M : Matrice des phi functions (CPT) d'un RBB de 16
%      noeuds.
%-------------------------------------------------------------
function m = mtx(limit)

18 m = zeros(16384,15);

for indice = -13:1

   p = params(indice);
23
   if( indice <limit)
       v   = [0:length(p)-1];
       app = polyfit(v,p,1);
       p   = app(1)*v + app(2);
28 end;

   % memorise les parametres
   for j=1:length(p)
       m(j,2-indice) = p(j);
33 end
end
```

Le générateur de la normale fait appel à sa propre fonction *mtx*() :

Listing I.13 Fonctions nodales du générateur de normale

```
1 %-------------------------------------------------------------
% FUNCTION : P = PARAMS(INDICE)
% Fonction qui cree retourne la fonctions nodales
% d'un RBB normale a 16 noeuds
%
6 % NOM DE FICHIER: PARAMS.M
%
% INPUT:
% INDICE : INDICE NORMALISE DU RBB (-13 a 1)
%
11 % OUTPUT:
% P : phi function (CPT) du noeud indice
%
%-------------------------------------------------------------

16 function p = params(indice)
```

```
% nombre de points
Maxn = 2^(1-indice);
% pas
frac = 1/2^(-indice);

% points
n=[0:Maxn-1];

% probabilites
p = (normcdf((2*n+2)*frac)-normcdf((2*n+1)*frac)) ./ ...
    (normcdf((2*n+2)*frac)-normcdf((2*n)*frac));
```

I.5 Monte Carlo

L'algorithme Monte Carlo étudié a été traduit en Matlab pour des raisons de facilité de manipulation :

Listing I.14 Algorithme Monte Carlo

```
%-----------------------------------------------------------
% FUNCTION : [radius,heat] = tiny_mc(photons)
% Equivalent MATAB du code C disponible a
% http://omlc.ogi.edu
%
% NOM DE FICHIER: TINY_MC.M
%
% INPUT:
% PHOTONS : Nombre de photons
%
% OUTPUT:
% radius : rayon de diffusion
% heat : chaleur absorbee
%
%-----------------------------------------------------------

function [radius,heat] = tiny_mc(photons)

t1 = 'Tiny Monte Carlo by Scott Prahl (http://omlc.ogi.edu)';
t2 = '1 W Point Source Heating in Infinite Isotropic Scattering Medium';

SHELL_MAX = 101;

mu_a = 2.0;  %          /* Absorption Coefficient in 1/cm !!non-zero!! */
mu_s = 20.0; %          /* Reduced Scattering Coefficient in 1/cm */
microns_per_shell = 50; %  /* Thickness of spherical shells in microns */
% photons = 12;

heat = zeros(SHELL_MAX,1);
radius = zeros(SHELL_MAX,1);

albedo = mu_s / (mu_s + mu_a);
shells_per_mfp = 1e4/microns_per_shell/(mu_a+mu_s);

for i = 1:photons
    %fprintf('photon no %d\n', i);
    x = 0.0; y = 0.0; z = 0.0;
  u = 0.0; v = 0.0; w = 1.0;
  weight = 1.0;

    while( true )

        t = -log( rand() ); % move
    x = x + t * u;
    y = y + t * v;
    z = z + t * w;

        shell= fix( sqrt(x*x+y*y+z*z)*shells_per_mfp ); % absorb

    if ( shell > (SHELL_MAX-1) )
```

```
            shell = SHELL_MAX-1;
        end

    heat( shell + 1, 1 ) = heat( shell + 1, 1 ) + (1.0-albedo)*weight;


        weight = weight * albedo;

    while( true )

        % new direction
      xi1=2.0*rand() - 1.0;
      xi2=2.0*rand() - 1.0;

            t=xi1*xi1+xi2*xi2;

            if ( t <=1)
                break;
            end

        end

        u = 2.0 * t - 1.0;
    v = xi1 * sqrt((1-u*u)/t);
    w = xi2 * sqrt((1-u*u)/t);

    if (weight < 0.001)

            % roulette
      if ( rand() > 0.1 )
                break;
            end

            weight = weight/0.1;

        end

    end

    %fprintf('nb de shell %d\n', j);

end

fprintf('%s\n%s\n\nScattering = %8.3f/cm\nAbsorption = %8.3f/cm\n',t1,t2,mu_s,mu_a);
fprintf('Photons     = %8ld\n\n Radius          Heat\n[microns]      [W/cm^3]\n',photons);
t = 4*3.14159*(microns_per_shell^3)*photons/1e12;

for i=1:SHELL_MAX-1
    fprintf('%6.0f      %12.5f\n',i*microns_per_shell, heat( i )/t/(i*i+i+1.0/3.0));
  heat( i ) = heat( i )/t/(i*i+i+1.0/3.0); % retourne la valeur affiche
  radius( i ) = i*microns_per_shell;
end

fprintf(' extra      %12.5f\n',heat( SHELL_MAX )/photons);

heat = heat(1:SHELL_MAX-1);
```

Le vecteur extrait du FPGA est normalisé pour des fins de comparaison :

Listing I.15 Normalisation du résultat du FPGA

```
%-------------------------------------------------------------
% FUNCTION : [x, ncourbe] = my_tiny_mc( heat )
% Fraction de tiny_mc.m
% permettant de normaliser les valeurs extraites du FPGA
%
% NOM DE FICHIER: MY_TINY_MC.M
%
% INPUT:
% HEAT : Chaleur totale (non normalisee)
%
% OUTPUT:
% x : rayon de diffusion
% ncourbe : chaleur absorbee
%
%-------------------------------------------------------------

function [x, ncourbe] = my_tiny_mc( heat );

SHELL_MAX = 101;
```

```
mu_a = 2.0;  %          /* Absorption Coefficient in 1/cm !!non-zero!! */
mu_s = 20.0; %          /* Reduced Scattering Coefficient in 1/cm */
microns_per_shell = 50; % /* Thickness of spherical shells in microns */
photons = 10000;

albedo = mu_s / (mu_s + mu_a);
shells_per_mfp = 1e4/microns_per_shell/(mu_a+mu_s);

t = 4*3.14159*(microns_per_shell^3)*photons/1e12;

for i=1:SHELL_MAX-1
    x(i) = i*microns_per_shell;
    ncourbe(i) = heat(i)/t/(i*i+i+1.0/3.0);
end
```

ANNEXE II

SOURCES VHDL

II.1 Sources génériques

Le générateur Tausworthe combiné :

Listing II.1 Générateur Tausworthe

```
1  --=============================================================================--
   -- Fichier       : tausworthe.vhd
   -- Auteur        : Tarek Ould Bachir
   -- Description   : Generateur aleatoire 32 bits
   -- Modification  : 24 février 2007
6  --=============================================================================--

   LIBRARY ieee;
   use ieee.std_logic_1164.all;
   USE ieee.std_logic_arith.all;
11 use ieee.std_logic_unsigned.all;

   entity tausworthe is
     port (
     clk_port     : in std_logic;
16   ce_port      : in  std_logic;
     en_port      : in  std_logic;
     rst            : in  std_logic;
     seed_1           : in std_logic_vector(31 downto 0);
     seed_2           : in std_logic_vector(31 downto 0);
21   seed_3           : in std_logic_vector(31 downto 0);
     uniform          : out std_logic_vector(31 downto 0)
     );
   end tausworthe;

26 architecture rtl of tausworthe is

   signal x_1     : std_logic_vector(31 downto 0);
   signal x_2     : std_logic_vector(31 downto 0);
   signal x_3     : std_logic_vector(31 downto 0);
31 signal f_x_1     : std_logic_vector(31 downto 0);
   signal f_x_2     : std_logic_vector(31 downto 0);
   signal f_x_3     : std_logic_vector(31 downto 0);

   signal etp_x_1     : std_logic_vector(31 downto 0);
36 signal etp_x_2     : std_logic_vector(31 downto 0);
   signal etp_x_3     : std_logic_vector(31 downto 0);

   signal et_x_1     : std_logic_vector(31 downto 0);
   signal et_x_2     : std_logic_vector(31 downto 0);
41 signal et_x_3     : std_logic_vector(31 downto 0);

   signal xor_x_1     : std_logic_vector(31 downto 0);
   signal xor_x_2     : std_logic_vector(31 downto 0);
   signal xor_x_3     : std_logic_vector(31 downto 0);
46
   signal xxor_x_1     : std_logic_vector(31 downto 0);
   signal xxor_x_2     : std_logic_vector(31 downto 0);
   signal xxor_x_3     : std_logic_vector(31 downto 0);

51 signal c_1  : std_logic_vector(31 downto 0);
   signal c_2  : std_logic_vector(31 downto 0);
   signal c_3  : std_logic_vector(31 downto 0);

   signal local_enable : std_logic;
56
   begin
```

```
  process ( clk_port, rst, seed_1, seed_2, seed_3 )
  begin
    if( rst = '1' ) then
      x_1 <= seed_1;
      x_2 <= seed_2;
      x_3 <= seed_3;
    else
      if (clk_port'event and clk_port = '1') then
        if ( local_enable = '1') then
          x_1 <= f_x_1;
          x_2 <= f_x_2;
          x_3 <= f_x_3;

          uniform <= f_x_1 xor f_x_2 xor f_x_3;

        end if;
      end if;
    end if;
  end process;

  etp_x_1  <= c_1 and x_1;
  et_x_1   <= etp_x_1(31-12 downto 0) & x"000";
  xor_x_1  <= (x_1(31-13 downto 0)& x"000" & '0' ) xor x_1;
  xxor_x_1 <= "000" & x"0000" & xor_x_1(31 downto 19);
  f_x_1    <= (et_x_1) xor (xxor_x_1);

  etp_x_2  <= c_2 and x_2;
  et_x_2   <= etp_x_2(31-4 downto 0) & "0000";
  xor_x_2  <= (x_2(31-2 downto 0)& "00" ) xor x_2;
  xxor_x_2 <= '0' & x"000000" & xor_x_2(31 downto 25);
  f_x_2    <= (et_x_2) xor (xxor_x_2);

  etp_x_3  <= c_3 and x_3;
  et_x_3   <= etp_x_3(31-17 downto 0) & x"0000" & '0';
  xor_x_3  <= (x_3(31-3 downto 0)& "000" ) xor x_2;
  xxor_x_3 <= "000" & x"00" & xor_x_3(31 downto 11);
  f_x_3    <= (et_x_3) xor (xxor_x_3);

  c_1 <= x"FFFFFFFE";
  c_2 <= x"FFFFFFF8";
  c_3 <= x"FFFFFFF2";

  -- Local enable for XSG purposes
  local_enable <= ce_port and en_port;

end rtl;
```

Encodeur de priorité itératif :

Listing II.2 Encodeur de priorité itératif 32 a 5

```
--================================================================================--
-- Fichier      : encprio32to5exp.vhd
-- Auteur       : Tarek Ould Bachir
-- Description  : Encodeur de priorite synchrone
--                32 à 5
-- Modification : 09 novembre 2007
--================================================================================--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity encprio32to5exp is
  port (
    clk : in  std_logic;
    en  : in  std_logic;
    encinput: in  std_logic_vector(31 downto 0);
    outp: out std_logic_vector(4 downto 0) ;
    flag: out std_logic
   );
end encprio32to5exp;

architecture encprio32to5_synch of encprio32to5exp is

component encprio16to4exp
  port (
  encinput: in  std_logic_vector(15 downto 0);
```

```
  outp: out std_logic_vector( 3 downto 0) ;
  flag: out std_logic
   );
end component;

signal split_input_signal_1 : std_logic_vector(15 downto 0);
signal split_input_signal_0 : std_logic_vector(15 downto 0);
signal outp_from_enc2to1_1 : std_logic_vector(3 downto 0);
signal outp_from_enc2to1_0 : std_logic_vector(3 downto 0);
signal outp_mux : std_logic_vector(3 downto 0);
signal flag_from_enc2to1_1 : std_logic;
signal flag_from_enc2to1_0 : std_logic;

begin

  enc16t4_1 : encprio16to4exp
  port map(
    encinput => split_input_signal_1 ,
    outp => outp_from_enc2to1_1 ,
    flag => flag_from_enc2to1_1
   );

  enc16t4_0 : encprio16to4exp
  port map(
    encinput => split_input_signal_0 ,
    outp => outp_from_enc2to1_0 ,
    flag => flag_from_enc2to1_0
   );

  process (clk)
  begin
    if (clk'event and clk = '1') then
      if( en = '1' ) then
        outp <= flag_from_enc2to1_1 & outp_mux;
        flag <= flag_from_enc2to1_1 or flag_from_enc2to1_0;
      end if;
    end if;
  end process;

  process(flag_from_enc2to1_1 , outp_from_enc2to1_1 , outp_from_enc2to1_0)
  begin
  if( flag_from_enc2to1_1 = '1' ) then
    outp_mux <= outp_from_enc2to1_1;
  else
    outp_mux <= outp_from_enc2to1_0;
  end if;
  end process;

  split_input_signal_1 <= encinput(31 downto 16);
  split_input_signal_0 <= encinput(15 downto 0);

end encprio32to5_synch;
```

Listing II.3 Encodeur de priorité 16 a 4

```
--================================================================================--
-- Fichier       : encprio16to4exp.vhd
-- Auteur        : Tarek Ould Bachir
-- Description   : Encodeur de priorite
--                 16 a 4
-- Modification : 09 novembre 2007
--================================================================================--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity encprio16to4exp is
  port (
    encinput: in  std_logic_vector(15 downto 0);
    outp: out std_logic_vector(3 downto 0) ;
    flag: out std_logic
   );
end encprio16to4exp;

architecture archi of encprio16to4exp is
  signal signal_outp : std_logic_vector(3 downto 0);
  signal signal_flag : std_logic;
  begin

    signal_outp <= "1111" when encinput( 15 ) = '1' else
        "1110" when encinput( 14 ) = '1' else
        "1101" when encinput( 13 ) = '1' else
```

```
        "1100" when encinput( 12 ) = '1' else
        "1011" when encinput( 11 ) = '1' else
        "1010" when encinput( 10 ) = '1' else
        "1001" when encinput(  9 ) = '1' else
        "1000" when encinput(  8 ) = '1' else
        "0111" when encinput(  7 ) = '1' else
        "0110" when encinput(  6 ) = '1' else
        "0101" when encinput(  5 ) = '1' else
        "0100" when encinput(  4 ) = '1' else
        "0011" when encinput(  3 ) = '1' else
        "0010" when encinput(  2 ) = '1' else
        "0001" when encinput(  1 ) = '1' else
        "0000" when encinput(  0 ) = '1' else
        "0000";


    signal_flag <= '0' when encinput = "00000000000000000000000000000000" else
      '1';

  outp <= signal_outp;
  flag <= signal_flag;

end archi;
```

Listing II.4 Registre à décalage

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

entity shift_register is
generic (
    n           : integer);     -- longueur
port (
    clk     : in  std_logic;   -- horloge
    ce      : in std_logic;    -- clock enable (c.e.)
    din     : in  std_logic; -- vecteur d'entrÈe
    dout    : out std_logic); -- vecteur de sortie
end shift_register;

architecture rtl of shift_register is
    type T_SHIFT_REGISTER is array (0 to n-1) of std_logic;
    signal pregs : T_SHIFT_REGISTER;
    --signal regs : std_logic_vector(n-1 downto 0);

begin
    -- Processus sequentiel ou les donnees sont shiftee
    shift: process (clk)
    begin
        -- Fonctionnement avec controe de l'horloge par "ce"
        -- N.B. Si "ce" vaut '0', les donnees ne vont pas se decaler
        if (clk'event and clk = '1' and ce = '1') then  -- rising clock edge
    for i in n-2 downto 0 loop
       pregs(i+1) <= pregs(i);
    end loop;

            pregs(0) <= din;
        end if;
    end process shift;

    -- Assignation par deÈfaut
    dout <= pregs(n-1);
end rtl;
```

II.2 Sources du générateur de la distribution exponentielle

Le générateur à ROM compacte :

Listing II.5 Générateur d'exponentielle à ROM compacte

```
--================================================================================--
-- Fichier       : exp_generator.vhd
-- Auteur        : Tarek Ould Bachir
-- Description   : GÈnÈrateur d'Èchantillons issus de
--                 la distribution exp. de mu=1
-- Modification : 09 novembre 2007
--================================================================================--
--

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

entity exp_generator is
    port (
  clk_port    : in std_logic;
  ce_port     : in  std_logic;
  en_port     : in  std_logic;
  rst         : in std_logic;
  port_output : out std_logic_vector (31 downto 0)
  );
end exp_generator;


architecture rtl of exp_generator is

    component tausworthe
  port (
  clk_port    : in std_logic;
  ce_port     : in  std_logic;
  en_port     : in  std_logic;
  rst            : in  std_logic;
  seed_1           : in std_logic_vector(31 downto 0);
  seed_2           : in std_logic_vector(31 downto 0);
  seed_3           : in std_logic_vector(31 downto 0);
  uniform          : out std_logic_vector(31 downto 0)
  );
  end component;

  component encprio32to5exp
  port (
    clk : in  std_logic;
    en  : in  std_logic;
    encinput: in  std_logic_vector(31 downto 0);
    outp: out std_logic_vector(4 downto 0) ;
    flag: out std_logic
   );
  end component;

  component diagexp
  port (
    clk : in  std_logic;
    en  : in  std_logic;
    port_input  : in  std_logic_vector(31 downto 0);
    port_output : out std_logic_vector(31 downto 0)
   );
  end component;

  component romexp
  port (
    clk : in  std_logic;
    en  : in  std_logic;
    addr: in  std_logic_vector(4 downto 0);
    data: out std_logic_vector(31 downto 0)
   );
  end component;

    signal local_enable : std_logic;
    signal FillSel      : std_logic;
    signal flag_inutile     : std_logic;
    signal program_lfsr     : std_logic_vector (7 downto 0);
    signal echantillon_uniform  : std_logic_vector (31 downto 0);
    signal uniform_encode     : std_logic_vector (4 downto 0);
    signal donnees_rom      : std_logic_vector (31 downto 0);

    signal seed1 : std_logic_vector (31 downto 0);
    signal seed2 : std_logic_vector (31 downto 0);
    signal seed3 : std_logic_vector (31 downto 0);

begin  -- rtl



  encodeur : encprio32to5exp
```

```
    port map(
      clk => clk_port,
      en  => local_enable,
      encinput => echantillon_uniform,
      outp => uniform_encode,
      flag => flag_inutile
     );

    rom32x32: romexp
    port map(
      clk => clk_port,
      en  => local_enable,
      addr => uniform_encode,
      data => donnees_rom
     );

    reg_decal : diagexp
    port map(
      clk => clk_port,
      en  => local_enable,
      port_input  => donnees_rom,
      port_output => port_output
     );

    twth: tausworthe
    port map(
    clk_port  => clk_port,
    ce_port   => ce_port,
    en_port   => en_port,
    rst       => rst,
    seed_1    => seed1,
    seed_2    => seed2,
    seed_3    => seed3,
    uniform   => echantillon_uniform
    );

    local_enable <= ce_port and en_port;
    seed1 <= x"33333333";
    seed2 <= x"55555555";
    seed3 <= x"99999999";

end rtl;
```

Listing II.6 ROM de l'exponentielle

```
--================================================================================--
-- Fichier       : romexp.vhd
-- Auteur        : Tarek Ould Bachir
-- Description   : ROM synchrone de la distribution exponentielle
-- Modification : 09 novembre 2007
--================================================================================--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity romexp is
  port (
    clk : in  std_logic;
    en  : in  std_logic;
    addr: in  std_logic_vector(4 downto 0);
    data: out std_logic_vector(31 downto 0)
   );
end romexp;

architecture rom_synch of romexp is

type rom_type is array(0 to 31) of std_logic_vector(31 downto 0);

constant exp_table : rom_type := (
    "11111100001010000101010111110000",
    "10110011010100000101010111110000",
    "01100000101010000101010111110000",
    "01010111110100000101010111110000",
    "01001101001000000101010111110011",
    "11010010010100000101010111110111",
    "10001110101000000101010111111111",
    "10100001010000000101010111101111",
    "10010111101000000101010111011111",
    "01100010010000000101010110111111",
    "00110000100000000101010101111111",
    "01101001010000000101010011111111",
    "01111010100000000101011111111111",
```

```
    "01100011000000000101001111111111",
    "01001000100000000101111111111111",
    "01100101000000000100111111111111",
    "01001010000000000111111111111111",
    "00100101000000000011111111111111",
    "01010110000000001111111111111111",
    "00101000000000011111111111111111",
    "01101010000000111111111111111111",
    "00000100000001111111111111111111",
    "00001000000011111111111111111111",
    "00111100000111111111111111111111",
    "00000000001111111111111111111111",
    "00010000011111111111111111111111",
    "00111000111111111111111111111111",
    "00010001111111111111111111111111",
    "00010011111111111111111111111111",
    "00000111111111111111111111111111",
    "00001111111111111111111111111111",
    "00000000000000000000000000000000"
);

begin
  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (en = '1') then
        data <= exp_table(conv_integer(addr));
      end if;
    end if;
  end process;

end rom_synch;
```

Listing II.7 Étage de sortie

```
--==============================================================================--
-- Fichier      : diagexp.vhd
-- Auteur       : Tarek Ould Bachir
-- Description  : Registre a decalage a longueur decroissante
-- Modification : 09 novembre 2007
--==============================================================================--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity diagexp is
  port (
    clk : in  std_logic;
    en  : in  std_logic;
    port_input  : in  std_logic_vector(31 downto 0);
    port_output : out std_logic_vector(31 downto 0)
   );
end diagexp;

architecture diagexp_synch of diagexp is

component shift_register
generic (
    n          : integer := 1);
port (
    clk     : in  std_logic;   -- horloge
    ce      : in std_logic;    -- clock enable (c.e.)
    din     : in  std_logic; -- vecteur d'entrÈe
    dout    : out std_logic); -- vecteur de sortie
end component;

signal signal_input : std_logic_vector(31 downto 0);

begin

  G: for I in 0 to 31 generate
  srl0:shift_register
  generic map(
    n          => (I+1)*(I+2)/2
    )
  port map (
    clk     => clk,
    ce      => en,
    din     => signal_input( I ),
    dout    => port_output( I )
    );
  end generate G;
```

```
49
  signal_input <= port_input;

end diagexp_synch;
```

Le générateur d'exponentielle utilisant la cellule de van Daalen :

Listing II.8 Générateur utilisant la cellule de van Daalen (Shawe-Taylor)

```
--================================================================================--
2 -- Fichier       : exp_generator.vhd
-- Auteur        : Tarek Ould Bachir
-- Description   : Cellule d'un generateur 1 bit
-- Modification  : 24 fÈvrier 2007
--================================================================================--
7
LIBRARY ieee;
use ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
12
entity exp_generator is
  port (
  clk_port     : in std_logic;
  ce_port      : in  std_logic;
17 en_port    : in  std_logic;
  rst          : in std_logic;
  port_output : out std_logic_vector (31 downto 0)
  );
end exp_generator;
22
architecture rtl of exp_generator is

    signal local_enable : std_logic;
    signal clk : std_logic;
27  signal echantillon_uniform : std_logic_vector (31 downto 0);
    signal echantillon_exp : std_logic_vector (31 downto 0);
    signal seed1 : std_logic_vector (31 downto 0);
    signal seed2 : std_logic_vector (31 downto 0);
    signal seed3 : std_logic_vector (31 downto 0);
32



  component exp_shawe_generator
  port (
37 clk              : in  std_logic;
  en           : in  std_logic;
  alea_nbr          : in  std_logic_vector(31 downto 0);
  port_exp          : out std_logic_vector(31 downto 0)
  );
42 end component;

    component tausworthe
  port (
  clk_port     : in std_logic;
47 ce_port    : in  std_logic;
  en_port      : in  std_logic;
  rst             : in  std_logic;
  seed_1            : in std_logic_vector(31 downto 0);
  seed_2            : in std_logic_vector(31 downto 0);
52 seed_3            : in std_logic_vector(31 downto 0);
  uniform           : out std_logic_vector(31 downto 0)
  );
  end component;

57

begin

  expst: exp_shawe_generator
  port map(
62 clk               => clk_port,
  en                => local_enable,
  alea_nbr          => echantillon_uniform,
  port_exp          => port_output
  );
67

  twth: tausworthe
  port map(
  clk_port  => clk_port,
  ce_port   => ce_port,
```

```
72    en_port   => en_port,
      rst       => rst,
      seed_1    => seed1,
      seed_2    => seed2,
      seed_3    => seed3,
77    uniform   => echantillon_uniform
      );


      local_enable <= ce_port and en_port;
82    seed1 <= x"33333333";
      seed2 <= x"55555555";
      seed3 <= x"99999999";

end rtl;
```

Listing II.9 Ensemble de 32 générateurs de van Daalen

```
--================================================================================--
-- Fichier       : exp_shawe_generator.vhd
-- Auteur        : Tarek Ould Bachir
4  -- Description   : Cellule d'un generateur 1 bit
-- Modification : 24 fÈvrier 2007
--================================================================================--

LIBRARY ieee;
9  use ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity exp_shawe_generator is
14    port (
      clk               : in  std_logic;
      en                : in  std_logic;
      alea_nbr          : in  std_logic_vector(31 downto 0);
      port_exp          : out std_logic_vector(31 downto 0)
19    );
end exp_shawe_generator;

architecture rtl of exp_shawe_generator is

24

  component shawe_generator
  generic(
  N         : integer := 32
  );
29    port (
      clk               : in  std_logic;
      en                : in  std_logic;
      input_bit         : in  std_logic;
      mod_bits          : in  std_logic_vector(N-1 downto 0);
34    output_port       : out  std_logic
      );
    end component;

type lesn is array(0 to 31) of integer;
39  constant ENs : lesn := (
      32,32,32,32,32,32,32,32
      32,32,32,32,32,32,32,32
      32,32,32,32,32,32,32,32
      32,32,32,32,32,32,32,32);
44
type rom_type is array(0 to 31) of std_logic_vector(31 downto 0);
signal const_exp : rom_type := (
        "01111111111111111111111111110000",
        "01111111111111111111111111110000",
49      "01111111111111111111111111100000",
        "01111111111111111111111111000000",
        "01111111111111111111111110000000",
        "01111111111111111111111100000000",
        "01111111111111111111111000000000",
54      "01111111111111111111110000000000",
        "01111111111111111111100000000000",
        "01111111111111111111000000000000",
        "01111111111111111110000000000000",
        "01111111111111111100000000000000",
59      "01111111111111111000000000000000",
        "01111111111111110000000000000000",
        "01111111111111100000000000000000",
        "01111111111111000000000000000000",
        "01111111111110000000000000000000",
64      "01111111111100000000000000000000",
        "01111111111000000000000000000001",
```

```
        "01111111110000000000000000000101",
        "01111111100000000000000000010101",
        "01111111000000000000000010101011",
        "01111110000000000000010101010101",
        "01111100000000000010101010011101",
        "01111000000000101010100110011010",
        "01110000000101010011001101101010",
        "01100000101001101000000101011001",
        "01000100110110010101100001010001",  -- 31
        "00011110100001000001010100101100",  -- 29
        "00000100100110101011111010001000",  -- 27
        "00000000000101010111110100011110",  -- 21
        "00000000000000000000000111100011"   --  9
  );

begin

  gen_shawegens: for I in 0 to 31 generate
  shawe_gen_a: shawe_generator
  generic map(
  N => ENs( I )
  )
  port map(
  clk => clk,
  en  => en,
  input_bit   => alea_nbr( I ),
  mod_bits    => const_exp( I )(31 downto 31-(ENs( I )-1)),
  output_port => port_exp( I )
  );
  end generate gen_shawegens;

end rtl;
```

Listing II.10 Generateur de van Daalen

```
--==============================================================================--
-- Fichier       : shawe_generator.vhd
-- Auteur        : Tarek Ould Bachir
-- Description   : Cellule d'un generateur 1 bit
-- Modification : 24 fÈvrier 2007
--==============================================================================--

LIBRARY ieee;
use ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity shawe_generator is
  generic(
  N         : integer := 32
  );
  port (
  clk               : in  std_logic;
  en                : in  std_logic;
  input_bit         : in  std_logic;
  mod_bits          : in  std_logic_vector(N-1 downto 0);
  output_port       : out  std_logic
  );
end shawe_generator;

architecture rtl of shawe_generator is


  component cellule_shawe
  port (
  clk               : in  std_logic;
  en                : in  std_logic;
  input_bit         : in  std_logic;
  output_bit        : out std_logic;
  mod_bit           : in  std_logic;
  previous_stage    : in  std_logic;
  next_stage        : out  std_logic
  );
  end component;

signal prob_bits : std_logic_vector(N downto 0);
signal bernoulli : std_logic_vector(N downto 0);

begin

  gen_cells: for I in N-1 downto 0 generate

    cell_shawe: cellule_shawe
```

```
    port map(
    clk             => clk,
    en              => en,
    input_bit       => prob_bits( I+1 ),
    mod_bit         => mod_bits ( I ),
    output_bit      => prob_bits( I ),
    previous_stage  => bernoulli( I ),
    next_stage      => bernoulli( I+1 )
    );

  end generate gen_cells;

  prob_bits( N ) <= input_bit;
  bernoulli( 0 ) <= '0';
  output_port <= bernoulli( N );

end rtl;
```

Listing II.11 Cellule du générateur de van Daalen

```
--==============================================================================--
-- Fichier      : cellule_shawe.vhd
-- Auteur       : Tarek Ould Bachir
-- Description  : Cellule d'un generateur 1 bit
-- Modification : 24 fÈvrier 2007
--==============================================================================--

LIBRARY ieee;
use ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity cellule_shawe is
  port (
  clk             : in  std_logic;
  en              : in  std_logic;
  input_bit       : in  std_logic;
  mod_bit         : in  std_logic;
  output_bit      : out std_logic;
  previous_stage  : in  std_logic;
  next_stage      : out  std_logic
  );
end cellule_shawe;

architecture rtl of cellule_shawe is

signal prob_bit : std_logic;

begin

  prob_bit <= (mod_bit and previous_stage) when input_bit = '0' else
              (mod_bit  or previous_stage);

  process ( clk )
  begin
    if (clk'event and clk = '1') then
      if ( en = '1') then
        next_stage <= prob_bit;
        output_bit <= input_bit;
      end if;
    end if;
  end process;

end rtl;
```

ANNEXE III

SOURCES C

III.1 Monte Carlo

Le code Monte Carlo disponible à l'adresse `www.unifr.ch/physics/mm/work/monte_carlo/tiny_mc.c` (copie commentée du code Monte Carlo disponible à l'adresse du Oregon Medical Laser Center `http://omlc.ogi.edu/software/mc/`).

Listing III.1 Agorithme Monte Carlo

```
/*****************************************************************************/
/* Simplest Monte-Carlo simulation code for light propagation in scattering medium */
/*                Designed by Scott Prahl (http://omlc.ogi.edu)                */
/*                Commented by P. Zakharov                                     */
/* Legend:                                                                     */
/* The IW point (???) source located in the infinite isotropic scattering medium */
/* $Id: tiny_mc.c,v 1.1.1.1 2005/06/01 14:24:34 propan Exp $                   */
/*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define SHELL_MAX  101              /* Number of shells to be used for statistics  */

double mu_a = 2;                    /* Absorption Coefficient in 1/cm !!non-zero!! */
double mu_s = 20;                   /* Reduced Scattering Coefficient in 1/cm      */
double microns_per_shell = 50;      /* Thickness of spherical shells in microns    */
long   photons = 100000;            /* Number of photons to emit */
long   i, shell;
double x, y, z, u, v, w, weight;
double albedo, shells_per_mfp, xi1, xi2, t, heat[SHELL_MAX];

int main ()
{
  /* Calculating constants */
  albedo = mu_s / (mu_s + mu_a);
  shells_per_mfp = 1e4 / microns_per_shell / (mu_a + mu_s);

  /* Main loop */
  for (i = 1; i <= photons; i++)
    {
      x = 0.0; y = 0.0; z = 0.0;  /* Coordinates */
      u = 0.0; v = 0.0; w = 1.0;  /* Velocity components */
      weight = 1.0;

      for (;;) /* Main Photon loop */
  {
    /* displacement vector length in units of scattering length */
    t = -log((rand()+1.0) / (RAND_MAX+1.0));
    x += t * u;
    y += t * v;
    z += t * w;

    /* calculating the shell index */
    shell = sqrt(x*x + y*y + z*z) * shells_per_mfp;

    /* The number of shells is limited - we must check it */
```

```
    if (shell > SHELL_MAX-1)
      shell = SHELL_MAX-1;

    /* Heating certain shell in accordance to lost weight */
    heat[shell] += (1.0 - albedo) * weight;

    /* Decreasing weight of photo packet */
    weight *= albedo;


    /* the fast way to generate unit vector of random direction
       and find to     it's projections (u, v, w) */
    for(;;)
      {

        /* random number from uniform distribution [-1,1] */
        xi1 = 2.0 * rand() / RAND_MAX - 1.0;

        /* one more random number from uniform distribution [-1,1] */
        xi2 = 2.0 * rand() / RAND_MAX - 1.0;

        /* Stopping if it is less or equal to one */
        if ((t = xi1 * xi1 + xi2 * xi2) <= 1)
    break;

      } /* Otherwise continuing to search */

    u = 2.0 * t - 1.0;                          /* calculationg projections */
    v = xi1 * sqrt((1 - u * u) / t);            /* calculationg projections */
    w = xi2 * sqrt((1 - u * u) / t);            /* calculationg projections */

    /* Trying to terminate if the packet weight is too small */
    if (weight < 0.001)
      {

        /* Roulette technique - giving the chance to survive */
        if (rand() > 0.1 * RAND_MAX)
    break;

        /* Survived gets additional energy from the terminated photons
           to obey the energy conservation law */
        weight /= 0.1;

      }

  } /* End of photon loop */
    }

  /* Outputting the parameters */
  printf("Scattering = %8.3f/cm\nAbsorption = %8.3f/cm\n",
   mu_s, mu_a);

  /* Outputting the header for table */
  printf("Photons     = %8ld\n\n Radius          Heat\n[microns]     [W/cm^3]\n",
   photons);

  /* Normalization coefficient */
  t = 4 * 3.14159 * pow(microns_per_shell, 3) * photons / 1e12;

  /* Output fo the heat in certain shell of the inner radius i
     normalized by the shell volume factor */
  for (i = 0; i < SHELL_MAX - 1; i++)
    printf("%6.0f     %12.5f\n",
     i * microns_per_shell,
     heat[i] / t / (i * i + i + 1.0 / 3.0));

  /* These photons are outside of our volume of interest */
  printf(" extra     %12.5f\n", heat[SHELL_MAX-1] / photons);

  return 0;

}
```