Le hasard et sa production

Guilhem Marion

Table des matières

Introduction

De nos jours, l'ordinateur fait partie intégrante de nos vies quotidiennes, il nous permet d'acheter des objets sur internet, de jouer, d'écouter de la musique ... Derrière toutes ces activités que nous propose cet outil numérique se cache algorithmes et autres opérations complexes. Ce stage se propose d'en étudier une d'entre elles : les séquences de nombres aléatoires. Elles sont présentes dans de nombreux aspects de l'informatique et sont indispensables en cryptographie, la science de crypter l'information. Dans ce document nous verrons comment définir la notion d'*aléatoire* aux yeux des mathématiques, de la physique et de la philosophie, et ainsi définir ce qu'est une séquence de nombres aléatoires. Puis nous verrons comment générer ces séquences et en tester la validité. Dans un second temps, nous présenterons quelques experiences à propos de ces séquences : comment créer sois-même un générateur de qualité cryptographique à moindre coût en utilisant le bruit blanc d'une carte son.

1 Considérations historiques et épistémologiques

Notre problématique principale est de mettre en évidence le comment de la génération de nombres aléatoires et du test de son efficacité, et ce par des moyens algorithmiques. Mais derrière ces questions se pose un concept géant de la philosophie : le *hasard*. Avant de définir algorithmiquement ce qu'est une bonne suite de nombres aléatoires et comment la produire il paraît en effet indispensable de savoir ce qu'est l'aléatoire, question loin d'être triviale mettant en avant une fois de plus le lien étroit qu'entretiennent philosophie et mathématiques.

1.1 Considérations historiques : le hasard entre mathématiques, physique et philosophie

Mathématiques

Cette notion fut investie dés les premiers jeux de hasard, mais non pas par les mathématiques mais la philosophie. Les grecs de l'antiquité que nous considérons comme les premiers modernes des mathématiques n'ont pas développé de théorie des probabilités, et ce n'est pas faute de ne pas connaître les jeux de hasards, les problèmes de sondages ou d'assurance [1] [3].

C'est durant la renaissance italienne que les premières considérations mathématiques quant aux probabilités ont émergées. L'algébriste Gérome Cardan publie un traité dans lequel il utilise divers raisonnements d'analyse combinatoire élémentaire afin de determiner des probabilités de gain à un jeux de dés, mais s'il faut attribuer la paternité des probabilités , c'est plutôt au $XVII^{me}$ siècle qu'il convient de se placer, avec le travail de Blaise Pascal : il a résolu, en même temps que Pierre de Fermat mais avec une autre méthode, le problème des partis [2]. Mais l'apport le plus notable de Pascal à cette notion est avant tout épistémologique : le hasard n'est plus une notion incertaine, encombrante, mais peut être abordée par le prisme de la science. Il propose de développer cette nouvelle discipline qu'il appelle la *géométrie du hasard*. Il accepte le hasard comme objet d'investigation, et son objectif n'est pas de le faire disparaitre mais de l'étudier et d'en décrire les *lois* [3]

1. *Mathématiques et philosophie*, Benoit Rittaux, aux édition tangente, p.123
2. *Problème des partis*, Wikipédia
3. *Mathématiques et philosophie*, Benoit Rittaux, aux édition tangente, p.123.

C'est au XX^{me} que les probabilités sont élevées au rang de domaine à part entière avec les travaux de Andreï Kolmogorov qui en axiomatisant les probabilités -en répondant au 6^e problème de Hilbert -, [4]. Il est à noter que durant toute l'histoire de la théorie des probabilités, un débat eut lieu entre les probabilistes *objectivistes* et les probabilistes *subjectivistes*, respectivement ceux qui pensent que les probabilités existent dans la nature et ceux qui pensent qu'elles n'ont pas d'existences naturelles.

C'est à cette époque qu'émmergent les premieres tentatives de definitions mathématiques de la notion de hasard, plus précisément de suite aléatoire. Durant la seconde guerre mondiale, après plusieurs tentatives de formalisation du hasard par Richard von Mises en 1919, l'épistémologue Karl Popper en 1935 et Alonzo Church en 1940, on ne croyait plus en la possibilité d'une définition formelle. Emile Borel a proposé un paradoxe selon lequel la définition du hasard est par nature impossible [5] :

En effet, si on conçoit intuitivement une suite aléatoire comme une suite ne possédant absolument aucune caractéristique particulière, alors le simple fait de définir une suite aléatoire donne une caractéristique aux suites répondant à la définition, qui est le fait d'être « aléatoire ». Donc le simple fait de définir l'aléatoire est paradoxal par rapport à sa définition intuitive.

Philosophie

Pour la philosophie le hasard se définit comme une cause fortuite ou événement absent d'une cause clairement assignable, ce principe s'oppose donc à celui de causalité. Aristote le définit en deux principes : le *tuké* et l'*automaton* appartenant à aucune fin ni aucune volonté. Cependant au $XVII^{me}$ siècle une autre conception intervient : le mécanisme. C'est une conception matérialiste qui conçoit la plupart des phénomènes suivant le modèle de cause à effet. Il correspond à une révolution scientifique appelée révolution copernicienne. Il vise à réduire tous les phénomènes physique à des chocs entre particules, pour Descartes, un tel type d'explication permettrait de comprendre la totalité des phénomènes naturels à partir de principes simples.

A la même époque, Spinoza[17] donne une autre valeur à la notion :

> « Seule notre ignorance de l'enchainement particulier des causes nous fait donner le nom de hasard à ce que nous ne comprenons pas. [6] »

Ces deux idées nous amènent assez naturellement à l'idée bien connue du *déterminisme Laplacien* qui stipule que si un observateur omniscient connait la vitesse et la position de toutes les particules de l'univers à un instant t alors il est capable de prédire la configuration de l'univers à l'instant $t + 1$ et par induction, à tous les instants : c'est l'idée d'un modèle déterministe.

Cette idée est relativement admise au sein de la communauté scientifique -la vision *mécaniste* est même approuvée par la *théorie du chaos* en physique. C'est à l'apparition de la physique quantique que le statut ontologique du hasard vire de bord.

Physique

Alors que la théorie du chaos admet qu'il existe un déterminisme certain dans les phénomènes physiques mais qui nous apparait comme relevant d'une forme de hasard à cause d'une haute sensibilité des *conditions initiales* [7]. C'est notre ignorance de tous les paramètres qui

4. ibid, p.124
5. *Hasard et complexité en mathématiques* [4], G. Chaitin, Flammarion, 2009
6. *L'éthique*, Spinoza, appendice à la première partie, 1677
7. Théorie du chaos, Wikipédia

nous interdit toute prediction et non pas le caractère aléatoire des événements. Cette notion s'accorde parfaitement avec l'idée du déterminisme Laplacien et de la pensée mécaniste. On retrouve cette idée chez Poincaré [15] :

> « Une cause très petite, qui nous échappe, détermine un effet considérable que nous ne pouvons pas ne pas voir, et alors nous disons que cet effet est dû au hasard. Si nous connaissions exactement les lois de la nature et la situation de l'univers à l'instant initial, nous pourrions prédire exactement la situation de ce même univers à un instant ultérieur. Mais, lors même que les lois naturelles n'auraient plus de secret pour nous, nous ne pourrions connaître la situation qu'approximativement. Si cela nous permet de prévoir la situation ultérieure avec la même approximation, c'est tout ce qu'il nous faut, nous disons que le phénomène a été prévu, qu'il est régi par des lois ; mais il n'en est pas toujours ainsi, il peut arriver que de petites différences dans les conditions initiales en engendrent de très grandes dans les phénomènes finaux ; une petite erreur sur les premières produirait une erreur énorme sur les derniers. La prédiction devient impossible et nous avons le phénomène fortuit. [8] »

Cependant l'arrivée de la mécanique quantique nous amène à repenser la notion. Il existe des experiences qui nous montrent que le hasard fait partie entière des phénomènes physiques : c'est le cas de l'experience des fentes de Young avec un projecteur de photon. On bombarde un photon sur une lame semi réfléchissante : le passage ou non du photon relève d'un événement complètement aléatoire.

On met donc en évidence des phénomènes aléatoires dans la nature, de plus la mécanique quantique fait grand cas des probabilités, en effet la notion de *fonction d'onde* fait en fait appel à une probabilité de présence, c'est ainsi qu'est définie l'équation de Schrödinger qui donne les fonctions d'ondes de l'électron autour du noyau. On remarque ainsi que l'électron n'a pas de trajectoire comme la mécanique classique le voudrait mais une probabilité de présence [9]. Mais cela est peut être à une lacune théorique quant aux phénomènes quantiques.

1.2 Etat de l'art de la condition épistémologique du hasard

Nous avons vu qu'à l'heure actuelle la science des phénomènes non-quantiques tend vers une vision très spinoziste du hasard : il est le nom que l'on donne à un événement dont on ne peut connaitre précisément les conditions initiales et/ou les différents paramètres influençant le système. On peut donc percevoir le hasard comme une ignorance quant aux subtiles lois de la nature (même si le sujet fait toujours débat). On fait donc appel aux probabilités pour approcher ces paramètres, et on retrouve ces lois à partir du formalisme mathématiques. Cependant le modèle probabiliste s'accorde empiriquement à un grand nombre de phénomènes mais cette adéquation reste néanmoins impossible à prouver. Cependant, les probabilités nous serons d'une grande aide car elles seules semblent pouvoir nous permettre de définir le comportement d'une séquence aléatoire.

Par extension, une séquence de nombres aléatoires produite "artificiellement" par un ordinateur peut être définie par une totale absence de predictabilité pour l'observateur même si cette séquence est par definition purement déterministe (puisque produite par l'ordinateur). La notion d'observateur devient donc une notion très importante : une séquence de nombres peut être aléatoire ou non selon l'observateur. Si l'on considère l'utilisateur comme l'observateur, une séquence produite à partir de la vitesse de rotation du disque dur ou de quelconque bruit materiel sera complètement aléatoire. Or si l'observateur est un espion cherchant à casser une clef produite à partir de nombres aléatoires la séquence ne le sera plus.

8. *Calcul des probabilités*, Henri Poincaté, éditions Jacques Gabay, 1987
9. Chimie générale, Paul Arnaud, Dunod, 2016

1.3 Modélisation d'une notion hasardeuse

Les premiers fantasmes quant à l'étude du hasard furent de le contrôler, de supprimer cette notion afin de le faire jouer en faveur de son observateur [10] [7] (Fortune et ruine du chevalier de Méré, etc ...). Aujourd'hui l'informatique se retrouve confrontée à un tout nouveau problème : sa modélisation. Depuis les année 1970 de nombreux systèmes informatiques nécessitent la production de séquences aléatoires pour leur bon fonctionnement (Cryptographie, intelligence artificielle, simulation, échantillonnage, etc ...).

Suite aléatoire : une définition formelle difficile

En mathématiques, une suite aléatoire, ou suite infinie aléatoire, est une suite de symboles d'un alphabet ne possédant aucune structure, régularité, ou règle de prédiction identifiable. Une telle suite correspond à la notion intuitive de nombres tirés au hasard. La caractérisation mathématique de cette notion est extrêmement difficile, et a fait l'objet d'études et de débats tout au long du xxe siècle. Une première tentative de définition mathématique (insatisfaisante) a été réalisée en 1919 par Richard von Mises. Ce fut l'avènement de la théorie de la calculabilité, dans les années 1930, et de la théorie algorithmique de l'information qui permit d'aboutir dans les années 1970 à des définitions faisant aujourd'hui consensus.

Les définitions actuellement acceptées (démontrées équivalentes) du caractère aléatoire d'une suite sont les suivantes [11][8] :

1. Une suite aléatoire ne doit posséder aucune régularité « exceptionnelle et effectivement testable » (Martin-Löf 1966) ;
2. Une suite aléatoire doit posséder un « contenu informationnel incompressible » (Levin 1974, Chaitin 1975) ;
3. Une suite aléatoire doit être imprévisible, c'est-à-dire qu'aucune « stratégie effective » ne peut mener à un « gain infini » si l'on « parie » sur les termes de la suite (Schnorr 1971).

Notons qu'une suite de nombres aléatoires sur un alphabet Σ peut s'exprimer par une bijection $\{0,1\}^* \to \Sigma^*$. On exprime donc couramment un séquence aléatoire de n nombres comme un reel de taille $n*k$ exprimé en binaire, avec k la taille binaire d'un digit de l'alphabet Σ.

Intéressons nous à la definition (2) de Levin-Chaitin, elle se base sur la complexité de Kolmogorov notion clef de la théorie de l'information développée dans les années 60 par Ray Solomonoff et Andreï Kolmogorov.

Complexité de Kolmogorov

On pose une machine M pouvant executer des programmes : la machine de Turing par exemple. On note P_M l'ensemble des programmes pouvant être exécutés sur M.

$\forall p \in P_M$, on note $l(p)$ le nombre d'instruction de p sur M et $s(p)$ la taille la sortie de p sur M.

On définie la complexité de Kolmogorov d'une suite x notée $K_M(x)$ par

$$K_M(x) = \min_{p \in P_M} \{l(p) | s(p) = x\}.$$

10. *La probabilité, le hasard et la certitude*, Paul Deheuvels, Que sais-je ? puf, 2008, Chapitre 2
11. *Information, complexité et hasard*, Jean-Paul Delahaye, 1999, Hermes

```
Algo : Kolmo :
        n := 1
        Tant que K(n) < k + 100 :
                n := n + 1
        Fin Tant que
        Renvoyer n
```

FIGURE 1 – Algo Kolmogorov

On définit donc la complexité de Kolmogorov comme la longueur du plus petit programme de M qui engendre la suite x. La théorème d'invariance nous montre qu'à une constante additive prés $K_M(x)$ ne depend pas de la machine M, cela justifie son caractère absolu.

Tout le problème de cette notion est que $K_M(x)$ est indécidable, preuve par l'absurde :

Démonstration. On suppose que la fonction $K(s)$ existe et renvoie le complexité de Kolmogorov de s, soit k la taille de cette fonction, cf fig.1

Ce programme renvoie le plus petit entier n tel que sa complexité de Kolmogorov est supérieure ou égale à $k + 100$, n existe car il y a un nombre fini de programmes de taille inférieure à k+100.

Or, l'algorithme ci-dessus s'écrit en moins de k+100 caractères : il est donc de complexité inférieure à k+100, or il écrit justement un nombre de complexité supérieure à k+100, ce qui est absurde. Une fonction telle que K ne peut donc pas exister.

□

Cependant il faut bien garder en tête que ces définitions sont des definitions mathématiques formelles. En effet, la complexité de Kolmogorov n'étant pas calculable on ne peut tester ces définitions. Une séquence aléatoire sera alors une séquence dont on ne peut pas trouver statistiquement de programme plus petit la générant. Cette définition revient alors à dire que l'on ne peut pas compresser une séquence aléatoire avec un logiciel de compression si bon soit-il. Il est evident que dans le cas des PNRG, la complexité de Kolmogorov de ces séquences est forcément plus petite que la taille de la séquence, elle est au plus égale à la taille du PRNG correspondant.

2 Etat de l'art de la production et du test de séquences aléatoires

2.1 séquences aléatoires et pseudo-aléatoires

Il existe deux grandes familles de générateurs de séquences de nombres aléatoires : les générateurs physiques et déterministes.

2.1.1 Les générateurs physiques

On les appelle *True Random Number Generator* et produisent des séquences aléatoires. Ils doivent s'appuyer sur des sources physiques aléatoirement sûres. Certains peuvent se baser sur des phénomènes quantiques, ou sur le *bruit de Johnson* [12]. On peut aisément trouver de tels générateurs sur internet à des prix raisonnables [13]. La qualité de l'aléatoire de la source

12. bruit électronique produit par une resistance

13. On peut trouver ici un comparatif d'un grand nombre de générateurs : https ://en.wikipedia.org/wiki/Comparison_of_hardware_random_number_generators

peut être variable, il existe ainsi des extracteurs de hasards [2] permettant de produire des séquences aléatoires de bonne qualité à partir de générateurs basés sur des sources faibles. Les exemples les plus recurrent sont les boitiers générateur quantiques de nombres aléatoires, et les extensions usb basées sur le bruit de Johnson. Il existe une théorie à propos des différents bruits physiques, le site internet RANDOM.ORG [5] propose des nombres aléatoires issus du bruit atmosphérique, en d'autres termes il utilise des interférences radio pour produire des nombres aléatoires. Bien entendu ces différents générateurs doivent être testés statistiquement avant d'entrer sur le marché, nous en parlerons dans la prochaine partie.

2.1.2 Les générateurs déterministes

On les appelle *Pseudo Random Number Generator*. Ils s'appuient sur une *graine* -entier récupéré à partir d'une source physique. Puis, à l'aide d'une méthode algorithmique bien définie, déploie une séquence seulement à partir de cette graine. L'avantage de tels programmes est la génération très rapide et la possibilité de prouver la qualité de la sortie car toute suite est prédictible. L'inconvénient est que toute la séquence peut être déduite à partir de l'algorithme et de la graine, ce qui est possible si la graine provient d'une source non sécurisée (mouvement de souris, vitesse de rotation du disque dur, etc ...). Il peut paraître absurde de vouloir générer de l'aléatoire par un procédé déterministe [14], mais c'est ici que notre définition de l'aléatoire prend tout son sens. Une séquence aléatoire n'est définie comme découlant d'un procédé non-déterministe, elle doit seulement répondre au critère de non-predictabilité, si un observateur ne peut à aucun moment prédire quoi que ce soit (même statistiquement) de la séquence produite par un générateur, alors c'est une séquence aléatoire. Par conséquent le but d'une PRNG est de camoufler sa graine par des opérations complexes et ainsi empêcher à tout observateur de déduire le prochain élément.

La methode Von Neumann

En 1946 Joan Von Neumann propose a PRNG connu comme la méthode *middle-square*. Elle consiste à prendre le carré d'un nombre et d'en garder seulement les chiffres au centre, la valeur y correspondant sera utilisée comme valeur de départ de la prochaine étape.

Exemple :

1. On commence avec la valeur '1111',
2. $1111^2 = 1234321$
3. On retient les digits du centre : 3432
4. On continue avec 3432 comme valeur de départ

La valeur '1111' est appelée la graine et détermine toute la séquence.

L'algorithme décrit par Von Neumann utilisait des nombres de 10 chiffres, la période de cet algorithme est particulièrement faible et la qualité de la sortie dépend fortement de la graine. Par exemple '0000' est un état absorbant et produit toujours la même séquence.

Les générateurs Congruentiels Linéaires

Les générateurs Congruentiels Linéaires sont fondés sur une congruence et une fonction linéaire, l'algorithme est introduit par Derrick Lehmer en 1948

La séquence est définie par :

$$X_{n+1} = (a \cdot X_n + c) \mod m$$

a est le mutiplicateur, c l'increment and m le module.

14. «Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin.» -John Von Neumann (1951).

X_0 est la graine, il existe autant de séquences que de valeurs de départ, cependant certaines graines peuvent produire des séquences de mauvaise qualité.

A cause de l'opération modulo, les termes de la séquence sont à valeur dans $[0, m-1]$. De plus, chaque terme est entièrement dépendant du précédant, donc si un nombre apparaît une seconde fois, c'est toute la suite qui se reproduit à partir de celui-ci. Or, le nombre de valeurs possibles est fini (égal à m), la suite est donc obligée de se répéter. On parle de séquences utilement périodiques, ici la période est au maximum m.

Donald E. Knuth, célèbre informaticien américain a consacré tout un chapitre de son second volume de *The Art of Computer Programming[13]* aux nombres aléatoires. On y comprend les enjeux de la génération de nombres aléatoires, les méthodes de génération algorithmiques et de son test statistique. On retrouve notamment dans la dernière partie des conseils sur le paramètrage des Générateurs Congruentiels Linaires.

Ci dessous un exemple d'algorithme en C++ basé sur un LCG très mal paramétré en vue d'avoir un étalon de mauvais résultats :

```
int rand(void) // RAND_MAX assumed to be 32767
{
    next1 = next1 * 1103515245 + 12345;
    return (unsigned int)(next1/65536) % 32768;
}

void srand(unsigned int seed)
{
    next1 = seed;
}
```

Mais les LCG ne sont pas les seules méthodes possibles pour construire des séquences pseudo aléatoires. Marsenne Twister[14] par exemple est un des algorithmes les plus utilisés [15] car il passe tous les tests statistiques (en 1998 -soit en particulier Diehard) avec une période de $2^{19937} - 1$, en plus d'être relativement rapide. Il est disponible dans les librairies standard de la plupart des langages de programmation courants.

2.1.3 La technique Marsaglia[11]

Afin de pouvoir étudier les séquences de nombres aléatoires, il peut être intéressant de trouver un moyen de produire des séquences de bonne qualité afin d'établir un standard. Or nous avons vu au début que les vrais sources d'aléatoire n'existent en réalité pas, ce qui apparait aléatoire est en fait dû aux lacunes théoriques et expérimentales que nous avons du phénomène. Nous allons voir dans cette partie comment George Marsaglia, mathématicien et informaticien américain ayant consacré une grande partie de sa vie au sujet, a dans les années 90, procédé à de petites expériences afin de produire des séquences de nombres aléatoires de bonnes qualités.

Marsaglia, bien connu dans le domaine des nombres aléatoires a décidé en 1995 de publier un CD-ROM[10] distribué à de nombreux chercheurs du monde entier contenant les 'True Random Bits' que nous allons étudier, mais aussi le très célèbre Diehard battery of test que nous verrons plus tard.

Dans ce CD-ROM nous pouvons trouver plusieurs fichiers :

i) Les fichiers canada.bit germany.bit california.bit qui contiennent des nombres aléatoires provenant de sources physiques.

ii) Un ensemble de fichiers : $bit.i, \forall i < 60$ contenant 10Mo de nombres aléatoires fait à partir d'une fonction XOR d'une séquences d'un bon PRNG et de différentes sources physiques. Au total, 600 Mo de bon nombres aléatoires.

15. https ://en.wikipedia.org/wiki/Mersenne_Twister

Tout d'abord, Marsaglia a tenté de trouver des nombres aléatoires de qualité provenant de sources physiques. Il trouva une entreprise proposant :

> « a proprietary technology to create truly random numbers. Since the device is naturally occurring random phenomenon (Johnson Noise [16]) rather than a digital logic circuit or computer program, it requires no initial starting value and each new value is independent of all previous values»

Il acheta donc deux appareils, un provenant du Canada et un venant d'Allemagne, il avait aussi accès à une 'sortie aléatoire' d'un appareil situé en Californie. Il a essayé son test DieHard sur ces générateurs mais ils échouèrent de façon spectaculaire. Il est néanmoins intéressant de remarquer qu'un travail recent[6] montre que cet échec serait dû à une erreur de formatage des fichiers et non à la nature des générateurs. Robert Davis a essayé lui même les tests du DieHard avec les mêmes appareils et a obtenu différents résultats [17].

D'autre part, Marsaglia remarqua que beaucoup de générateurs déterministes passaient avec succès tous les test (The Kiss Generator and The Mother of All par exemple), il en vient à dire ceci à propos des séquences aléatoires. Si une séquence

$$x_1, x_2, ..., x_n$$

est vraiment une séquence aléatoire de bits indépendants et équiprobables, et que

$$y_1, y_2, ..., y_n$$

est une séquence quelconque de constantes ou bits aléatoires, alors la séquence

$$x_1 + y_1 mod 2, x_2 + y_2 mod 2, ..., x_n + y_n mod 2$$ [18]

est aussi une séquence aléatoire de bits indépendants.

Cela se montre très facilement de la sorte : Soit X une variable aléatoire uniformément distribuée, $\forall n, d \in \mathbb{N}, (X + n)/d$ est toujours une variable aléatoire uniformément distribuée, on applique en effet seulement une bijection.

Etant donné que seulement les bons PRNG passent le DieHard, Marsaglia considère les séquences renvoyées par ces PRNG comme x, et les séquences renvoyées par les générateurs physiques comme y.

Etant donné toutes ces données, il a construit 60 fichiers contenant les 'True Random Numbers' contenus dans : bit.01, bit.01, ..., bit.60. Ils on été formés en combinant des sorties de 32-bits provenant d'au moins un des générateurs déterministes cités plus haut, avec des entiers de 32-bits formés par des séquences binaires produites par les générateurs physiques. La plupart des fichiers hérite de la composante déterministe de la somme des générateurs Mother-of-All et KISS. Une partie des fichiers a aussi été combinée avec de la musique rap ou des images trouvées sur internet. Il utilise un XOR de toutes ces informations pour construire la séquence finale.

D'après Marsaglia, la fiabilité des fichiers finaux est causée par la qualité du générateur déterministe, les autres sources consistent essentiellement à ajouter un peu d'imprévisibilité.

La qualité des générateurs peut être très importante selon les utilisations. La cryptographie par exemple requiert une haute qualité afin d'éviter les attaques aux générateurs[9].

2.2 Tests statistiques

Lors de recherches bibliographiques sur les travaux produits en génération et test de l'aléatoire on s'aperçoit que deux tests sont cités dans quasiment tous les articles : Diehard et NIST

16. Bruit electronique produit par une resistance
17. On peut trouver dans l'article des résultats pour différents appareils de generation de nombres aléatoires.
18. Cela peut être vu comme un ou exclusif (XOR)

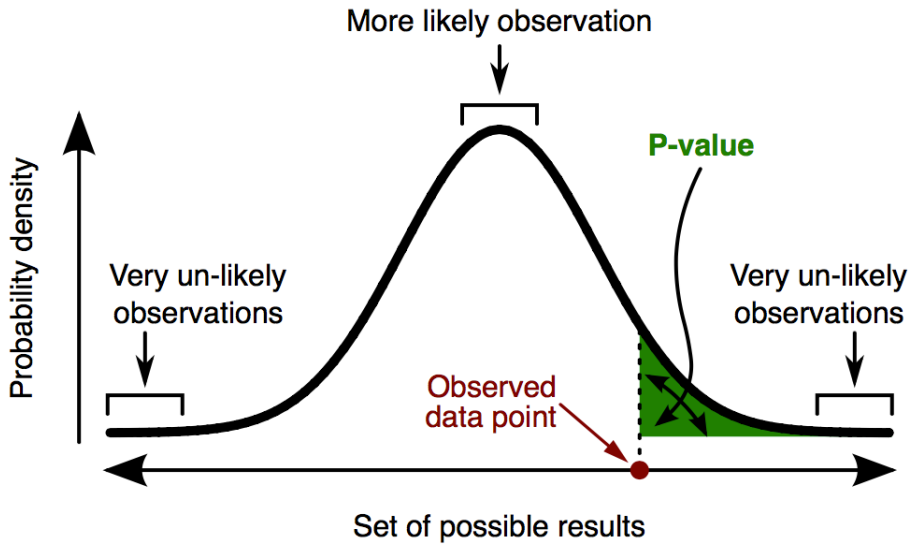
Test Suite. Ces tests font référence dans le domaine, mais se fondent sur des raisonnements et une conception du hasard très différent. Nous verrons dans les parties qui suivent que le Diehard nécessitait une réactualisation -le dieharder- mais plus que ça : qu'il était impératif de mettre au point une nouvelle manière de tester l'aléatoire.

2.2.1 p-value

Tous les tests d'aléatoire renvoient des *p-values*, essayons de définir ce qu'elles représentent. Quand on procède à un test statistique on peut vouloir savoir s'il faut rejeter ou non l'hypothèse nulle H_0. Dans notre cas l'hypothèse nulle est toujours : "notre séquence contient des nombres aléatoires indépendants et uniformément distribués".

La p-value nous indique la probabilité que sous l'hypothèse nulle, une variable aléatoire ait une valeur au moins aussi extrême qu'une valeur observée :

$$p - value(x) = 2 \cdot min\{Pr[X \leq x|H_0], Pr[X \geq x|H_0]\}$$



Si l'on répète ce même test sur un grand nombre de valeurs -l'ensemble des valeurs du fichier à tester- on a un ensemble de p-value. Si la séquence est réellement aléatoire, alors ces p-values doivent être uniformément distribuées dans $[0, 1]$. On peut utiliser un test de Kolmogorov-Smirnov ou χ^2 pour tester l'adéquation de la distribution des p-value à la loi uniforme, on obtient ensuite une unique p-value.

Si la p-value est très faible (e.g., 0,01%), c'est que l'événement aurait eu très peu de chance de se produire si la séquence était vraiment aléatoire, on peut donc penser que la séquence est suspecte. Pour clarifier ce point et ainsi permettre d'établir une méthode pour rejeter ou accepter H_0 on définit une valeur critique c pour chaque test de telle sorte que si la p-value est inférieure à cette valeur critique, alors on rejète H_0, sinon on l'accepte. Bien entendu cette méthode n'est pas rigoureuse, car un événement se produisant à faible probabilité peut se produire tout de même. On fait donc entrer des variables d'erreur dans notre méthode.

	CONCLUSION	
TRUE SITUATION	Accept H_0	Accept H_a (reject H_0)
Data is random (H_0 is true)	No error	Type I error
Data is not random (H_a is true)	Type II error	No error

Nous appellerons ces variables d'erreur α et β pour une erreur de type I et II respectivement. Il est important de choisir judicieusement notre valeur critique car nos variables d'erreur en

dépendent. La valeur de α est déduite directement de la valeur critique, en effet la probabilité de refuser un générateur valide est en fait

$$\alpha = Pr[\text{rejeter } H_0 | H_0 \text{ est vraie}] = Pr[X < c | H_0 \text{ est vraie}] = c$$

On peut désormais rejeter ou accepter H_0 avec un marge d'erreur α :

— Si $p - value \geq \alpha$, alors on accepte H_0 avec un seuil de confiance de (1-α)
— Si $p - value < \alpha$, alors on rejette H_0 avec un seuil de confiance de (1-α)

2.2.2 DieHard

Ce programme écrit en C par Marsaglia permet de tester n'importe quel générateur de nombre aléatoire à partir d'une séquence entre 10 et 12 Mo. Ce test implementant 15 tests statistiques. Trois d'entre eux sont précisément détaillés dans un article [12] de 2002 co-ecrit avec Tsang Wai Wan. Ces trois tests semblent être les plus difficiles à passer, les générateurs qui passent ces tests passent tous les autres, mais de nombreux générateurs ne les passent pas. Ils peuvent être vus comme une version distillée de Diehard.

The gcd Test

Ce test se base sur le calcul du PGCD de deux nombres à partir de l'algorithme d'Euclide, en voici un exemple :

On pose $u = 297$ et $v = 366$.
$$366 = 1 * 297 + 69$$
$$297 = 4 * 69 + 21$$
$$69 = 3 * 21 + 6$$
$$21 = 3 * 6 + 3$$
$$6 = 2 * 3 + 0$$

On appellera k le nombre d'iterations (ici k = 5), si on répète la procédure avec beaucoup de nombre aléatoires, on obtient une liste de k indépendants et uniformément distribués. Ce test s'intéresse à la distribution des k pour un nombre déterminé de nombres aléatoires présents dans le fichier à tester. Cependant, on ne connaît pas la distribution exacte des k. Marsaglia a donc décidé d'utiliser une analyse statistique à partir de 'standards aléatoires' pour en approcher la distribution.

The Gorilla Test

Imaginons un singe tapant sur une machine à écrire (on suppose les frappes du singe aléatoires, indépendantes et uniformément distribuées). Ce test s'intéresse à la distribution des mots écrits par ce singe. Pour ce faire il compte le nombre de mots de taille k manquant dans la séquence. Une fois encore Marsaglia decide d'approcher la distribution par une analyse statistique d'un 'standard aléatoire.'

The Birthday Spacing

On imagine n anniversaires choisis aléatoirement dans une année à $2^{32} - 1$ jours, ils sont ensuite triés. Le test s'intéresse au nombre d'espacements égaux entre deux anniversaires. La distribution est asymptotiquement distribuée par une loi de Poisson de paramètre $\lambda = m^3/(4n)$. Le problème de ce test et que pour une sous-séquence s il renvoie une seule p-value. Pour ce faire il fait passer le test à n nombres en partant du debut de la sous séquence, puis fait un décalage binaire pour passer à 'd'autres nombres'.

Afin de pouvoir combiner les p-values avec un test de Kolmogorov-Smirnov, il faut que les sous séquences s_i qui ont servi à les produire soient indépendantes, or ici ce n'est clairement pas le cas. Il est donc statistiquement erroné de faire ce que Marsaglia fait afin de retourner la p-values finale du test du fait du recouvrement des sous-séquences utilisées. On peut donc remettre en question la validité de ce test ainsi que des résultats qu'il fourni.

Discussion sur la construction des tests

Ce qui est particulièrement interpelant dans le raisonnement de Marsaglia est l'omnipresence d'analyse statistique pour determiner la distribution standard.

La méthode de Marsaglia pour construire ses tests semble être la suivant : on prend une situation que l'on peut modéliser avec notre fichier à tester, on calcule approximativement la distribution -la plupart du temps à l'aide d' *empirical study* et d' *extensive simulation*, ensuite on construit un test dans le but de comparer la distribution empirique du fichier à une distribution 'standard'. Finalement on juge de la qualité du test de façon empirique, si de nombreux PRNG suspects échouent le test, c'est qu'il doit être bien construit.

Marsaglia est très clair quant à sa relation avec la construction empirique : "

> « Note that we do not need the true distributions to develop a test of randomness. All we need do is compare the sample distribution from a particular RNG with a standard provided by a number a presumably good RNG's. »

Cependant, cela semble être une mauvaise idée. Tout d'abord, comment peut on construire un test à partir d'un standard aléatoire si le rôle du test est justement d'évaluer la qualité de l'aléatoire, alors le test ne fait que comparer la distribution à un "presumably good RNG". On ne decide donc pas de la qualité du générateur, mais de sa proximité par rapport à la distribution du standard.

Nous avons vu au debut de ce document que l'aléatoire était une notion très difficile à définir du fait que ne nous sommes même pas dur de sa presence dans la nature, on a tendance à appeler aléatoire seulement ce que nous ne savons pas expliquer -phénomène chaotique le plus souvent. Dans notre cas, si le test compare la distribution d'un phénomène chaotique à un générateur, on sera seulement capable de dire si ce dernier se comporte comme le phénomène physique -qui est biaisé de multiple façon, indiscernable à l'oeil humain, il faudrait pour les detecter de reel tests statistiques- et non pas s'il est aléatoire.

Il semble que le seul moyen valide de décrire l'aléatoire soit les probabilités. Ce que nous voulons tester dans notre séquence est que chaque valeur est uniformément distribuée et indépendante des autres. Par conséquent le meilleur moyen semble être de modéliser un situation que l'on peut expliquer essentiellement avec des probabilités théoriques en supposant l'indépendance et l'equiprobabilité de la distribution, et finalement comparer les résultats de notre générateur aux probabilités théoriques. Bien entendu cela ne veut pas dire que les test empirique sont faux, mais ils ne sont pas théoriquement valide, il faudrait effectuer des experiences afin de voir si les tests de Marsaglia testent bien la même chose que les tests du NIST qui quand à eux sont théoriquement valides.

Un autre point est que l'on ne connait pas le standard qui a été utilisé, peut être ses séquences du CD-ROM, or il explique justement que ces séquences on été construites à partir des tests statistiques du DieHard, le mystère reste entier.

2.2.3 DieHarder

DieHarder est la version revisitée par Robert G. Brown dont la dernière version date de 2016. Elle permet dans un premier temps d'installer le programme avec un simple sudo apt-get install sous linux -le diehard est trop vieux pour compiler avec le makefile original, cela rend l'installation très pénible. De plus, de nouveaux tests sont implementés, et chaque test renvoie une unique p-value permettant de rejeter ou non l'hypothèse nulle, Diehard donnait

pour la plus part des tests un ensemble de p-values que l'utilisateur devait interpreter selon ses connaissances en statistiques. Le point le plus important avec cette version et qu'elle permet d'interdire les recouvrement de sous-séquences -le fameux problème du *birthday spacings*, cela permet ensuite de pouvoir réellement combiner les p-values au sein d'une unique p-value servant à rejeter ou non H_0. Cependant la construction des tests reste la même et l'utilisation de 'standard' d'aléatoire reste toujours d'actualité.

2.2.4 NIST Test Suite

Ce test a été construit par l'Institut Notional des Technologies et Standards Americain, le raisonnement de construction ainsi que les details techniques sont décrits dans un document[16] très bien rédigé.

Le raisonnement de construction de l'équipe de NIST est très clair :

> « Randomness is a probabilistic property ; that is, the properties of a random séquence can be characterized and described in terms of probability. The likely outcome of statistical tests, when applied to a truly random séquence, is known a priori and can be described in probabilistic terms. »

Ce raisonnement est aux antipodes du raisonnement de Marsaglia qui fonde ses tests statistiques sur l'adéquation de séquences à une séquence 'standard'. Cela rend les tests de NIST beaucoup plus fiable et rigoureux, de plus il fait référence en qualité de test cryptographique. Nous les préférerons durant nos experiences personnelles.

2.2.5 Ent

C'est un test très simple à executer, il permet de donner de premieres information sur la possible faiblesse d'un générateur, il permet par exemple de voir s'il est possible de compresser la séquence avec une méthode standard, il est clair que si un de ces tests échoue le générateur est clairement faible. Nous l'utiliserons comme premier test afin d'éliminer certaines méthodes lors de nos experiences.

Il permet de calculer plusieurs valeurs dont on connait les valeurs attendus pour une séquence aléatoire, il suffit d'observer si certaines de ces valeurs sont suspectes [19].

3 Apports personnels

3.1 Générateur de π

Une autre idée a été de concevoir π comme un PRNG pour cela il m'a fallut écrire un générateur de ses décimales, il est bien entendu aisé de trouver des fichiers contenant pi sur internet, mais pas sous la forme d'un fichier binaire représentant pi comme un seul flottant, je me suis donc attelé à cette tache afin d'obtenir le fichier dont j'avais besoin pour les tests statistiques. J'ai malheureusement rencontré de nombreux écueils dans dans cette partie à cause des vitesses de convergence des suites qui était trop lente pour me permettre de produire le nombre de décimales dont j'avais besoin.

3.1.1 Combien de décimales ?

La premiere question est : de combien de décimales ai-je besoin afin de produire mes n bits, la réponse est simple :

19. Toutes les explications des tests : http ://www.fourmilab.ch/random/

On représente les décimales de pi comme un entier, on le nomme M. Si on a besoin de n bits, on peut dire que

$$M < 2^n + 1$$

Maintenant, il nous faut savoir combien de digits décimaux il nous faut pour representer $2^n + 1$, on doit donc connaitre x tel que

$$2^n + 1 \leq 10^x$$

$$\log 2^n + 1 \leq x$$

Par changement de base,

$$x = \frac{log_2\ 2^n + 1}{log_2 10} = \frac{n+1}{log_2 10}$$

Dans notre cas, DieHard nécessite un fichier f tel que $10Mo \leq size(f) \leq 12Mo$, on a donc,

$8\ .\ 10^7$ bits $\leq size(f) \leq 9,6.10^7$ bits

Et donc,

$$8.10^7/lg10 \leq x \leq 9,6.10^7/lg10$$

$$2,40.10^7 \leq x \leq 2,89.10^7$$

Nous avons donc besoin d'un flottant d'une precision x, on utilisera donc la librairie GMP pour construire un objet qui peut contenir un flottant de la precision désirée. La prochaine question est : quand devrons nous nous arrêter de calculer pour avoir suffisamment d'information correctes ? D'une autre façon : quelle est le plus petit n afin que la série soit stationnaire avec une precision x ?

3.1.2 Nilakantha

Dans un premier temps, j'ai utilisé la suite de Nilakantha :

$$\pi = 3 + \frac{4}{2*3*4} - \frac{4}{4*5*6} + \frac{4}{6*7*8} - \frac{4}{8*9*10} + \frac{4}{10*11*12} + ...$$

Nous avons besoin de x décimales, si on additionne N tel que $N < 10^{-x}$, alors la série est stationnaire. On cherche donc n tel que,

$$\frac{4}{n(n+1)(n+2)} < 10^{-x}$$

On veut donc résoudre

$$0 < n^3 + 3n^2 + 2n - \frac{4}{10^{-x}}$$

Et $\frac{4}{10^{-x}} = 4.10^x$
On cherche donc

$$0 < n^3 + 3n^2 + 2n - 4.10^x$$

On utilise la méthode classique de resolution d'une equation cubique :
Si on veut calculer la solution de

$$ax^3 + bx^2 + cx + d = 0$$

On pose p, q et r tels que

$$p = \frac{-b}{3a}$$
$$q = p^3 + \frac{bc - 3ad}{6a^2}$$
$$r = \frac{c}{3a}$$

Alors la solution positive est :

$$X = \sqrt[3]{q + [q^2 + (r - p^2)^3]^{1/2}} + \sqrt[3]{q - [q^2 + (r - p^2)^3]^{1/2}}$$

Dans notre cas, on trouve $X = \sqrt[3]{4.10^{2,4.10^7}}$
Ce qui est clairement trop grand pour être calculer sur nos ordinateur, nous devons donc trouver une autre série convergeant plus rapidement.

3.1.3 BBP Formula

Jetons un oeil à la formule de Bailey–Borwein–Plouffe[1]

$$\pi = \sum_{k=0}^{\infty} \left[\frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right) \right]$$

Nous voulons savoir s'il est raisonnable de calculer pi avec cette formule, nous savons que nous avons besoin de x décimales, nous appliquons le même raisonnement.

$$\frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right) \Big\langle 10^{-x}$$

Afin de simplifier le calcul on desiderera

$$\frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right) \text{ as } \frac{K}{16^n}$$

Avec K une constante arbitraire

$$\frac{K}{16^n} < 10^{-x}$$
$$log_{16}(K.10^x) < n$$

Par changement de log,

$$2.10^7 + log_{16}K < n$$

K étant petit, il est raisonnable de supposer

$$2.10^7 + log_{16}K \approx 2.10^7$$

On doit donc prendre n tel que

$$n > 2.10^7$$

Qui est éventuellement envisageable par mon ordinateur.

Pour être plus efficace nous calculerons la série sous une autre forme [20] :

$$\pi = \sum_{k=0}^{\infty} \frac{120k^2 + 151k + 47}{(2k+1)(4k+3)(8k+1)(8k+5)16^k}$$

L'avantage de calculer la série sous cette forme est que l'on peut calculer le numérateur et le dénominateur avec le type *mpz_t* [21] cela permet de n'effectuer qu'une seule division et de n'avoir qu'une seul *mpf_t* [22] à traiter.

$(2k+1)(8k+1) < 6 \cdot 10^{15}$
$(4k+3)(8k+5) < 1.28 \cdot 10^{16}$
$120k^2 + 151k + 47 < 7.68 \cdot 10^{17}$ [23].

Le type *unsigned long int* peut contenir des nombres jusqu'à 10^19, de plus je peux initialiser mes variables GMP depuis un *unsigned long int*. Je vais donc utiliser ce type afin de calculer le plus de termes que je peux, ce sera plus rapide que de les calculer dans un type GMP.

Calculons maintenant la complexité de l'algorithme,

1. La premiere partie fait une division par boucle avec une precision of n, nous faisons n boucles, la complexité est donc de $\Theta(n^2)$.
2. La seconde partie fait n multiplications avec une precision de n, la complexité est donc aussi de $\Theta(n^2)$.

Nous observons ces temps d'execution suivants :

1. Pour 100 digits : 0.00043 seconds
2. Pour 1000 digits : 0.00232 seconds
3. Pour 10,000 digits : 0.04756 seconds
4. Pour 100,000 digits : 12.144 seconds
5. Pour 800,000 digits : 1470.15 seconds

Nous savons que la complexité exacte est donnée par $C(n) = k \cdot n^2$, maintenant que nous avons quelques valeurs, nous pouvons approcher k.

$$k_{moyen} = 4.5 \cdot 10^{-10}$$

Par conséquent, pour $n = 8 \cdot 10^7$ le programme prendra 33 jours à faire le calcul. [24]. Il est clair que cela n'est pas envisageable durant mon stage, je dois donc trouver une autre methode pour calculer mon fichier binaire de pi.

3.1.4 Convertir π

Il est très aisé de trouver des décimales de pi en ascii sur internet, j'ai pu télécharger un fichier de 1 Go [25] et créer un programme qui extrait le bon nombre de décimales et les convertis en representation binaire. Cela est un peu plus compliqué qu'il n'y parait car on ne peut pas convertir octet par octet ou décimale par décimale, il est nécessaire d'avoir la representation dans un flottant ou entier, et ensuite de le convertir par une méthode choisie. Ma méthode fut

20. On peut tester l'égalité avec la formule BBP à l'aide du site WolframBeta .
21. type de GMP implémentant de grands entiers
22. Grand flottant
23. On le sait car on ne fera jamais plus de $8 \cdot 10^7$ tours de boucle.
24. J'ai construis deux autres programmes dans le but de calculer en parallèle et de faire tourner le programme sur le serveur de calcul du LIRMM. Un parallelise le calcul à n fixé, l'autre fait la même chose mais en parallelisant aussi pour différents valeurs de n. Aucun de ces deux programmes d'améliore la vitesse d'execution malgré 100 coeurs sur le serveur. Cele est surement du au fait que les données sont trop grandes pour entrer dans le cache des coeurs.
25. http ://www.mit.edu/afs.new/sipb/contrib/pi/

de lire le fichier dans un *char** et de le convertir en grand entier. Enfin, j'ai utilisé la méthode par division par 2 pour le convertir en binaire. La complexité de l'algorithme est en $\Theta(n^2)$, voici les temps d'execution :

1. n = 10 : 0.000159 seconds
2. n = 100 : 0.000248 seconds
3. n = 1,000 : 0.000323 seconds
4. n = 10,000 : 0.00143 seconds
5. n = 100,000 : 0.053 seconds
6. n = 1,000,000 : 5.0 seconds
7. n = 10,000,000 : 553.916 seconds

On peut aisément espérer un temps d'execution d'approximativement 9 heures pour $n = 8 \cdot 10^7$ [26].

3.2 Construction d'un générateur de qualité cryptographique

Nous avons eu l'idée de faire des experience sur les séquences aléatoires physiques. Où trouver de bonnes séquences aléatoires ? Il se trouve que la notion physique de bruit est une des sources les plus utilisés pour cela, nous avons alors eu l'idée de tester la qualité aléatoire du bruit d'une carte son. En effet, de tels périphérique produise un bruit blanc si l'on pousse le gain [27] au delà d'une certaine valeur, et ce même si aucun cable/microphone n'y est branché. Pour cette experience j'ai décidé d'utiliser une de mes cartes son personnelles : M-audio Fast Track Pro.

3.2.1 Materiel et méthode

Materiel

La premiere chose fut de savoir comment récupérer les valeurs enregistrées par la carte son sans aucune modifications, nous avons en effet remarqué que si l'on enregistrait ce bruit à l'aide d'un logiciel de musique avancé -ableton live par exemple- le fichier de sortie, malgré le fait d'être non compressé et encapsulé dans un fichier *.wav*, ne ressemblait pas ce qu'une transformation basique Anologique/Numerique devrait produire : une séquence de valeur de 16bits représentant la courbe du bruit (visible sur notre logiciel).

Nous avons décidé d'utiliser un logiciel plus basique : *arecord* dont une simple experience nous a permis de verifier la qualité originelle des valeurs. Nous avons enregistré un son quelconque puis nous l'avons envoyer vers le flux se sortie de la carte son, si le bruit était identique au fichier alors les valeurs sont identiques à celles récupérées par la carte son, ce fut le cas.

Pour l'enregistrement du bruit à proprement parlé, la réponse est simple, le bruit est tellement bas que le seul moyen est de pousser le gain au maximum, en utilisant les réglages permettant d'augmenter au maximum le niveau. L'enregistrement en mono fut choisi car il est loins d'être evident que les deux canaux soit indépendants. Finalement il faut realiser des experiences pour choisir la taille des mots (8,16 ou 25 bits) et la fréquence d'échantillonnage.

Au moment de l'experience j'avais à ma disposition une ancienne radio portative avec une sortie mini-jack. Nous avons donc voulu aussi essayer d'enregistrer le bruit produit par cette radio quand elle est réglée sur une fréquence extreme et ne produit que du bruit blanc. En montant le gain au maximum et le volume de la radio au minimum cela permet d'enregistrer les deux bruits simultanément.

26. Car $k_{moyen} = 5.26 \cdot 10^{-12}$
27. niveau d'entrée

Methode

La méthode est simple, il faut enregistrer un grand nombre de fois le bruit en faisant varier chacun des paramètres et ainsi avoir plusieurs repetitions de chaque valeur de paramètre. Il est plus que supposable que la séquence brute sorite de la carte son ne soit pas tout à fait aléatoire : les valeurs risques d'être très basse en raison du faible niveau du bruit, de plus chaque valeur n'est pas tout à fait indépendante de son voisinage en raison de sa qualité de modélisation d'une courbe analogique. Il faudra donc améliorer cette séquence, nous avons choisi de choisir un bit par valeur en prenant une valeur sur d. Il faut donc choisir la valeur d du décalage et la position du bit que l'on veut choisir, ce serait dommage de laisser ces paramètres au hasard.

Les tests visant à comparer plusieurs valeur d'un même paramètre ont été realiser avec le programme *ent* car rapide d'execution et donnant des résultats clairs.

3.2.2 Resultats

Nous avons remarqué qu'à la vue du rapport qualité aléatoire/perte de vitesse les meilleurs paramètres d'enregistrement était 16bits pour une fréquence de 44kHz.

On observe immédiatement que le bruit venant de la carte son ne peut en l'état être considerée comme aléatoire, malgré le fait que les operations de décalage l'ait nettement amélioré. Le bruit venant de la carte son et de la radio quant à lui semble être de bonne qualité, ent ne nous montre en effet aucune lacune lors de ses tests. Ces résultats ont été vérifiés et validés lors d'une seconde session de test avec NIST Test Suite. On remarque en effet que les fichiers issus du bruit de la carte son et de la radio en passent tous les tests. cf. Figure 2, Figure 3, Figure 4 et Figure 5 en annexe pour le tableau précis des résultats.

Dans un second temps on voit que le fait de ne retenir q'un seul bit par nombre permet d'améliorer grandement la qualité, mais que la position du bit choisi ne semble pas avoir d'influence.

3.2.3 Discussion

Les résultats precedents valident notre méthode d'amelioration de l'aléatoire basée sur du bruit physique. De plus, il nous permet de découvrir une source d'aléatoire de qualité cryptographique : un mélange de bruit électronique et de bruit atmosphérique. De plus nous observons que plus ou moins toutes les valeurs des paramètres sont bonnes, il est alors possibles d'utilisers toutes les digits du fichier ainsi que tous les bits et ainsi produire un fichier de la même taille que le fichier initial mais passant tous les tests du NIST. De manière plus générale cela nous permet de produire un flux de nombres aléatoires d'une qualité cryptographique avérée en temps réel avec un debit théorique de :

$$4,4 \cdot \text{ kbits/s}$$

En effet on enregistre $44 \cdot 10^3$ sample de 16bits par seconde et on en retient que 1 bit sur 160.

Une seconde étape de l'experience aurait été de reproduire chaque experience un grand nombre de fois afin d'obtenir des valeurs plus precises et fiables avec des écart-types cohérents. Sera nous aurait permis d'expliquer plus de choses quant aux résultats -pourquoi certaines positions de bits semblent être un petit meilleures, pourquoi certaines décalages fonctionnent mieux que d'autres- la fiabilité actuelle des valeurs ne nous permet même pas de valider significativement ces écarts de valeurs.

Ce générateur peut tout de même être utilisé comme générateur dans de nombreux domaines nécessitant une grande qualité d'aléatoire sans reproductibilité. Les applications cryptographique sont alors multiples car il est impossibles de prédire quoi que ce soit de la séquences produites, ni par espionage materiel, ni par attaque algorithmique.

Conclusion

Ce stage m'a permis de me confronter à de réels problèmes scientifiques où une bibliographie conséquente et une bonne compréhension des notions étaient essentiels, de comprendre un grand nombre de choses quant aux séquences aléatoires et leur production mais aussi de répondre à des questionnements personnels relatifs au sujet. J'ai pu ainsi mettre en place des techniques apprises durant ma scolarité et plus précisément durant mon année à l'Ecole Normale Supérieure. Je tiens à remercier chaleureusement M. Alexander Shen qui a su m'encadrer avec beaucoup d'attention, de sympathie et de rigueur. Il m'a permis d'effectuer ce stage dans les meilleures conditions et ainsi de m'épanouir pleinement dans mon travail.

Références

[1] David H. Bailey. A compendium of bbp-type formulas for mathematical constants. 2013.

[2] B.Bencsath and !.Vajda. Collecting randomness from the net. *R. Steinmetz et al. (eds.), Communications and Multimedia Security Issues of the New Century*, 2008.

[3] Rittaux Benoit. *Mathématiques et philosophie.* édition tangente edition, 2010.

[4] G. Chaitin. *Hasard et complexité en mathématiques.* Flammarion edition, 2009.

[5] Kenny Charmaine and Mosurski Krzysztof. *Random Number Generators : An Evaluation and Comparison of Random.org and Some Commonly Used Generators.* The distributed systems group, computer science department, tcd from trinity college dublin edition, 2005.

[6] Robert Davies. True random number generators. *Personal Webpage : http : //www.robertnz.net/true$_r$ng.html*, 2013.

[7] Paul Deheuvels. *La probabilité, le hasard et la certitude.* Que sais-je ? puf edition, 2008.

[8] Jean-Paul Delahaye. *Information, complexité et hasard.* Hermes edition, 1999.

[9] Dörre Felix. Attacks on random number generators. *On internet*, 2009.

[10] Marsaglia George. Diehard cd-rom : A battery of tests of randomness. *http : //stat.fsu.edu/geo/diehard.html*, 1996.

[11] Marsaglia George. The marsaglia random number cd-rom. *Note on the marsaglia CD-ROM : cdmake.ps*, 1996.

[12] Marsaglia George and Tsang Wai Wan. Some difficult-to-pass tests of randomness. *J. Statist. Soft. 7, 3, 1–9*, 2002.

[13] Donald E. Knuth. *The Art Of Computer Programming - Seminumerical Algorithms (3rd edition Volume 2).* Reading, massachusetts : Addison-wesley edition, 1997.

[14] Matsumoto Makoto and Nishimura Takuji. Mersenne twister : A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modelling and Computer Simulations : Special Issue on Uniform Random Number Generation*, 1998.

[15] Henri Poincaré. *Calcul des probabilités.* éditions jacques gabay edition, 1987.

[16] Andrew Rukhin and Al. A statistical test suite for random and pseudorandom number generators for cryptographic applications. *NIST Special Publication 800-22 Revision 1at*, 2010.

[17] Spinoza. *L'éthique.* 1677.

4 Annexe

4.1 Contexte socio-professionnel

Le stage s'est effectué au Laboratoire d'Informatique, Robotique et Micro-électronique de Montpellier, au sein de l'équipe ESCAPE qui se charge de la branche la plus théorique du laboratoire. L'essentiel de mon stage s'est effectué en anglais, les éléments présents dans ce document ont donc été, pour la plupart, rédigé une premiere fois en anglais puis traduits pour être présentés sous cette forme. La partie collective de mon travail s'est organisée autour de discussions entre stagiaires, avec les doctorants du laboratoire, de conseils avisés de la part de Andreï Romashchenko, et finalement de rencontres et échanges par mail réguliers avec Alexander Shen, mon encadrant.

4.2 Impressions

Ce fut la seconde fois que j'effectuais un stage au sein de l'équipe ESCAPE. Le premier était dirigé durant l'été de fin de L2 par Mlle Sabrina Ouazzani doctorante de Bruno Durand et Gregory Lafitte. Durant ce stage j'ai eu la chance de pouvoir réaliser un travail très personnel -modélisation de langage musicaux- en autonomie. Ce dernier m'a permis de m'initier à la démarche académique dans un laboratoire de recherche ainsi qu'à l'organisation de mon travail.

Cette année j'ai pu me confronter à de nouveaux aspects du travail de chercheur. Dans un premier temps, je fus amené à découvrir le travail en équipe. Il m'a alors fallu m'organiser afin d'avancer sur un projet tout en pouvant mener des investigations personnelles.

Ce stage m'a aussi permis de faire mes propres erreurs, dont la plus grande fut de me lancer tête baissée dans une idée sans faire en amont un travail théorique suffisant. Ce fut le cas lors de la conception du générateur de pi, j'ai trop souvent entamé la réalisation d'un programme sans vérifier si ses temps d'exécution étaient raisonnables. Mais aussi durant les expériences : mon but était de trouver une méthode pour produire des séquences aléatoires de qualité, en oubliant que mon travail était aussi de présenter des expériences reproductibles. Il m'a alors fallut refaire une partie des expériences après coup en notant toutes les valeurs afin de les compiler dans ce document.

Pour terminer, ce stage m'a permis de comprendre que la reproductibilité n'est pas qu'une vertu des sciences expérimentales. Lors de la lecture du travail d'un mathématicien, il est fondamental de vérifier chacune des étapes de son raisonnement et de reproduire son travail afin d'en tester la véracité. Il peut même être très dangereux de réutiliser un résultat sans effectuer ce travail.

4.3 Tableaux des résultats

bits/test	Entropy by bytes	Compression	Chi-Square p-value	Aritmetic mean (125,5 is random)	Monte Carlo value of pi (% error)	Serial correlation coefficient (0 is random)
Temoin 1 (sans décalage)	3.953667	50	0.01	143.3368	29.73	0.569856
Temoin 2 (avec décalage)	4.932149	38	0.01	139.7138	20.65	0.230091
1 (poids faible)	7.949923	0	0.01	139.3287	9.6	0.001438
2	7.939722	0	0.01	140.4728	10.66	0.000195
3	7.950077	0	0.01	139.3573	9.69	0.00059
4	7.939955	0	0.01	140.5596	10.68	0.000334
5	7.950477	0	0.01	139.3305	9.57	-0.00146
6	7.939535	0	0.01	140.5479	10.94	0.000184
7	7.950135	0	0.01	139.3168	9.7	-0.000539
8	7.939771	0	0.01	140.4795	10.72	0.000689
9	7.950331	0	0.01	139.3538	9.7	-0.000187
10	7.939784	0	0.01	140.4804	10.59	0.001203
11	7.950477	0	0.01	139.3184	9.68	-0.000568
12	7.940026	0	0.01	140.4896	10.77	-0.000465
13	7.950216	0	0.01	139.3081	9.57	0.000136
14	7.939811	0	0.01	140.4799	10.78	-0.000873
15	7.950093	0	0.01	139.3697	9.62	0.000379
16 (poids fort)	7.940047	0	0.01	140.496	10.73	0.000374

FIGURE 2 – Resultats position bit pour la carte son

Décalage/test	Entropy by bytes	Compression	Chi-Square p-value	Aritmetic mean (125,5 is random)	Monte Carlo value of pi (% error)	Serial correlation coefficient (0 is random)
Temoin (pas de décalage)	7.937509	0	0.01	139.2864	10.87	0.001291
2	7.949775	0	0.01	139.3224	9.55	-0.002777
3	7.950657	0	0.01	139.2466	9.62	-0.000957
4	7.949832	0	0.01	139.3201	9.31	-0.000605
5	7.950472	0	0.01	139.3051	9.74	-0.00058
6	7.950568	0	0.01	139.2479	9.5	0.000887
7	7.900782	1	0.01	144.1608	14.22	-0.000302
8	7.950387	0	0.01	139.2972	9.66	0.000605
9	7.950303	0	0.01	139.3165	9.65	-0.000231
10	7.950477	0	0.01	139.3184	9.68	-0.000568
11	7.950131	0	0.01	139.2334	9.57	0.000407
12	7.950657	0	0.01	139.2355	9.75	0.000726
13	7.950191	0	0.01	139.2681	9.73	0.002053
14	7.900826	1	0.01	144.2292	14.06	0.000458
15	7.95044	0	0.01	139.3637	9.51	0.00127
16	7.950336	0	0.01	139.2694	9.72	-0.00105

FIGURE 3 – Resultats décalage pour la carte son

bits/test	Entropy by bytes	Compression	Chi-Square p-value	Aritmetic mean (125,5 is random)	Monte Carlo value of pi (% error)	Serial correlation coefficient (0 is random)
Temoin 1 (sans décalage)	6.588884	17	0.01	127.476	28.85	0.041565
Temoin 2 (avec décalage)	7.119046	11	0.01	127.5005	4.52	0.044818
1 (poids faible)	7.999724	0	72.29	127.5149	0.11	-0.001061
2	7.999702	0	39.11	127.398	0.17	0.000195
3	7.999706	0	46.47	127.5842	0.09	-0.000867
4	7.999735	0	84.6	127.3443	0.28	0.000035
5	7.999721	0	67.67	127.4008	0.14	0.001699
6	7.999739	0	88.66	127.4305	0.15	-0.000702
7	7.999718	0	63.66	127.4763	0.19	-0.001628
8	7.999714	0	58.51	127.4063	0.33	-0.001298
9	7.999727	0	76.12	127.3601	0.09	0.002562
10	7.999724	0	72.53	127.5524	0.04	0.00081
11	7.99971	0	53.03	127.3855	0.19	0.001315
12	7.99969	0	22.99	127.3947	0.03	-0.001797
13	7.999692	0	25.8	127.3191	0.07	0.000515
14	7.999671	0	7.7	127.4999	0.17	0.001003
15	7.999716	0	62.7	127.5185	0.31	0.000146
16 (poids fort)	7.999701	0	38.32	127.2813	0.11	-0.001134

FIGURE 4 – Resultats position bit pour la carte son et la radio

Décalage/test	Entropy by bytes	Compression	Chi-Square p-value	Aritmetic mean (125,5 is random)	Monte Carlo value of pi (% error)	Serial correlation coefficient (0 is random)
Temoin (pas de décalage)	7.871849	1	0.01	127.4683	4.2	-0.009133
2	7.997713	0	0.01	127.462	0.47	0.000064
3	7.999721	0	0.01	127.5192	0.1	-0.000664
4	7.999569	0	0.01	127.5109	0.03	-0.001396
5	7.999492	0	0.01	127.4661	0.24	-0.000209
6	7.999624	0	0.01	127.4863	0.24	0.000705
7	7.995163	0	0.01	127.5285	0.73	-0.003789
8	7.999453	0	0.01	127.5002	0.13	-0.000618
9	7.999728	0	32.56	127.454	0.09	-0.001043
10	7.999724	0	72.53	127.5524	0.04	0.00081
11	7.999661	0	25.4	127.3378	0.21	0.001787
12	7.999681	0	83.8	127.4594	0.05	0.00113
13	7.999563	0	5.01	127.6153	0.01	-0.000533
14	7.999623	0	82.08	127.4949	0.02	-0.003349
15	7.99951	0	9.13	127.558	0.15	-0.000474
16	7.999524	0	40.56	127.446	0.4	-0.000292

FIGURE 5 – Resultats décalage pour la carte son et la radio