Générateurs pseudo-aléatoires

« ...its very name RANDU is enough to bring dismay into the eyes and stomachs of many computer scientists! »

Donald E. Knuth

1 Introduction

1.1 Récupération des fichiers

Depuis la ligne de commande, naviguez vers un répertoire vous appartenant, et exécutez la commande suivante, qui téléchargera et décompressera un répertoire nommé `random` contenant plusieurs fichiers dont une source C :

```
curl cdn.sci-phy.org/mp2i/tp10-random.tgz | tar xvz
```

Le fichier à compléter est le fichier `random.c`. On y trouvera quelques fonctions utiles qui seront présentées au fur et à mesure de nos besoins. On y trouve également un fichier `makefile` aidant à la compilation, et le module permettant de créer des images `ppm` que nous avons déjà utilisé à plusieurs reprises.

1.2 Buts de l'étude

Dans la vie de tous les jours, si l'on souhaite obtenir un élément choisi aléatoirement parmi un ensemble fini E, la solution la plus simple consiste à lancer un dé équilibré ayant un nombre de faces égal au cardinal de l'ensemble en question, après avoir associé chacune des faces du dé à un des éléments de E. Par exemple, si $E = [\![1 \,..\, 6]\!]$, on peut utiliser un dé « courant » à six faces.

Cela fonctionne car le jet de dé est chaotique : une toute petite variation des conditions initiales (que l'on ne peut reproduire de façon exactement identique à chaque lancer) causera un changement dans le résultat obtenu, et l'on obtiendra les six résultats de façon à peu près équiprobables (des expériences récentes ont montré que le résultat n'était pas rigoureusement équiprobable, même pour un dé équilibré, si le dé est toujours lancé avec la même orientation initiale).

Pour un ordinateur, par construction un système déterministe, la tâche est plus ardue. On peut toujours créer un dispositif contrôlé par l'ordinateur lançant un dé physique et utilisant une caméra pour visualiser le résultat (ce qui, comme tout projet farfelu, a évidemment été fait : http://gamesbyemail.com/News/DiceOMatic) mais outre le fait que cela ne garantira pas une équiprobabilité rigoureuse des résultats, chose qui peut être importante pour des simulations numériques, un tel dispositif est très peu pratique.

Il a donc fallu trouver des sources d'entropie, l'équivalent microscopique d'un lancer de dé. Certains ordinateurs sont équippés d'un dispositif électronique créant un bruit d'origine quantique, qui peut être utilisé pour obtenir des bits aléatoires. Plus généralement, on se sert de l'environnement de l'ordinateur, imprévisible, pour obtenir de tels bits aléatoires. Par exemple, les mouvements de la souris (si vous déplacez votre souris deux fois, il est peu probable que vous soyez capable de faire le même mouvement au micromètre près et à la microseconde près).

Seulement, ces sources d'entropie ne permettent généralement pas de produire plus d'une poignée de valeurs aléatoires par seconde. Pour une application qui en a besoin d'une grande quantité, comme celles que nous allons étudier, on préfère en général utiliser des générateurs *pseudo-aléatoires*.

Ce sont des algorithmes qui fournissent des séquences de valeurs (entiers, flottants...) de façon déterministe, mais ayant l'apparence du hasard. Il s'agit toujours de la même suite de valeurs, mais elle est tellement longue ($2^{19937} - 1$ valeurs pour l'un des plus populaires, utilisé par le module `random` de Python, le *Mersenne Twister*) qu'on ne s'apercevra pas, dans la pratique, de sa périodicité. Il suffit alors de choisir un point de départ au hasard dans cette séquence (ce point de départ est appelé *graine*, ou *seed* en anglais).

Le langage C fourni notamment un générateur pseudo-aléatoire au travers de la fonction `rand` qui retourne un entier entre `0` et `RAND_MAX` (inclus). Il ne garantit cependant pas grand-chose sur sa qualité, si ce n'est que sa période doit être au moins égale à 2^{32}.

2 Générateurs pseudo-aléatoires

Puisque les générateurs pseudo-aléatoires génèrent une suite de valeurs $(u_n) \in \mathbb{N}$, beaucoup utilisent une relation $u_{n+1} = f(u_n)$ pour calculer les différents u_n. La difficulté est de choisir correctement la fonction f.

Une fonction très utilisée, utilisée pour les générateurs de Lehmer (ou générateurs de Park-Miller), est la fonction $f(u_{n+1}) = (a \times u_n + c) \mod m$ où a, c et m sont des entiers naturels, et l'opérateur $\mod$ calcule le reste de la division entière (ici par le diviseur m). De façon évidente, on obtient des valeurs dans $[\![0 \,..\, m-1]\!]$. La valeur initiale u_0 correspond à la graine choisie pour la séquence. De très nombreuses possibilités ont été proposées pour les valeurs de `a`, `c` et `m`. Parmi celles-ci, citons :

nom	a	c	m
RANDU	65539	0	2^{31}
Knuth	1664525	1013904223	2^{32}
Standard Minimal	16807	0	$2^{31} - 1$

Dans un premier temps, on souhaite écrire des fonctions permettant de générer les

valeurs u_1 à u_n pour chacun des trois générateurs précédents. Ces valeurs seront rangées dans un tableau de n entiers non-signés 32 bits fourni à cet usage.

1. Compléter les fonctions `gen_XXX(uint32_t seed, uint32_t tab[], int n)` pour implémenter chacun des trois générateurs pseudo-aléatoires. On réfléchira à la meilleure façon d'utiliser les opérations bit-à-bit du langage et les propriétés des non-signés pour implémenter certaines opérations.

On rappelle que les types `uint16_t`, `uint32_t` et `uint64_t` représentent des entiers non signés sur respectivement 16, 32 et 64 bits, que les calculs sur les entiers non signés sont effectués en arithmétique modulaire et qu'un « et » et un « ou » logiques sur les bits de deux entiers peuvent être obtenus avec les opérateurs « `&` » et « `|` ».

On fournit une fonction `disp` prenant en argument une fonction génératrice de valeurs pseudo-aléatoires, une graine et un nombre de valeurs à déterminer, et génère puis affiche ces nombres pseudo-aléatoires.

2. Afficher les dix premières valeurs pour chacun de ces trois générateurs pour la graine 42, et **vérifier le bon fonctionnement des trois fonctions**.

RANDU : 2752638, 16515450, 74318958, 297274698, 1114777566, 1865709466, 1161258702, 913585258, 1472634174, 613537722

Knuth : 1083814273, 378494188, 2479403867, 955863294, 1613448261, 110225632, 1921058495, 508781842, 3753001289, 4271921684

Standard Minimal : 705894, 1126542223, 1579310009, 565444343, 807934826, 421520601, 2095673201, 1100194760, 1139130650, 552121545

Il vous est également fourni, dans le fichier `random.c`, un générateur pseudo-aléatoire d'un principe différent : il s'agit d'une version simplifiée d'un *Mersenne Twister* (appeée TinyMT32), qui n'est pas lui-même sans défaut mais qui pourra servir de point de comparaison pour les tests.

3 Premiers tests

3.1 Introduction

Tous les générateurs pseudo-aléatoires ne sont donc pas de bonne qualité. Il nous faut des outils permettant de les comparer. L'ennui, c'est qu'il est impossible de déterminer si une séquence de valeurs a les caractéristiques d'une séquence « aléatoire » : pour un générateur parfaitement aléatoire, la probabilité pour qu'il retourne successivement mille ou un million de fois la valeur 1 est certes très faible, mais pas nulle.

Les premiers générateurs pseudo-aléatoires proposés en informatique n'étaient pas très bons. Cela a une conséquence importante : si un ordinateur a vérifié une propriété ou calculé quelque chose avec un corpus qui n'était pas aussi aléatoire qu'il l'aurait dû, le résultat obtenu peut être remis en cause.

Il a donc dû falloir développer des outils statistiques qui permettent de déterminer si un générateur pseudo-aléatoire est « bon » ou pas. Il existe de nombreuses suites de tests permettant d'évaluer la qualité d'un générateur aléatoire, nous allons étudier quelques critères qu'un générateur pseudo-aléatoire devrait vérifier la plupart du temps pour que l'on puisse le considérer comme un générateur de qualité raisonnable.

3.2 Fréquences

On peut déjà souhaiter que chacune des valeurs apparaissent, et qu'il n'y ait pas de valeurs indûment favorisées par rapport à d'autres. Comme les générateurs fournissent des entiers entre 1 et m, il faudrait générer une énorme quantité de nombres afin de pouvoir voir s'ils sont tous présents, avec des fréquences d'apparition compatibles avec une équiprobabilité de leur sortie.

Pour réduire les besoins, nous allons étudier les fréquences des restes des nombres générés par une division entière par 256 (en d'autres termes, nous allons étudier les fréquences d'apparitions pour chacun des 256 octets de poids faible des nombres générés).

3. Compléter la fonction `stats` qui prend en argument une fonction générant des nombres aléatoires, une graîne, un nombre n de tirages et un nom de fichier `ppm` pour ranger le résultat, alloue un tableau de n entiers non-signés 32 bits, le fait remplir par la fonction fournie en paramètre, calcule dans un tableau `count` le nombre d'apparitions de chacun des restes par une division entière par 256, et construit une image représentant un histogramme (cette dernière partie est déjà programmée).

4. Tracer et étudier les histogrammes pour $n = 10000$ pour chacun des quatre générateurs, avec une graine 17. Quel(s) est (sont) celui (ceux) qui passe(nt) le test de façon satisfaisante ?

En fait, les résultats décevants de certains générateurs ne sont pas nécessairement rédhibitoires. Pour obtenir de bons résultats, mieux vaut que m soit un premier, ce qui n'est pas le cas bien évidemment pour deux de nos générateurs. Lorsque m n'est pas premier, les bits de poids faibles sont de mauvaise qualité aléatoire, mais les bits de poids fort peuvent être de qualité.

5. Proposer une fonction `uint32_t extrait(uint32 n, int i, int p)` prenant en argument un entier non-signé sur 32 bits $n = a_{31}a_{30}\dots a_1a_0$ et construisant et retournant un entier non-signé sur 32 bits $n' = b_{31}b_{30}\dots b_1b_0$ de la façon suivante :

- pour $j < p$, $b_j = a_{i+j}$;
- pour $j \geqslant p$, $b_j = 0$.

Cela revient en pratique à écrire $n' = \left\lfloor \frac{n}{2^i} \right\rfloor \mod 2^p$.

Cependant, on cherchera à construire le résultat avec des opérations bit à bit (telles que `>>`, `<<` ou `&`) et éventuellement une addition ou une soustraction, mais sans division ou utilisation de l'opérateur `%`.

6. Modifier la fonction `stats` pour qu'elle prennent en compte les bits 23 à 30 (inclus) des entiers générés plutôt que les bits 0 à 7 (inclus) comme précédemment.

7. Tester à nouveau les quatre générateurs. Les résultats sont-ils meilleurs ?

3.3 Génération de nombres flottants

Dans la suite de nos tests, nous aimerions pouvoir générer des flottants dans l'intervalle $[0,1[$, avec une distribution « uniforme » sur cet intervalle [1]

Pour ce faire, on va générer des valeurs entre 0 et 2^{31} avec un de nos générateurs pseudo-aléatoire (on laissera tomber le bit de poids fort pour le générateur de Knuth, et on négligera le fait que le générateur standard minimal ne peut pas retourner $2^{31}-1$), et on divisera les résultats par 2^{31} dans un calcul en double précision.

8. Génère-t-on de cette façon bien des flottants entre 0 et 1 (exclu) ? Tous les flottants entre 0 et 1 peuvent-ils être obtenus ?

9. Compléter la fonction `gen_double` qui prend en argument un générateur pseudo-aléatoire, une graîne, un tableau avec assez de place pour n flottants en double précision et un entier n, et remplit le tableau avec des flottants aléatoire.

3.4 Valeurs successives

Même si toutes les valeurs apparaissent de façon équiprobable, cela ne suffit pas pour qu'un générateur pseudo-périodique soit de qualité. Il est nécessaire d'avoir une certaine indépendance entre un tirage et celui qui le suit.

Pour le vérifier, on va générer 40000 nombres flottants aléatoires, les considérer deux par deux comme deux flottants x et y, et tracer 20000 de points de coordonnées (x,y). Les points, dont les coordonnées en x et y seront comprises entre 0 et 1, seront placés dans une image de taille 500×500, le point $(0,0)$ devant être le pixel en bas à gauche de l'image, le point $(1,1)$ le point en haut à droite (on choisira n'importe quelle règle d'arrondi pour obtenir des coordonnées entières, cela n'a pas d'importance).

10. Compléter la fonction `draw2D` pour qu'elle effectue ce travail, et tester les différents générateurs. Est-ce satisfaisant ?

3.5 Et les séries de trois ?

Malheureusement, les faiblesses des générateurs peuvent passer inaperçues pendant très longtemps. Et c'est ce qui est arrivé avec l'un des générateurs étudiés ici, qui a été utilisé pendant plusieurs décennies de façon quasi-universelle avant qu'on ne lui découvre un énorme défaut. Nous allons essayer à présent de le mettre en évidence.

1. Uniforme n'est pas un terme correct, car il n'y a qu'un nombre fini de réels dans cet intervalle.

Pour ce faire, on va générer 60000 flottants, les prendre trois par trois, et les considérer comme autant de points (x,y,z) dans l'espace. En théorie, si le générateur est de bonne qualité, ces points devraient être répartis, sans structure visible, dans un cube de côté 1.

Comme il est délicat de tracer des images en trois dimensions, nous allons projeter ce nuage dans le plan, en considérant les points du plan $(0.6x+0.4(1-y),z)$.

11. Que devrait-on logiquement obtenir si l'on trace le nuage de points ? La répartition théorique tend-elle vers un résultat homogène ou non ?

12. Compléter la fonction `draw3D` pour qu'elle trace le nuage de points dans le plan.

13. Utiliser cette fonction avec chacun des générateurs. Lequel est le coupable ?

4 Plus loin avec DieHard

Ce regrettable accident a montré que tester les générateurs n'était pas une question simple, mais qu'elle était cruciale. Afin de mettre à l'épreuve de façon plus exhaustive les qualités d'un générateur pseudo-aléatoire, G. Marsaglia a proposé en 1995 un ensemble de tests statistiques appelée DieHard [2]. Un générateur pseudo-aléatoire de bonne qualité devrait passer avec succès tous les tests.

Réussir un test est toutefois une notion assez complexe, dans la mesure où, encore une fois, il y a une probabilité non nulle qu'un générateur réellement aléatoire propose une séquence de n zéros consécutifs. Les statistiques derrière étant relativement complexes, on n'en touchera dans cette séance que la surface.

Précisons tout de suite que ces tests sont conçus pour poser problème à des générateurs pseudo-aléatoires de bonne qualité. Il n'est donc pas étonnant si nos générateurs de Lehmer (ou le Mersenne Twister simplifié) ne donnent pas des résultats très probants sur ces tests !

4.1 OPSO

Dans le test OPSO, on génère $2^{21}+1$ entiers sur au moins 31 bits. On extrait les 10 bits de poids faible de chacun de ces entiers, ce qui donne $2^{21}+1$ valeurs, que l'on considère comme des « lettres ». Il y a donc 2^{10} lettres possibles. On considère ensuite les 2^{21} paires consécutives de lettres (avec recouvrement, donc), ce qui nous donne 2^{21} « mots » de deux lettres.

Il y a, en théorie, 2^{20} mots de deux lettres différents possibles. On compte alors, parmi les 2^{21} mots générés, combien de ces mots n'apparaissent pas. Ce nombre ne doit pas être trop grand (le générateur aléatoire doit en théorie les générer tous si on produit suffisamment de « lettres ») ni trop petit. La théorie indique que le nombre de mots absent devrait suivre une loi normale de moyenne 141909 et d'écart-type 290.

2. Un ensemble plus complet et plus difficile de tests, appelé DieHarder, l'a désormais remplacé.

14. Écrire une fonction `test_OPSO` effectuant la tâche décrite et affichant le nombre de mots absents. Vérifier dans quelle mesure ce nombre de mots ne s'écarte pas trop de 141909 (quelques écarts-types au plus) de la valeur théorique pour chacun des générateurs aléatoires.

Il nous faut aller un peu plus loin qu'un vague contrôle de ce genre. On va donc effectuer au total 22 fois ce dénombrement, avec **la même série** de $2^{21}+1$ nombres aléatoires, mais en prenant successivement les bits i à $i+9$ pour i variant de 0 à 21.

On obtient donc 21 valeurs qui devraient toutes être proches de 141909 et avoir une distribution normale.

Pour le vérifier, trier les valeurs par ordre croissant (pas besoin de le faire faire par la machine) et reporter celles-ci sur le papier millimétré spécial. Si la distribution est compatible avec une loi normale, elles devraient apparaître alignées, et si c'est le cas, la droite obtenue vous permet d'obtenir une estimation de la moyenne et de l'écart-type.

15. Vérifier le comportement d'un ou deux générateurs parmi ceux proposés (partagez-vous la tâche entre vous !)

Note : il existe deux autres tests similaires dans la série DieHard :

- OQSO, où l'on construit des mots de quatres lettres, chaque lettre utilisant cinq bits d'un nombre pseudo-aléatoire généré ;
- ADN, où l'on construit des mots de dix lettres, chaque lettre utilisant deux bits d'un nombre pseudo-aléatoire généré (le nom venant du fait que l'on a quatre lettres dans notre alphabet, comme dans le cas de l'ADN).

4.2 Test du parking

Dans ce second test, on commencera par générer 528000 flottants aléatoires dans $[0,1[$. On va tenter de « garer » des « voitures » dans un parking. Pour ce faire, on considère les 24000 premiers flottants comme autant de couples (x,y) correspondant à un emplacement auquel on essaie de garer un véhicule.

Chaque véhicule est un carré de côté 0.01 centré en (x,y), et orienté selon les axes du repère. Pour chaque véhicule, pris dans l'ordre, s'il est possible de le placer sans qu'il y ait recouvrement avec un autre véhicule, on le fait, sinon la tentative échoue et on passe simplement au véhicule suivant.

La question est de savoir combien on va parvenir à garer de véhicules dans le parking. En théorie, après 12000 tentatives, ce nombre devrait suivre approximativement [3] une loi normale de moyenne 3523 d'écart-type 21.9.

16. Proposer une fonction `test_parking` déterminant le nombre de véhicules que l'on parvient à garer avec 12000 tentatives, effectuées avec les 24000 premiers flottants générés, et regarder avec les générateurs proposés si on est effectivement proche de la moyenne attendue.

17. Modifier la fonction pour qu'elle produise une image de taille 500×500 où un point correspond à la position où on a pu garer un véhicule.

Note : la génération de positions aléatoire dans un carré telles qu'il n'y ait pas deux points trop proches est un problème algorithmique très important avec d'innombrables applications pratiques. Pour citer un exemple concret parmi d'autres, pour modéliser une forêt naturelle, par exemple, il faut choisir des positions pour les arbres qui ne doivent pas être trop proches, mais n'ont aucune raison d'être régulières.

18. Modifier la fonction pour qu'elle effectue le traitement en prenant en compte les 24000 premiers flottants, puis les 24000 suivants, et ainsi de suite, 22 fois [4], et retourne les 22 nombres de véhicules placés à chaque tentative.

19. Vérifier, par lecture graphique sur le papier fourni, en choisissant un des générateurs, s'il passe le test avec succès.

4.3 Test du rang

Dans ce test, on va considérer des matrices de taille 31×31 dont les éléments sont dans l'ensemble $\{0,1\}$, et on va calculer leur rang dans $\mathbb{Z}/2\mathbb{Z}$.

Ce rang peut être déterminé par la méthode du pivot de Gauss. Le fait de se trouver dans $\mathbb{Z}/2\mathbb{Z}$ simplifie cependant quelque peu les choses :

- il n'y aura pas besoin d'effectuer de dilatations (multiplications) ;
- les additions et soustractions sont simplement effectuées avec un « ou exclusif » logique, c'est-à-dire avec l'opérateur `^`.

Pour simplifier, on représente une matrice comme un tableau de 31 entiers sur 31 bits, chaque entier correspondant à une ligne de la matrice, chaque bit à un élément. Par simplicité, l'algorithme du pivot se fera de la droite vers la gauche, des poids faibles vers les poids forts. On procède de la sorte :

- on initialise un compteur qui contiendra le rang à 0
- 31 fois, on effectue la démarche suivante :
  - si tous les entiers sont pairs, on les divise tous par deux, et on passe à l'itération suivante
  - sinon, on incrémente le compteur, puis on choisit un des entiers impair, que l'on note p ; on l'élimine de la liste des entiers et on effectue un « ou exclusif » entre cet entier p et tous les entiers impairs (ce sont les transvexions du pivot de Gauss), et on divise tous les entiers (dorénavant tous pairs) par deux.

On remarquera qu'à tout instant, le nombre d'entiers restant à considérer est égal à 31 moins le contenu du compteur. On pourra s'arranger, en procédant avec des échanges, pour que ces entiers soient toujours ceux qui se trouvent au début du tableau.

3. Le résultat est expérimental, on ne dispose pas d'une théorie mathématique pour l'affirmer.

4. Dans le test original, ce n'est pas 22, mais cela permet d'utiliser le même papier millimétré que pour le test précédent, ce qui simplifie les choses.

20. Compléter la fonction `rang`.

Pour le test, nous allons donc générer un tableau de 1240000 entiers sur 31 bits (si on génère des entiers sur 32 bits, ce n'est pas bien grave, car notre fonction calculant le rang, telle que décrite précédemment, va simplement ignorer le bit de poids fort), que l'on va considérer comme 40000 matrices de taille 31×31, dont on va déterminer le rang.

21. Compléter la fonction `test_rang` pour qu'elle calcule le rang de chacune des matrices, et qu'elle détermine le nombre de matrices de rang 31, le nombre de matrices de rang 30, le nombre de matrices de rang 29, et le nombre de matrices de rang inférieur ou égal à 28.

22. Afficher ces nombres pour chacun des générateurs aléatoires. Les valeurs théoriques devraient être proches respectivement de 11551.5, 23103, 5134 et 211.5.

En théorie, il conviendrait d'évaluer la quantité $1 - e^{-\sum \frac{e_i - r_i}{2r_i}}$ pour tenter de conclure :

- si la valeur est proche de 1, on est « anormalement loin » de la répartition théorique ,
- si la valeur est proche de 0, on est au contraire « anormalement près » de la distribution théorique, ce qui n'est pas une bonne chose non plus.

Cela étant dit, accidentellement, il est possible d'être trop près ou trop loin de la distribution théorique. On fera donc en principe plusieurs essais, et on regardera si ces événements sont bien « extraordinaires » (autrement dit n'arrivent pas trop souvent, ce qui là aussi est difficile à quantifier.