Comprendre les générateurs de nombres pseudo-aléatoires

Yann Guidon

EN DEUX MOTS **Au détour d'un article sur les CRC publié en décembre 2005 (GLMF n°78), j'avais bricolé un petit morceau de code générant une séquence cyclique de quatre milliards de bits. La motivation était de se passer de /dev/urandom (pas très rapide). Mais pour l'utilisation qui en était faite (tests statistiques pseudo-aléatoires), ce code était encore trop lent et la séquence bien trop courte. Nous allons maintenant corriger cela et en profiter pour découvrir les idées (assez curieuses) qui se cachent derrière ces algorithmes. Au menu de cet article : des nombres qui réagissent entre eux, des polynômes dans GF(2), des batteries de LFSR et quelques techniques d'optimisation « classiques » (sans écrire une seule ligne d'assembleur).**

Cet article fait écho à celui sur les CRC, où j'avais laissé de côté les notions qui n'étaient pas directement liées à la programmation, afin de passer directement à l'optimisation. Les lecteurs intéressés par la théorie étaient encouragés à consulter des références externes mais beaucoup s'attendaient à trouver les explications fondamentales dans l'article. Il aurait été vraiment beaucoup trop long, comme on peut le voir dans le présent opus qui n'aborde pourtant que le strict minimum des idées à comprendre (et à connecter entre elles). Paradoxalement, cela rend la compréhension plus difficile car l'article doit rester compact, au risque de ne pas tout dire. Il y a tellement à dire mais il faut faire des choix, j'ai fait celui d'exposer les concepts et le cheminement qui déboulent sur les algorithmes. Si la théorie pure vous intéresse, les liens en fin d'article sont là pour ça : l'objet de cette série n'est pas d'expliquer toute la théorie, mais de la comprendre assez pour l'appliquer le plus efficacement possible.

Nous partons maintenant dans un article parallèle, car les CRC et les générateurs de nombres pseudo-aléatoires sont presque la même chose sur de nombreux points. Et cette fois-ci, nous allons faire, pas à pas, la jonction entre la théorie mathématique et l'utilisation pratique. Les lecteurs qui arrivent à accrocher tous les wagons entre eux pourront peut-être améliorer au passage leur compréhension des CRC, mais il n'y a pas de honte à sauter une étape pour la relire plus tard avec un regard nouveau. Je ne dis pas que c'est évident, mais si j'ai réussi à comprendre et expliquer tout cela malgré un 4 en maths à mon BAC S, d'autres peuvent certainement me suivre. Alors prenez votre après-midi (il me faut déjà une heure pour me relire), la trinité indispensable (papier, crayon et aspirine) et commencez par oublier les ennuyeux cours au lycée : la Théorie des Nombres est réellement un domaine surprenant et fascinant, sinon il n'y aurait pas tant de personnes dédiant leur temps au nombre Pi ou aux nombres de Mersenne...

D'où viennent les générateurs de nombres pseudo-aléatoires ?

Nous allons maintenant parler de suites de nombres *qui ont l'air aléatoires* pour un observateur humain consentant mais non averti (pensez au test de Turing). Puisqu'ils sont créés par ordinateur, il est possible de prévoir les prochains nombres à partir d'un seul élément mais la relation entre eux ne saute pas naturellement aux yeux. Si une suite *réellement* aléatoire est nécessaire, on peut employer des dispositifs physiques mesurant des phénomènes réels et chaotiques (le bruit brownien de l'air ou dans un semi-conducteur), mais seules des applications cryptographiques ou de sécurité en ont vraiment besoin. Pour tester des algorithmes, nous n'avons pas de contraintes d'ordre cryptographique donc un petit bout de code suffit. Pour garantir son efficacité, il faut quand même faire appel à *un peu de Mathématiques*.

Les premiers indices qui vont nous conduire à l'algorithme final sont tellement anodins qu'on n'y prête aucune attention, même avec une calculatrice. Commençons avec une banale division entre deux nombres entiers :

```
1 / 7 = 0,142857142857142857142857142857...
```

Nous voyons que la séquence 142857 se répète à l'infini. Pour obtenir les décimales les unes après les autres, il suffit d'utiliser ce bon vieil algorithme de division appris il y a si longtemps à l'école :

```
 1      / 7
-0|||||  0
 10|||||
 -7|||||  1
  30||||
 -28||||  4
   20|||
  -14|||  2
    60||
   -56||  8
     40|
    -35|  5
      50
     -49  7
```

```
       1
      ....

int i=1;   /* valeur initiale de la suite, appelée "seed" */
while (i) {
  j=0;
  while (i>=7)  /* division par soustractions */
    j++, i-=7;
  printf("%d ", j);
  i*=10;     /* on ajoute un zéro */
}
```

Pourquoi obtenons-nous une telle suite de chiffres ? Prenons un contre-exemple : si 1 avait été divisé par 2 ou 5 (ou leurs multiples comme 4, 8, 25...), un 0 serait apparu dans la suite à un moment ou un autre et n'aurait plus bougé. Par exemple : 1/125=0,00800... A l'inverse, 3 et ses multiples (tels 6 et 9) auraient donné une suite monotone de 3, de 6 ou de 1. Par exemple : 1/6=0,1666666...

Dans ces exemples, j'ai choisi 1 comme nombre à diviser mais d'autres nombres entiers auraient convenu. En effet, le nombre à diviser risque de se reproduire dans le résultat, ce n'est pas lui qui contrôle si une séquence va apparaître ou pas. En fait, c'est la base du calcul (ici notre base décimale, donc 10) qui a des interactions particulières (comme s'il entrait en résonance) avec le diviseur. Et il faut des conditions particulières pour qu'une longue suite cyclique apparaisse.

La première clé est que 10 est multiple de 5 et 2 mais pas de 7 ou 3, **10 et 7 sont donc premiers entre eux**. Cela garantit que nous obtenons des nombres rationnels et empêche la suite de décimales de s'arrêter.

Mais ce n'est pas tout : 3 n'a pas de facteur commun avec 10 et pourtant nous n'obtenons pas de séquence cyclique, ou plus exactement : le cycle ne contient qu'un seul chiffre. Il se trouve que certains nombres réagissent mieux entre eux que d'autres (un peu comme des éléments chimiques).

NOTE

Ces premières observations nous permettent de dégager trois classes de nombres, en fonction du comportement des décimales d'une division entre deux entiers :

- Les rationnels avec un nombre fini de décimales non nulles. C'est le cas lorsque le diviseur est composé des multiples de la base (2 et 5 dans notre cas).
- Les rationnels avec un nombre infini de décimales, mais avec des cycles courts (une décimale ou quelques-unes). Cela se produit lorsque le diviseur est multiple d'un nombre (ou plus) qui ne divise(nt) pas la base. Par exemple, 6 est multiple de 2 et 3, et la suite des décimales de 1/6 est une infinité de 6 en base 10.
- Les rationnels qui ont un cycle « long » sortent du lot, tel 1/7 en base 10.

Ce sont ces derniers nombres qui nous intéressent et nous allons chercher à les différencier de la deuxième classe (les différencier de la première classe est très facile).

Cette idée restera la même tout au long de cet article, quelle que soit la forme, mais notre exemple de 1/7 n'a pas assez de chiffres dans le cycle, et tous les chiffres de la base ne sont pas utilisés.

Une forme de calcul plus adaptée

Afin d'obtenir une suite non biaisée et statistiquement monotone, nous cherchons à obtenir une permutation de la suite de tous les nombres de 1 à N-1 (où N est la base). 0 ne peut pas être utilisé car cela bloquerait la génération de la suite (on tomberait dans la première classe de rationnels). Garantir que toutes les valeurs de sortie seront utilisées (et juste une fois par cycle) évite d'avoir à surdimensionner l'algorithme. Si cette condition est remplie, on dit que ce générateur est **de période maximale** (on évite autant que possible d'utiliser les autres).

En utilisant d'autres bases que la base décimale, d'autres diviseurs et des formes de calcul un peu plus adaptées, on peut concevoir des suites de nombres entiers qui ne reboucleront qu'après avoir fourni tous les nombres dans un ordre qui semble aléatoire. Par exemple, les « générateurs linéaires congruentiels » travaillent dans des bases non décimales en faisant appel à l'opérateur *modulo* dans la fonction de récurrence suivante :

```
N = ((N * a) + b ) % c;
```

Voici quelques valeurs de a, b, c pour des générateurs de période maximale (extraites du livre de Bruce Schneier, qui les a empruntées au livre *Numerical Recipes in C*).

```
a        b        c    nombre de bits nécessaires
  106     1283     6075      20
  171    11213    53125      24
  421    54773   259200      27
  741    66037   312500      28
 1366   150889   714025      30
 2416   374441  1771875      31

/* petit programme de test, à lancer avec
         ./a.out |sort|less
   pour vérifier que toutes les valeurs
   sont bien visitées dans un cycle : */

#include <stdio.h>
#define a 106
#define b 1283
#define c 6075

int main() {
  unsigned long int i, N=0; /* "seed" */

  for (i=0; i<c; i++) {
    N = ((N * a) + b ) % c;
    printf("%04d\n", N);
  }
```

```
  return 0;
}
```

Puisque tous les nombres sont modulo c, c détermine la quantité de nombres générés dans un seul cycle. Juste pour vérifier que ces nombres ne sont pas issus du hasard, factorisons les nombres a, b et c de la première ligne :

```
106 = 2 * 53
1283 est premier
6075 = 5^2 * 3^5
```

Ils ne sont pas tous premiers, mais ils ne partagent pas de facteur premier entre eux. Nous sommes déjà sûrs que le générateur ne va pas s'arrêter sur un zéro, mais cela ne suffit pas. Il manque le moyen de distinguer entre la deuxième et la troisième catégorie de nombres, bien que nous pouvons déjà tenter une recherche exhaustive avec un petit programme informatique.

On s'approche lentement de l'objectif, mais ce n'est toujours pas très pratique, la multiplication et la division (incluse implicitement dans l'opérateur modulo) étant relativement lourdes. De plus, des bits nécessaires au calcul ne sont pas utilisés (à cause de la multiplication qui augmente la taille du nombre avant le modulo) et l'opération d'addition pourrait être enlevée si on renonce au nombre 0 en sortie.

NOTE

Dans cette partie de l'explication, nous avons identifié la troisième catégorie de nombres comme ceux donnant une « période maximale ». Ce comportement se retrouve aussi dans d'autres calculs qui n'utilisent pas directement la division, ce qui laisse penser qu'il existe une théorie plus générale que celle que nous avons tenté d'ébaucher avec une calculatrice.

Jouons avec les bases

L'algorithme que nous allons voir plus loin, appelé « LFSR » (pour *Linear Feedback Shift Register* ou en français : Registre à Décalage à Rétroaction Linéaire), travaille en base 2 sans utiliser d'opération arithmétique classique. Comme indiqué dans l'article précédent, les LFSR peuvent être vus comme des générateurs linéaires congruentiels simplifiés à la base 2, ou à l'inverse, les générateurs congruentiels sont des LFSR généralisés dans d'autres bases. L'avantage est immense car pour un LFSR, la division ou la multiplication sont réduites à un décalage d'un bit.

En réalité, le principe n'est pas *exactement* le même, mais on retrouve les mêmes ingrédients à base de primalité et de ce dernier élément qui nous manque. Justement, 7 est un Premier que nous venons d'utiliser. Prenons-le comme base, à la place de 10, et jouons avec les puissances des nombres inférieurs à 7, en rebouclant automatiquement en cas de dépassement :

```
2^1 = 2  mod 7 = 2      3^1 = 3   mod 7 = 3
2^2 = 4  mod 7 = 4      3^2 = 9   mod 7 = 2
2^3 = 8  mod 7 = 1      3^3 = 27  mod 7 = 6
2^4 = 16 mod 7 = 2      3^4 = 81  mod 7 = 4
2^5 = 32 mod 7 = 4      3^5 = 243 mod 7 = 5
2^6 = 64 mod 7 = 1      3^6 = 729 mod 7 = 1

4^1 = 4    mod 7 = 4    5^1 = 5     mod 7 = 5
4^2 = 16   mod 7 = 2    5^2 = 25    mod 7 = 4
4^3 = 64   mod 7 = 1    5^3 = 125   mod 7 = 6
4^4 = 256  mod 7 = 4    5^4 = 625   mod 7 = 2
4^5 = 1024 mod 7 = 2    5^5 = 3125  mod 7 = 3
4^6 = 4096 mod 7 = 1    5^6 = 15625 mod 7 = 1

6^1 = 6     mod 7 = 6
6^2 = 36    mod 7 = 1
6^3 = 216   mod 7 = 6
6^4 = 1296  mod 7 = 1
6^5 = 7776  mod 7 = 6
6^6 = 46656 mod 7 = 1
```

On voit que les puissances successives de 3 et de 5, modulo 7, donnent des suites de nombres sans *ordre apparent*, contrairement à 2 et ses multiples. En plus, ces deux suites contiennent **tous** les nombres (sauf zéro), et dans un ordre différent ! Comme les puissances de 3 et de 5 permettent de « générer » tous les nombres dans la base 7, les mathématiciens disent que 3 et 5 sont des **générateurs modulo 7**, ou bien que 3 et 5 sont **primitifs** par rapport à 7. Ce qui est synonyme d'un générateur à période maximale !

Une autre chose intéressante à remarquer est qu'il ne peut y avoir que 6 puissances (non nulles) dans la base 7 (ensuite, ça reboucle). Or la taille du cycle dans les décimales de 1/7 en base 10 est justement de 6 chiffres. Si vous n'êtes toujours pas convaincu du lien intime entre tous ces éléments, regardez bien les *restes* (mis en évidence) à chaque étape de la division manuelle : `1, 3, 2, 6, 4, 5`. C'est exactement les chiffres (en partant du bas) de la séquence générée par `(5^n)%7` ! Comme le résultat est en base 10, il manquait bien 10-7=3 chiffres dans la séquence. Mais en fait, les restes de la division sont tous modulo 7, donc en base 7.

NOTE

A cet endroit de la lecture, je vous invite à recommencer une ou deux fois à partir du début, avec la connaissance des différences entre les trois classes de nombre (qui ont maintenant des propriétés plus claires). Ce début introduit des notions qui seront étendues par la suite dans d'autres formes. Il faut les avoir bien assimilées pour pouvoir comprendre les transformations qu'elles vont subir.

Ici, nous parlons de puissances mais ce ne sont qu'une succession de multiplications. Et pour générer la suite de chiffres, nous effectuons une multiplication à chaque étape, sautant ainsi d'une puissance à la suivante, en générant un nouveau nombre de la permutation.

```
1 * 5 = 5,   5 mod 7 = 5
5 * 5 = 25, 25 mod 7 = 4
4 * 5 = 20, 20 mod 7 = 6
6 * 5 = 30, 30 mod 7 = 2
2 * 5 = 10, 10 mod 7 = 3
3 * 5 = 15, 15 mod 7 = 1
```

Clairement, la séquence est stable : elle se répétera toujours dans le même ordre quelle que soit la valeur de départ (et elle n'avancera pas si on met 0). Nous voulons obtenir le même genre de résultat mais de manière ultra-simplifiée. Pour cela, la théorie doit être un peu plus alambiquée...

Les nombres premiers sont bien indispensables pour générer des bonnes séquences pseudo-aléatoires. Car sans vous le dire, nous sommes entrés dans le monde des **corps de Galois** : c'est le fait d'effectuer toutes les opérations *modulo un nombre premier* ou une de ses puissances (avec des règles précises). Dans notre premier exemple en base 7, nous étions dans `GF(7)` (les initiales de « *Galois Field* » en anglais), qui s'écrit aussi « Z/7 ».

ATTENTION

Dans les corps de Galois, les nombres premiers (qui sont la brique de base de tous les autres nombres dans Z) sont remplacés par les nombres générateurs. Par exemple, on peut remarquer que 2, qui est un nombre premier, n'est pas générateur dans Z/7. Pourtant, les générateurs dans Z/11 sont 2, 6, 7 et 8 : en changeant de base, nous changeons de générateurs.

PRÉCISIONS À CARACTÈRE NON MATHÉMATIQUE

Nous ne parlons pas ici de corps de Gallois. GLMF ne signifie pas GNU/Linux Morgue France et il n'existe pas de tueur en série s'attaquant spécifiquement aux natifs du Pays de Galle... /pataper

Évariste Galois (avec un seul 'l'), fougueux républicain et prodige mathématique français, est mort à 21 ans en 1832 dans un duel stupide, non sans avoir préalablement révolutionné les mathématiques avec un manuscrit de quelques pages... Presque deux siècles plus tard, ses découvertes sont au cœur de nombreux systèmes numériques, en particulier les codes correcteurs d'erreurs indispensables pour stocker ou transmettre des données de manière fiable. En 2003, le magazine *Pour La Science* a consacré son numéro spécial *Les génies de la science n°14* à ce garnement génial.

Maintenant, nous avons un cadre et des outils mathématiques permettant d'explorer les séquences cycliques. Je vous conseille, après une relecture complète de cette première partie, de chercher sur Internet des références et tutoriaux car nous entrons maintenant dans un monde très *décalé*...

Le travail en base 2

En base 2, on ne peut avoir que 0 (l'élément neutre de l'addition) et 1 (l'élément neutre pour la multiplication). On ne peut donc rien faire directement dans Z/2, ce qui n'est pas très intéressant. Par contre, nous pouvons utiliser des polynômes puisqu'ils utilisent des multiples des puissances. Nous sommes déjà habitués à cette représentation lorsqu'il faut convertir de la base décimale au binaire ou vice versa :

$$11010010_2 = 2^7 + 2^6 + 2^4 + 2^1 = 128 + 64 + 16 + 2 = 210_{10}$$

Mais nous ne sommes plus dans l'arithmétique classique. En remplaçant 2 par *x*, les polynômes ont *remplacé* les nombres. Ce qui explique pourquoi, par facilité d'écriture, les polynômes de CRC ou de LFSR sont souvent représentés par des nombres. Nous voilà plongés dans $Z/2^n$ (ou `GF(2n)` en notation anglaise), avec n égal au degré du polynôme. La table 1 permet de s'y retrouver plus facilement en fournissant une équivalence entre $Z/2^3$ et les bases plus connues.

Base 10	Base 2	Z/23
0	000	0
1	001	1
2	010	x
3	011	x + 1
4	100	x2
5	101	x2 + 1
6	110	x2 + x
7	111	x2 + x + 1

Table 1 : Représentation des nombres 0 à 7 dans trois systèmes de codage

Par exemple, voici l'écriture de 210 dans $Z/2^8$:

$$210_{10} = x^7 + x^6 + x^4 + x$$

Puisque les nombres sont remplacés par des polynômes dans un *corps fini* (c'est-à-dire en vase clos, tous les résultats font partie du même ensemble de nombres du départ), les opérations arithmétiques de base sont « un peu changées ».

Par exemple, l'addition de deux polynômes s'effectue toujours en additionnant un à un les coefficients de même rang (comme avec des polynômes classiques). Mais ces facteurs sont dans `GF(2)`, donc modulo 2. Cela se résume, en informatique ou en électronique, à un simple *XOR* !

Curieusement, l'addition est ainsi confondue avec la soustraction, ce qui brouille toute notion d'ordre (infériorité ou supériorité) entre les nombres. Je vais donc utiliser le symbole ± pour insister sur ce fait. Le résultat peut paraître délirant, mais nous ne sommes plus en arithmétique classique :

$$210_{10} = x^7 + x^6 + x^4 + x$$
$$143_{10} = x^7 + x^3 + x^2 + x + 1$$

$$x^7 + x^6 + x^4 + x$$
$$\pm\ x^7 + x^3 + x^2 + x + 1$$
$$= x^6 + x^4 + x^3 + x^2 + 1 = 93_{10}$$

Pour faire ce calcul à la main, on recopie tous les facteurs dans la somme, et s'il y en a deux du même rang, on élimine la paire. Ou on peut passer directement par une représentation binaire des coefficients de ces facteurs, puisque c'est exactement la même chose :

$$210_{10} = 11010010$$
$$143_{10} = 10001111$$
$$10001111 \text{ xor } 11010010 = 01011101 = 93_{10}$$

Pour la multiplication, c'est le même principe qu'avec les nombres ou les polynômes classiques, c'est une répétition d'additions et de décalages. Bien sûr, ici, l'addition est un simple XOR et il n'y a pas de retenue à propager. Pour la division, c'est *presque pareil* aussi. Au détail près que, comme nous sommes dans un corps fini, la taille du polynôme résultant doit rester la même, et le résultat doit être modulo... un polynôme.

La foire aux polynômes

Ne plus se soucier de la propagation des retenues lors des calculs a un impact radical sur la vitesse des applications. Selon la taille des données, l'accélération peut être d'un facteur 5 rien que pour l'addition. Pour la multiplication ou la division, cela n'a plus rien à voir ! En plus, le XOR est commutatif, distributif, etc. ce qui permet de paralléliser encore plus les opérations et d'optimiser les calculs par des réorganisations judicieuses. Mais en utilisant cette arithmétique si particulière, ne risque-t-on pas de perdre les propriétés si intéressantes des nombres premiers, capables de générer des séquences de permutations ?

Cela peut sembler étrange mais non : le même principe existe aussi avec les polynômes ! Il en existe qui ne peuvent être décomposés en facteurs, ce sont des *polynômes irréductibles* (l'équivalent d'un nombre Premier mais pour les polynômes). Parmi ceux-ci, certains sont *primitifs* (dans un corps de Galois choisi) et servent aussi à « générer » tous les autres polynômes de ce corps. Tout comme dans `GF(7)` où les calculs étaient modulo 7, les calculs dans `GF(2n)` sont modulo un polynôme primitif choisi, qui va contrôler l'ordre de génération des nombres-polynômes. Cette permutation s'effectue avec la multiplication ou la division, comme dans `GF(7)` : on obtient le prochain nombre de la suite avec une itération tout aussi simple.

NOTE

La multiplication et la division sont quasiment la même chose en ce qui nous concerne : un polynôme primitif peut être réfléchi (l'ordre de ses coefficients est inversé) et cela donne un autre polynôme primitif. Par exemple, `x4 + x + 1` est générateur dans `GF(24)`, donc `x4 + x3 + 1` aussi.

De même, dans `GF(7)`, la série de nombres générés par 5 est l'inverse de celle générée par 3. Dans ce corps, multiplier par 3 est la même chose que diviser par 5, et vice versa : 3 est donc l'inverse de 5. Cela semble fou mais pas impossible puisque nous ne sommes plus en arithmétique classique.

Ainsi, tant que les opérations restent cohérentes, peu importe si on décale à gauche ou à droite, si on multiplie ou si on divise, on obtient toujours *un* nombre suivant (peu nous importe lequel) dans la suite pseudo-aléatoire ; et nous allons bien sûr prendre la solution qui nous arrange le plus.

On ne peut pas utiliser les mêmes « valeurs » primitives que dans `GF(n)`. Un nombre premier dans Z (nos entiers à nous) ne peut pas être traduit directement dans $Z/2^n$ pour créer un polynôme primitif. Heureusement, il n'est pas nécessaire de chercher loin, car il existe une famille de polynômes bien connue pour ses propriétés intéressantes. « On sait que » x^n+x+1 est primitif pour :

```
n = 1, 3, 4, 6, 9, 15, 22, 28, 30, 46, 60, 63, 127, 153, 172...
```

Dans cette liste, on trouve quelques valeurs voisines de 2^n, donc très pratiques en informatique. En plus, elles ont un lien avec les nombres de Mersenne (encore des nombres premiers remarquables, si cela peut encore exciter votre curiosité). Toutes les puissances de 2 ne sont pas présentes mais cela nous suffit pour commencer.

L'autre avantage est que c'est un « polynôme clairsemé », c'est-à-dire avec très peu de termes, et en fait nous avons ici le *minimum* de termes pour un polynôme primitif. En résumé : c'est pratique, c'est efficace et on ne peut pas faire plus simple. Il existe bien d'autres polynômes, mais ils n'ont pas tous ces avantages, c'est pourquoi on rencontre ce « trinôme » très souvent.

Ma première multiplication polynômiale modulaire...

Voici maintenant un exemple qui va mettre en scène tous ces éléments étranges et contre-intuitifs, en effectuant quasiment la même chose que pour l'exemple de suite dans `GF(7)`. C'est l'illustration qu'une multiplication polynômiale modulaire dans $Z/2^n$ permet de générer une suite de 2^n-1 nombres, qui est *une* permutation des nombres de 1 à 2^n-1.

- Nous allons nous placer dans $Z/2^4$, c'est à dire travailler avec 4 « bits ».
- Le polynôme générateur est le trinôme présenté précédemment, avec n=4, donc : `p(x) = x4 + x + 1`
- Nous allons multiplier notre nombre-polynôme par `x`. Celui-ci contrôle l'ordre de permutation et n'importe quel autre nombre-polynôme (autre que 0 et 1) aurait convenu, mais la quantité de calculs aurait augmenté sans donner de résultat qualitativement différent (n'importe quelle permutation nous convient ici).

Je précise ici les règles de la multiplication dans $Z/2^4$:

- Comme en arithmétique classique ou en arithmétique polynômiale, la multiplication est à base de décalages et d'additions successives. Multiplier un nombre de `a` bits par un nombre de `b` bits donnera un produit (temporaire) sur `a+b` bits. Comme nous multiplions par x, nous obtenons 4+1=5 bits effectifs :

$$\begin{array}{r} ax^3 + bx^2 + cx + d \\ \times \qquad\qquad\quad x \\ \hline = ax^4 + bx^3 + cx^2 + dx \end{array}$$

- Puisque nous travaillons dans un *corps fini*, tous les nombres sont sur `n` bits. Il faut donc réduire la taille du produit par une opération de modulo, qui soustrait autant de fois que nécessaire le polynôme générateur au résultat intermédiaire précédent. Dans notre cas, si `a=1`, on doit « soustraire » `p(x)` pour que le résultat définitif tienne dans 4 bits :

$$\begin{array}{r} ax^4 + bx^3 + cx^2 + dx \\ \pm\ ax^4 + \qquad\quad ax + a \\ \hline = \qquad bx^3 + cx^2 + adx + a \end{array}$$

Nous avons obtenu la formule de récurrence qui permet de générer la table de multiplication par `x`, qui est aussi notre suite de nombres pseudo-aléatoires. Nous pouvons exprimer ce calcul sous forme informatique très facilement :

- La multiplication, comme promis, est un décalage. Le sens importe peu, comme indiqué plus haut, tant que c'est cohérent et toujours le même :-) Par contre, le coefficient de plus haut degré se retrouve au plus bas degré, laissé libre : nous avons en fait affaire à une rotation.
- Le modulo est effectué par un XOR contrôlé par `a`. Si nous avions multiplié par un polynôme plus complexe, il aurait fallu plus d'opérations mais dans notre cas, tout se simplifie. Le terme `adx` signifie qu'il faut juste effectuer un XOR entre `a` et `d`.

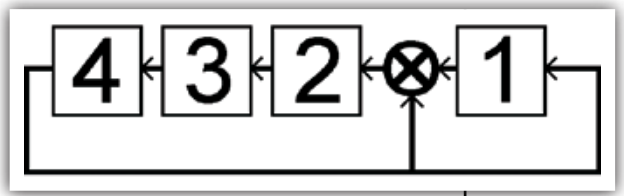

Fig. 1 : Structure d'un LFSR 4 bits en configuration de Galois

NOTE

Un décalage et un XOR : je vous avais bien dit que ce serait simple :-)

Tout ce qui suit va exploiter les notions précédentes. J'espère que c'était assez clair car maintenant, nous passons (enfin) à la programmation. D'autres idées importantes seront encore introduites pour faciliter la réalisation, mais la base fondamentale ne change pas : nous obtenons une suite de nombres pseudo-aléatoires en effectuant une multiplication dans un domaine des mathématiques qui rend l'opération très simple (même si la compréhension n'a rien d'évident).

Mon premier programme de LFSR

La représentation compacte de `x4+x+1` est `(4,1,Ø)` :

- Le premier nombre indique combien de bits compte le registre à décalage.
- Le zéro est toujours présent mais ignoré en pratique, son rôle mathématique est de reboucler le registre sur lui-même. En l'absence du terme `1` dans le polynôme générateur, il est impossible de réinjecter le terme `a` à l'autre bout du registre.
- Les autres termes sont appelés la *séquence de dérivation*. Ils indiquent les numéros des bits sur lesquels il faut effectuer un XOR.

Le codage d'une étape de LFSR est facile. En voici une version naïve :

```
signed int i, N = 1; /* seed */

for (i=0; i<16; i++) {
  N <<= 1; /* la multiplication par x */
  N ^= (N >> 3) & 2; /* calcule le terme adx */
  N ^= (N >> 4) & 1; /* "calcule" le terme a */
  N &= 15; /* supprime le terme ax^4*/
  printf("%02d\n", N);
}
```

L'exécution de ce code donne le résultat suivant :

```
02   0010
04    0100
08     1000
03      0011
06       0110
12        1100
11         1011
05          0101
10           1010
07            0111
14             1110
15              1111
13               1101
09                1001
01                 0001
02                  0010
```

A partir de là, on peut construire presque n'importe quel LFSR en connaissant sa séquence de dérivation (donc son polynôme). On peut trouver une page entière de telles séquences dans le livre *Cryptographie appliquée* de Bruce Schneier, par exemple.

La suite de nombres que nous venons de créer n'a pas l'air très aléatoire, mais un LFSR est normalement utilisé pour générer un seul bit par cycle. La séquence de 2^N-1 bits générée par un LFSR de N bits satisfait alors certaines propriétés statistiques de base, indispensables (mais pas suffisantes) à un générateur de nombres pseudo-aléatoires :

- La séquence contient 2^{N-1} bits à 1 et 2^{N-1}-1 bits à 0. Avec 4 bits et une séquence de 15 bits, il y a donc 8×1 et 7×0 (on peut compter dans la séquence précédente pour vérifier).

Cette propriété est facilement expliquée par le fait que les bits que nous obtenons proviennent du LSB (ou MSB, selon) de la liste de *tous les nombres* de 1 à 2^N-1 (l'ordre n'est pas important) et la quantité de nombres pairs est la même (moins un) que celle de nombres impairs. Le léger biais vers 1 est généralement considéré comme insignifiant, il devient rapidement inquantifiable quand N augmente.

- La longueur d'une séquence de bits identiques consécutifs est inversement proportionnelle à sa probabilité. Plus exactement, pour N très grand, on va trouver deux fois moins de d'occurrences de 11 que de 1, deux fois moins de 111 que de 11, etc. On va aussi trouver une seule fois la séquence de N bits à 1.

Histoires de configuration

La méthode que nous venons de voir, appelée « configuration de Galois », est prisée en électronique car la structure de la rétroaction permet des temps de propagation courts, donc un fonctionnement à très haute vitesse. Notre cas particulier n'est pas critique, avec juste un seul XOR, mais dans un récepteur GPS, par exemple, il y en a des dizaines. Par contre, en informatique, cela implique de nombreuses lectures et écritures de bits, ce qui est loin d'être l'idéal.

On peut réduire le nombre d'écritures à une seule en changeant la structure de la rétroaction et du registre : c'est la *configuration de Fibonacci*, en référence à la fameuse suite de Fibonacci `n[i+2]=n[i+1]+n[i]`. Mais dans le cas présent, `n[]` est un tableau de bits, l'addition s'effectue dans `GF(2)` et les indices correspondent à la séquence de dérivation. Et bien que cette structure utilise les mêmes mathématiques et polynômes que précédemment, elle ne renvoie pas tout à fait les mêmes résultats.

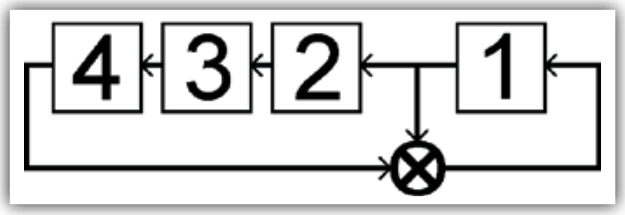

Fig. 2 : Structure d'un LFSR en configuration de Fibonacci

Tous les bits indiqués par la séquence de rétroaction (sauf zéro) sont combinés par un XOR et le résultat remplace un seul bit. La simplicité se paie par un *mélange* moins rapide des bits : un seul change à la fois, donc la suite des valeurs du registre a l'air moins aléatoire, comme l'indique la sortie du programme suivant :

```
/* Exemple de LFSR en configuration de Fibonacci
   sur 4 bits, de polynôme x^4 + x + 1 :
   la rétroaction est le XOR des bits 4 et 1  */
#include <stdio.h>
int main() {
  signed int i, N = 1; /* seed */
  for (i=0; i<16; i++) {
    N <<= 1; /* la multiplication par x */
    N |= ((N >> 4) ^ (N >> 1)) & 1; /* calcule le LSB */
    N &= 15; /* supprime le terme ax^4*/
    printf("%02d\n", N);
  }
  return 0;
}
$ gcc -g lfsr-fibo.c -o lfsr-fibo && ./lfsr-fibo
03  0011
07   0111
15    1111
14     1110
13      1101
10       1010
05        0101
11         1011
06          0110
12           1100
09            1001
02             0010
04              0100
08               1000
01                0001
03                 0011
```

La suite de nombres ainsi créée a l'air encore moins bonne qu'en configuration de Galois, et pourtant la séquence de bits est presque la même. Nous observons qu'elle ne diffère que par le sens :

```
MSB de Fibonacci :  001111010110010
Fibonacci inversé :  010011010111100
MSB de Galois :     001001101011110
```

Puisque la séquence de bits est inversée, on pourrait penser que la configuration de Fibonacci calcule la division alors que la configuration de Galois calcule la multiplication (ou vice versa). C'est faux puisqu'on voit bien que les états internes ne forment pas la même séquence. Par contre, on peut générer la même séquence binaire qu'un registre en configuration de Galois au moyen d'un registre en configuration de Fibonacci si le polynôme est inversé, par exemple `(4,1,0)` pour l'un et `(4,3,0)` pour l'autre. Et en fait, l'état interne nous intéresse peu, seuls les bits générés importent.

NOTE

Pour un résultat final identique, nous avons réussi à économiser le changement de quelques bits et réduit le nombre d'opérations à un seul XOR et un décalage. Faisons maintenant sauter ce dernier !

Élevage de LFSR en batterie

Un LFSR ne produit qu'un seul « bit utile » par itération, mais nous en avons normalement besoin de beaucoup, beaucoup plus. L'article précédent avait présenté une méthode permettant d'obtenir plusieurs bits à la fois dans une configuration de Galois, le nombre de bits étant proportionnel au logarithme de la taille du tableau. Pour obtenir 32 bits à la fois, il faudrait dédier un disque dur à cet usage !

Heureusement, cela n'est pas nécessaire : il suffit de paralléliser plusieurs LFSR. En organisant astucieusement les données, on ne traite plus un seul bit à la fois, mais un mot entier. L'ensemble des registres à décalage est donc logé en mémoire, dans un tableau dont l'index correspond au numéro du bit désiré.

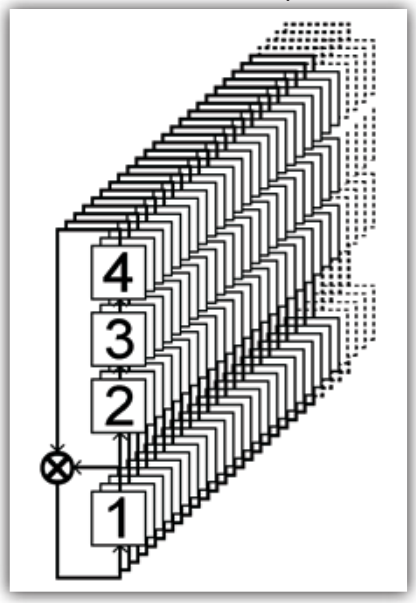

Fig. 3 : Une batterie de LFSR en parallèle. En pratique, la largeur est de 32 à 64 bits et il y a beaucoup plus que quatre mots !

Le calcul d'un nouveau bit (pour tous les LFSR à la fois) revient à lire deux entrées du tableau et à effectuer un XOR entre eux. Pour le décalage, pas question d'effectuer une copie de bloc car le tableau peut être très grand, il suffit de recalculer les pointeurs.

```
/* LFSR 4 bits en configuration de Fibonacci,
   poly = x^4 + x + 1, en parallèle */

#include <stdio.h>

int main() {
  unsigned int i, N,
  T[4]= {0xFFFFFFFF, 0, 0, 0}; /* seed */

  for (i=0; i<16; i++) {
    N  = T[i     & 3];
    N ^= T[(i+3) & 3];
    T[i & 3] = N;
    printf("%08X\n", N);
  }
  return 0;
}
```

Constatons avec joie que le résultat est le même (avec un petit décalage) que le code *bit à bit* précédent :

```
FFFFFFFF
FFFFFFFF
FFFFFFFF
FFFFFFFF
00000000
FFFFFFFF
00000000
FFFFFFFF
FFFFFFFF
00000000
00000000
FFFFFFFF
00000000
00000000
00000000
FFFFFFFF
```

Mais au fait, pourquoi recalculer tous les pointeurs ? On peut faire bien mieux lorsqu'on s'aperçoit que `T[(i+3) & 3]` va devenir `T[i & 3]` au cycle suivant. En gardant dans une variable séparée (assignable à un registre) cette valeur, on peut économiser un accès au tableau. Le code suivant donne un résultat identique, mais le corps de la boucle a été simplifié à une lecture, un XOR, une écriture et une mise à jour d'index :

```
int main() {
  unsigned int i, j, N,
  T[4]= {0xFFFFFFFF, 0, 0, 0}; /* seed */

  j=0;
  N=T[3]; /* remplissage du "pipeline" */

  for (i=0; i<16; i++) {
    N ^= T[j];
```

```
    T[j] = N;
    j++;
    if (j >= 4)
      j=0;
    printf("%08X\n", N);
  }
  return 0;
}
```

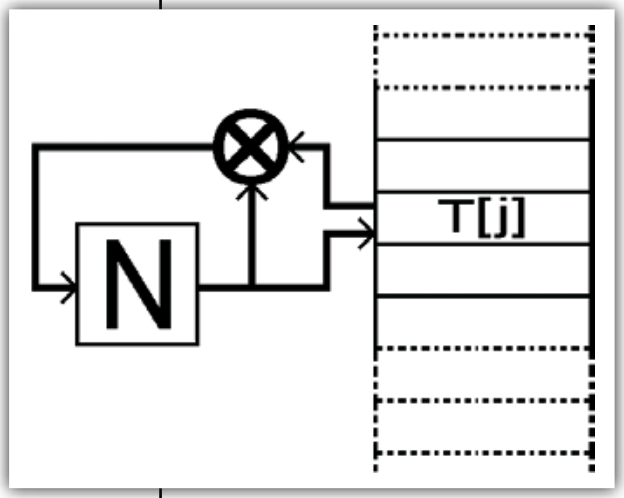

Fig 4 : Structure améliorée

Implications

Nous pouvons ainsi générer des grandes quantités de nombres pseudo-aléatoires, pour des tableaux de 15, 22, 28, 30, 46, 60, 63, 127, 153 ou même 172 cases si vous le désirez. La période peut devenir arbitrairement grande avec 303, 471, 532, 865 ou 900 cases. Si cela ne vous convenait toujours pas, essayez avec un des derniers nombres de Mersenne découverts par le GIMPS, tel 25964951, la période est alors virtuellement infinie...

On peut encore augmenter la période du générateur en utilisant une autre combinaison que le XOR. En faisant interagir les registres parallèles entre eux, on obtient (presque) l'équivalent d'un seul registre long comme tous les registres mis bout à bout. L'addition et la soustraction entre les mots sont les premières possibilités mais il y a mieux pour *brasser* les bits. L'utilisation de la retenue dans l'itération suivante est une première amélioration mais cela ne restera jamais qu'*une suite de Fibonacci un peu bricolée*.

Dans le noyau Linux, le code qui fournit les nombres de `/dev/random` et `/dev/urandom` remplace l'addition par une opération de CRC pour améliorer la qualité de la suite. Un CRC étant un LFSR modifié, on peut s'apercevoir que cela revient à ajouter un autre LFSR orthogonal aux LFSR parallèles.

Et un LFSR étant une multiplication (ou une division, selon les goûts) modulo un polynôme, nous restons confortablement dans $Z/2^N$:-).

Application directe

Pour réaliser plus tard des tests statistiques sur nos algorithmes, il nous faut un générateur de nombres pseudo-aléatoires meilleur que celui présenté en décembre ; nous allons donc utiliser les techniques présentées jusqu'ici.

- Pour la largeur des mots, on peut utiliser 8, 16, 32 ou 64 bits selon le besoin, mais nous allons travailler avec 32 bits par commodité.
- Nous disposons ainsi de 32 LFSR. Nous n'avons pas besoin d'une période pseudo-infinie, juste très très grande. Avec seulement 15 mots, le générateur obtenu est équivalent (en théorie) à un LFSR unique de 480 bits, ce qui est largement supérieur aux 32 bits utilisés en décembre et permet de conserver l'algorithme trinômial développé jusqu'ici.
- En ce qui concerne la *fonction de brassage*, je réutilise l'idée du CRC utilisée dans le code de Linux, mais avec seulement un bit. La différence majeure entre les deux approches est que le générateur de Linux est « perturbé » par des sources extérieures d'entropie alors que notre générateur est en circuit fermé. Le concepteur de la version Linux part du postulat que l'entropie des perturbations est localisée sur les 3 bits de poids faible, le CRC permet donc de les brasser un peu. Cependant, dans notre générateur en vase clos et aux contraintes de sécurité inexistantes, une telle précaution n'est pas nécessaire. De plus, la version Linux fait appel à une table et nous avons vu en décembre que c'est l'opération la plus lente dans le calcul d'un CRC standard.
- Il faut aussi penser à initialiser le tableau avec des valeurs « aléatoires » (pas comme dans le premier exemple) : un autre générateur de nombres pseudo-aléatoires plus petit, initialisé avec soin, fait très bien l'affaire. Par souci de reproductibilité, la « graine » sera une constante arbitraire plutôt que `/dev/urandom`. Ici, j'initialise arbitrairement le LFSR avec 0x1A2B3C4D. Le polynôme générateur est celui du CRC32 standard, puisqu'il n'est pas clairsemé et brouille donc plus rapidement les bits entre eux.

Fig. 5 : Quelques décalages et XOR supplémentaires aident à gommer les petits défauts d'un LFSR.

Pour augmenter le brassage des bits dans cette suite initiale, quelques décalages et XOR suffisent. Leur valeur a été déterminée expérimentalement et convient pour la plupart des utilisations courantes. Ils permettent au générateur d'atteindre un état homogène plus rapidement. Je n'ai pas vérifié la période de l'initialisateur, ni même vérifié qu'il ne s'arrête pas, alors faites attention si de longues séquences initiales sont nécessaires.

Nous obtenons au final un tGFSR : *twisted Generalized Feedback Shift Register* de 15×32 bits et de période 2^{475} (idéalement). Je n'ai pas effectué de tests statistiques poussés, mais les premières analyses de la suite générée n'ont pas montré de défaut flagrant ou de comportement anormal, ni de rebouclage inattendu. Nous ne nous soucions pas ici des « qualités cryptographiques » du résultat, ce qui nous permet d'utiliser un « polynôme faible » qui simplifie et accélère le code. Le plus important est d'une part que la période du générateur soit très au-delà de ce dont nous avons besoin, et d'autre part que l'homogénéisation des nombres soit convaincante.

```
/*
   fichier gfsr32.c
  créé le jeudi 5 janvier 2006 par Yann GUIDON (whygee@f-cpu.org)
  version 30 janvier 2006

Ce fichier crée un "Registre à Décalage à Rétroaction Généralisée"
plus ou moins paramétrique fournissant des nombres pseudo-
aléatoires avec une très longue période et un encombrement modeste
en mémoire.

Le LFSR principal contient 32 registres de N bits chacun,
avec comme trinôme générateur x^N + x + 1. Les 32 registres
sont brassés par un LFSR utilisant le polynôme de CRC32.
La période théorique de l'ensemble est de 2^(N*32 -5).

Au besoin, N (défini par TAILLE_GFSR) peut être fixé à 4, 6, 9,
15, 22, 28, 30, 46, 60, 63, 127, 153, 172, 303, 471, 532, 865,
900...
La période pour N=3 a été mesurée à 4294967296 (2^32) (pas bien)
La période pour N=4 est inconnue mais supérieure à 2,5*10^12.
La période pour N=15 est inconnue mais supérieure à 3,2*10^12.

Autotest :
  gcc -DTEST_GFSR -Os -march=pentium3 -fomit-frame-pointer -o
gfsr32 gfsr32-15.c
(définir d'abord le test à exécuter)
*/

#define CRC32_standard  0x04C11DB7
#define TAILLE_GFSR     15

typedef unsigned int U32; /* DOIT être sur 32 bits */
typedef   signed int S32; /* DOIT être sur 32 bits */

U32 GFSR_table[TAILLE_GFSR], GFSR_temp, GFSR_index;

void init_GFSR(void) {
  int i=0, j;
  U32 t=0x1A2B3C4D;  /* seed */

  do {
    /* applique une petite couche d'uniformisation : */
    t ^= (t>>5) ^ (t<<1);

    /* calcule un pas de LFSR : */
    j=t>>31;
    t<<=1;
    if (j) /* on teste l'ancien MSB de t */
      t^=CRC32_standard;

    GFSR_table[i++]=t;
  } while (i<TAILLE_GFSR);

  GFSR_temp=GSFR_table[TAILLE_GFSR-1];
  GFSR_index=0;
}

inline U32 random_GSFR (void) {
  S32 t;

  GFSR_temp^=GFSR_table[GFSR_index]; /* lecture et XOR */

  t=GFSR_temp;                       /* étape de LFSR */
  GFSR_temp<<=1;
  if (t<0)     /* teste le MSB en lisant le bit de signe */
    GFSR_temp^=CRC32_standard;

  GFSR_table[GFSR_index]=GFSR_temp;  /* écriture */

  if (++GFSR_index >= TAILLE_GFSR)
    GFSR_index = 0;

  return GFSR_temp;
}
```

Avec 3 mots, le générateur reboucle au bout de 2^{32} itérations au lieu de 2^{3*32}, ce qui suggère que la formule qui détermine la période n'est pas correcte. Effectivement, en ajoutant le brassage par CRC32, nous ne sommes plus dans le cadre d'un LFSR normal et le polynôme équivalent au nouveau système n'est plus primitif, donc la période n'est plus maximale.

Par contre, je n'ai pas été en mesure de déterminer expérimentalement la période pour 4 et 15 mots, j'ai arrêté la recherche après respectivement 2500 et 3266 milliards d'itérations. Si quelqu'un veut aller plus loin... En ce qui concerne la vitesse, avec un processeur à 550MHz et le code ci-dessus compilé avec gcc2.95, j'obtiens environ 37 millions de mots par secondes, soit 148M octets par seconde, ce qui est déjà beaucoup plus rapide que le LFSR 32 bits utilisé en décembre. Et ce n'est qu'un début.

Petites optimisations

On peut facilement enlever le `if (bit 31 à 1)` en changeant légèrement le code, mais j'ai préféré laisser le code ci-dessus clair et facile à comprendre. Il suffit de remplacer

```
  t=GFSR_temp;
  GFSR_temp<<=1;
  if (t<0)
    GFSR_temp^=CRC32_standard;
```

par

```
t=((signed int)GFSR_temp) >> 31;
GFSR_temp<<=1; /* ou GFSR_temp+=GFSR_temp; */
GFSR_temp^=t & CRC32_standard;
```

Cette astuce, bénéfique pour la plupart des processeurs, utilise le décalage à droite de la variable `t` pour recopier le MSB (le bit de signe) sur tout le mot. On se sert ensuite du polynôme comme masque, et on évite ainsi au processeur d'effectuer un saut très difficile à prévoir. A la place, nous avons 4 opérations logiques et de décalage faciles à gérer par le compilateur.

GCC3.3 fait mieux qu'un simple décalage : il « comprend » ce qu'il faut faire et utilise l'instruction `CDQ` des x86 qui recopie le MSB de EAX dans EDX, ce qui contourne la limitation des autres instructions usuelles (il aurait fallu d'abord recopier le registre, puis ensuite le décaler). Et il n'y a rien à changer au code source initial :-)

La même idée peut être appliquée lorsque le sens de décalage est inversé. Cette fois-ci, `t` n'a pas besoin d'être signé car la recopie du LSB (le bit 0) est effectuée par propagation de retenue. Le LSB est d'abord masqué, puis on décrémente le résultat : 1 devient 0 et 0 devient -1, ce qui donne un masque « tout ou rien ».

```
#define CRC32_reflechi  0xDB710641

  t=GFSR_temp & 1;
  GFSR_temp>>=1;
  GFSR_temp^= CRC32_reflechi & (t-1);
```

En pratique, il y a plusieurs inconvénients. D'abord, le masque est le complément de celui désiré : on va donc appliquer le polynôme quand il ne faut pas.

Pour corriger cela, il faut ajouter une opération de complémentation à un moment ou un autre, ce qui augmente le nombre d'instructions. Il n'y a pas d'endroit où cette complémentation pourrait être fusionnée en gardant le code portable. Quelques alternatives peu pratiques existent, essentiellement sur x86 :

- L'opération ANDN (AND avec une des entrées complémentée) permet de remplacer `t=GFSR_temp & 1;` par `t=~GFSR_temp & 1;`. Malheureusement, cette instruction n'est disponible que dans certaines conditions (MMX, SSE...).
- On peut passer par le bit de retenue du registre de conditions, au moyen d'un décalage normal, puis le propager au moyen d'une soustraction avec retenue.

```
xor ebx,ebx     ; met ebx à 0
shr eax, 1      ; met le LSB de eax dans le bit de retenue
sbb ebx, byte 0 ; soustrait (0+carry) à 0, donnant 0 ou -1
```

En passant par le bit de retenue, plusieurs autres méthodes sont possibles, sans compter les combinaisons. La plupart nécessitent 3 instructions (comme dans l'exemple), 2 auraient été bienvenus (on aurait voulu ne pas avoir à initialiser EBX à 0).

Les instructions `SETcc` auraient été utiles si elles ne travaillaient pas que sur un octet (ce qui est stupide). De plus, à l'inverse de la première alternative, ces méthodes ne sont pas disponibles avec les registres MMX ou SSE et il faut avoir recours à du code assembleur.

D'autre part, le décalage à droite (division par 2) offre moins d'alternatives que le décalage à gauche. Ce dernier peut être réalisé par une addition (`GFSR_temp+=GFSR_temp`) ce qui libère l'accès à l'unité de décalage, qui pourrait être déjà occupée à copier le bit de signe. C'est important à noter car la majorité des processeurs, même s'ils peuvent exécuter deux ou trois instructions par cycle, ne peuvent effectuer qu'un seul décalage à la fois. Nous conserverons donc le sens usuel des bits.

Grosses optimisations

A priori, sans tout recoder en langage assembleur, on ne peut pas améliorer beaucoup le code précédent. Comme le code de CRC de décembre, il n'est pas parallélisable et peu de techniques d'optimisation s'appliquent, sauf la réduction des mouvements de données entre les registres et la mémoire, ce qui implique un déroulage de la boucle.

Dans cette situation, il faut faire tenir la table entière dans les registres internes du processeur. Aucun souci pour les architectures MIPS, PowerPC ou Alpha qui disposent de 32 registres entiers. Pour x86-64 ou ARM, il n'y en a que 16 (et encore...), donc impossible de travailler avec 15 mots. Et comme 4 mots fournissent déjà une période suffisante, nous allons nous en contenter, ce qui permet de travailler avec des x86 usuels.

La première étape consiste à transformer le code `inline` en macro. D'une part, GCC supporte mieux ce type de code quand les options d'optimisation ne sont pas activées. C'est pratique lorsqu'il faut déterminer l'algorithme avec `gcc -g`, sinon on ne comprend rien du tout car en compilant avec `gcc -Os`, les lignes du source ne sont pas exécutées dans l'ordre initial.

D'autre part, une fonction `inline` a des propriétés subtiles dues à son statut à mi-chemin entre une macro et une fonction. Et comme elle reste une fonction, la sémantique d'accès aux données doit être respectée même lorsque la fonction est *inlinée*. Cela implique des accès superflus aux données globales et des inférences de valeurs non exploitées. C'est essentiellement ce qui ralentit le programme puisque le calcul ne peut pas être plus optimisé.

```
#define GFSR_MACRO(index)    {       \
  signed int t;                      \
  GFSR_temp^=GFSR_table[index];      \
```

```
  t=((signed int)GFSR_temp) >> 31;     \
  GFSR_temp<<=1;                       \
  GFSR_temp^=t & CRC32_standard;       \
  GFSR_table[index]=GFSR_temp;  }
```

Cette version ne diffère presque pas de la version `inline`. Le calcul a été amélioré, mais il faut surtout noter les accolades, qui définissent un sous-bloc d'instructions à l'intérieur duquel on peut définir une variable locale. Si `t` est défini ailleurs, GCC va sauver sa valeur en mémoire, essentiellement pour permettre à d'autres parties du code de le lire, alors que nous savons bien que cela ne se produira pas. Cette différence subtile contribue déjà à une amélioration (environ 10%) de la vitesse du code utilisant cette macro.

L'autre différence est que cette macro a besoin du numéro d'index dans la table. Utiliser des indices constants contribue encore à réduire (d'environ 25%) le nombre d'instructions inutiles (d'une part l'accès lui-même, d'autre part le rebouclage de l'index en cas de dépassement).

De telles améliorations, qui semblent minimes, se ressentent lorsque leur fonction est centrale dans un programme ; tel est le cas lorsqu'on veut tester la suite, en particulier chercher sa période. A force d'optimisations, la vitesse a été doublée mais le code utilisant cette macro doit gérer correctement les variables autour de la boucle interne, ce qui le rend un peu complexe. C'est la variable `GFSR_index` qui cause le plus de soucis, puisque la boucle interne ne la met pas à jour :

```
/* Dans le main() de l'autotest : */
#ifdef test_pattern2
#define TAILLE_SEQUENCE (10)
  U32 sequence[TAILLE_SEQUENCE], debut, j;

  /* enregistre le début de la séquence à rechercher */
  debut=random_GSFR();
  for (iteration=1; iteration<TAILLE_SEQUENCE; iteration++)
    sequence[iteration]=random_GSFR();

  do { /* while (1) */

    /* re-alignement de l'index */
    while (GFSR_index!=0) {
      if (random_GSFR()==debut)
        goto match;
    }
    /* donc ici, GFSR_index==0, on peut alors coder les indices en
constantes */

    while(1) { /* c'est la boucle qui consomme 99,9999% du CPU */
      GFSR_MACRO(0)
      if (debut==GFSR_temp) goto match0;
      GFSR_MACRO(1)
      if (debut==GFSR_temp) goto match1;
      GFSR_MACRO(2)
      if (debut==GFSR_temp) goto match2;
      GFSR_MACRO(3)
      if (debut==GFSR_temp) goto match3;
      iteration+=TAILLE_GFSR;

      if ((iteration & 0xFFFFFFF)==0) {
        printf("%citeration %Ld ...", 0xd,iteration);
        fflush(NULL);
      }
    }
    match3:  iteration  ++, GFSR_index ++;
    match2:  iteration  ++, GFSR_index ++;
    match1:  iteration  ++, GFSR_index ++;
    match0:  iteration  ++, GFSR_index ++;
    if (GFSR_index >= TAILLE_GFSR)
      GFSR_index -=TAILLE_GFSR;
match:
    printf ("%cIteration %Ld : ",0xd,iteration);

    /* boucle qui consulte le reste de la liste */
    j=0;
    do {
      if (++j==TAILLE_SEQUENCE) {
        printf ("%cBoucle trouvée !\n%c",7,7); /* réveiller l'opérateur*/
        return 0;
      }
      iteration++;
    } while (random_GSFR()==sequence[j]);
    /* Si pas d'autre correspondance, continuer */
    printf ("Match partiel de %d nombres.\n", j);
  } while (1);
#endif /* test_pattern2 */
```

L'utilisation de sauts, de labels et d'autres structures peu recommandées est indispensable pour conserver l'avantage de la vitesse. Seule une petite partie du code occupe le processeur, le reste a donc peu d'importance. Ainsi, il suffit de seulement 40 secondes pour tester 2^{32} mots à 700MHz, soit environ 400M octets par seconde.

Application utile

Le code suivant est une amélioration plus poussée, utilisant une autre version de la macro. Par simplicité, seuls des mots de 32 bits sont gérés et il n'y a pas d'alignement sur des frontières d'octets. Si vous voulez coder l'alignement, la technique la plus simple consiste à générer un mot entier pour n'écrire en mémoire que les octets voulus.

```
/* Remplit un tampon donné avec de la bouillie sur 32 bits.
   Ici on prend le parti que tout doit être fait "en interne"
   pour éviter que le compilo ne fasse des accès indésirés en mémoire,
   donc on maximise les inférences de valeurs. En particulier,
   on élimine la dernière assignation GFSR_table[index]=GFSR_temp
   et on effectue un renommage cyclique : */

#define GFSR_MACRO2(current, last) { \
  S32 t;                             \
  current^=last;                     \
  t=((signed int)current) >> 31;     \
  current<<=1;                       \
  current^=t & CRC32_standard;  }

/* size et ptr sont alignés sur 4 octets par simplicité
     33.86s / 2^32 mots @700MHz, ou 500Moctets/s */
void random_buffer(U32 *ptr, int size) {
  U32 GFSR0, GFSR1, GFSR2, GFSR3;
  int index2, size2;

/* Chargement de la table, modulo l'index de départ : */
  index2 = (GFSR_index+1) & 3;
```

```
                                /* +  00 01 10 11 */
  GFSR0 =GFSR_table[GFSR_index];    /* 00 00 01 10 11 */
  GFSR1 =GFSR_table[index2];        /* 01 01 10 11 00 */
  GFSR2 =GFSR_table[GFSR_index^2];  /* 10 10 11 00 01 */
  GFSR3 =GFSR_table[index2^2];      /* 11 11 00 01 10 */
/* nb: par définition, GFSR_temp=GFSR_table[GFSR_index+3],
   donc pas besoin d'y toucher */

  size2=size>>2;
  while (size2--) { /* La boucle principale */
    GFSR_MACRO2(GFSR0, GFSR3)   ptr[0]=GFSR0;
    GFSR_MACRO2(GFSR1, GFSR0)   ptr[1]=GFSR1;
    GFSR_MACRO2(GFSR2, GFSR1)   ptr[2]=GFSR2;
    GFSR_MACRO2(GFSR3, GFSR2)   ptr[3]=GFSR3;
    ptr+=4;
  }

  /* Sauve la table (exactement l'inverse du chargement) */
           GFSR_table[GFSR_index]   =GFSR0;
           GFSR_table[index2]       =GFSR1;
           GFSR_table[GFSR_index^2] =GFSR2;
  GFSR_temp=GFSR_table[index2^2]    =GFSR3;

  /* ajoute le petit bout qui dépasse de la boucle */
  size &= 3;
  while (size--) {
    *(ptr++) = random_GSFR();
  }
}
```

Par souci d'interfaçage avec d'autres fonctions utilisant `random_GSFR()`, la cohérence de la table et des variables globales est conservée, ce qui alourdit un peu le code, mais comme 4 est une petite puissance de 2, le tout reste compact. La vitesse est encore améliorée, il ne faut plus que 33s pour générer 4 milliards de mots, soit environ 500Mo (un demi-gigaoctet !) par seconde si la destination est en mémoire cache L1.

La réduction du temps de génération provient de la suppression des sauts conditionnels (remplacés par des écritures en mémoire, moins lourdes) et de la simplification (ultime) par inférence de la variable `GFSR_temp`. Puisque celle-ci est (par définition) égale à la valeur de la table à l'index précédent, et comme la table est contenue entièrement en registres, il suffit d'accéder directement au registre, et non à la variable temporaire. Finalement, l'emploi de variables scalaires locales à la place d'un tableau global garantit que le compilateur ne va pas écrire en mémoire les résultats intermédiaires à l'intérieur de la boucle.

Je ne pense pas pouvoir écrire du code encore plus rapide tout en restant portable, général et pratique. La fonction ci-dessus devrait satisfaire la majorité des besoins usuels, à l'exception, peut-être, des concours de type *Obfuscated* :-)

ATTENTION

Ce code n'est absolument pas adapté à un usage cryptographique ou ayant rapport avec la protection ou la confidentialité de données, même de loin ! Les LFSR, en particulier d'une taille si ridiculement petite ou avec un polynôme clairsemé, sont extrêmement simples à « casser ». Ici, il suffit d'obtenir 2×4×32 = 256 bits (au plus, et c'est généralement excessivement facile) pour reconstruire l'état interne des variables. On peut alors prédire ou reconstituer tous les autres bits du flux.

Moralité : au moindre doute, utiliser `/dev/random` ou, au pire, `/dev/urandom` (plus rapide, non bloquant mais un peu moins sûr en théorie).

Conclusion

J'ai fait mon possible pour expliquer de manière concise et accessible les notions de base utilisées par les générateurs de nombres pseudo-aléatoires. En dehors de son intérêt intrinsèque, cet exposé était aussi nécessaire pour au moins trois raisons :

- Comprendre comment sont conçus les polynômes générateurs, pour ensuite savoir comment les bricoler sans briser leurs propriétés.
- Faire le pont entre les configurations de Galois et de Fibonacci, pour pouvoir économiser quelques opérations d'écriture.
- Introduire l'idée de LFSR parallèles.

Tous ces points seront exploités dans un prochain article traitant (encore) de CRC. Je pressens déjà une accélération d'un facteur 4 par rapport au code (pourtant hyper optimisé) publié en décembre. Ensuite, la série pourra se recentrer sur la compression :-)

LIENS

- Miroir des sources : http://ygdes.com
- L'excellent, l'incontournable, le compréhensible, l'anglophone texte de Ross N. Williams, « *A painless guide to CRC error detection algorithms* » : http://www.ross.net/crc/crcpaper.html (à lire et à relire jusqu'à ce que cela semble évident)
- L'indispensable référence, pas seulement pour les apprentis-cryptographes : Bruce Schneier, *Cryptographie Appliquée*, Editions Wiley/*International Thomson Publishing* (les Corps de Galois sont expliqués dans les chapitres 11.3, 16.1 et 16.2)
- *The Great Internet Mersenne Prime Search*: http://www.gimps.org
- Et dans votre noyau favori : `linux/drivers/char/random.c` avec les fonctions `add_entropy_words` et `poolinfo_table[]` qui contiennent quelques polynômes assez spéciaux.

Yann Guidon,