Esirem Infotronique 3A

Projet de Mathématiques

Générateurs de nombres pseudo-aléatoires

Auteurs:
CHENY Valentin, IMHOFF Guillaume, COUTAREL Allan

2021-2022

Sommaire

Liste des figures

Liste des tableaux

1 Introduction

1.1 Introduction au sujet

Des méthodes probabilistes bien connues permettent de simuler des variables aléatoires suivant une loi prescrite à partir de variable aléatoires uniformes sur]0, 1[. La question de générer le hasard informatiquement semble quant à elle inextricable puisque le hasard serait le fruit d'un algorithme déterministe. Il est pourtant possible de générer des nombres pseudo-aléatoires, utilisant l'algèbre et permettant d' « imiter » le hasard. On s'intéressera en particulier aux générateurs congruentiels que l'on implémentera dans un langage informatique.

1.2 Les générateurs de nombres pseudo-aléatoires

Bien qu'une véritable source de nombres aléatoires se doit d'être créée à partir de procédés physiques, il est néanmoins possible de générer informatiquement un nombre qui semble aléatoire mais qui dans les faits est le résultat d'un calcul mathématique. On parle de nombres pseudo-aléatoires et l'algorithme qui les génère est alors appelé générateur de nombres pseudo-aléatoires.

Le principe de fonctionnement est le suivant : l'algorithme a besoin d'un nombre de départ appelé graine (seed en anglais) sur laquelle une formule mathématique est appliquée. On obtient alors un nouveau nombre, pseudo-aléatoire, qui sera utilisé ensuite en entrée de l'algorithme pour obtenir le prochain nombre pseudo-aléatoire. On répète ce processus autant de fois que l'on veut de nombres pseudo-aléatoires.

Il est important de remarquer que si l'on fournit toujours la même graine au générateur pseudo-aléatoire, celui-ci donnera toujours la même suite de nombres pseudo-aléatoires. Il est donc essentiel de penser à changer la graine autant que possible.

Ces générateurs ont des utilités dont de nombreux domaines comme les jeux vidéo, la simulation, l'analyse, l'échantillonnage, la cryptologie... Les générateurs les plus connus sont ceux utilisant la méthode de Von Neumann, appelée méthode middle square, ou bien la méthode de Fibonacci ou encore les générateurs congruentiels linéaires auxquels nous allons nous intéresser.

2 Générateur congruentiel linéaire

2.1 Définition

Le générateur congruentiel linéaire produit des nombres pseudo-aléatoires formant une suite dont chaque terme dépend du précédent suivant la formule suivante :

$$X_{n+1} = (a * X_n + c) \bmod m$$

où a est le multiplicateur, c l'incrément, et m le module.

$$\text{On a } m \in \mathbb{N}, \text{ et } (a,\ c,\ X_0) \in [\![0\ ;\ m[\![^3$$

Le premier terme de la suite X_0 est appelé la graine, comme énoncé dans la partie précédente. C'est donc elle qui va permettre de commencer à générer une suite pseudo-aléatoire et pour chaque graine différente, on obtient une suite différente, d'où la nécessité de faire varier la graine. Nous verrons par la suite comment choisir les paramètres du générateur, à savoir la graine, le multiplicateur, et l'incrément afin d'obtenir la suite la plus aléatoire possible.
La suite (X_n) ainsi définie est périodique à partir d'un certain rang. En effet, du fait de l'opérateur *mod*, tous les termes de cette suite sont compris entre 0 et m-1 et comme chaque terme dépend du précédent, si un nombre apparaît une deuxième fois, toute la suite se reproduit à partir de ce nombre. Afin d'obtenir un générateur pseudo-aléatoire performant, il est donc primordial d'avoir une période suffisamment longue, au mieux de longueur m.

2.2 Implémentation

Nous allons implémenter le générateur congruentiel linéaire en langage c++. Pour se faire, nous avons procédé à la création d'un objet geneCongruentiel correspondant au générateur congruentiel linéaire. Le code comporte donc un fichier geneCongruentiel.h et geneCongruentiel.cpp pour définir et créer la classe de l'objet et un fichier main.cpp dans lequel nous effectuons nos tests de générations de nombres pseudo-aléatoires avec le générateur que nous avons conçu. L'entièreté du code commenté est disponible en annexe.

Parmi les attributs de la classe, on retrouve un entier a (le multiplicateur), un entier c (l'incrément), un entier m (le module), un entier seed (la graine) et un entier X étant le dernier nombre pseudo-aléatoire calculé par le générateur.

La classe dispose d'abord d'un simple constructeur standard et de simples accesseurs en lecture de ses attributs dont nous ne détaillerons pas l'implémentation.

La méthode de la classe qui nous intéresse est genNb() :

```
1 int GeneCongruentiel::genNb() {
2     _X = (_a*_X+_c)%_m;
3     return _X;
4 }
```

En effet, cette méthode est l'essence même de notre générateur congruentiel linéaire puisqu'elle renvoie et stocke en mémoire dans l'attribut X de la classe un nombre pseudo-aléatoire calculé par la formule énoncée précédemment dans la définition.

La fonction makeSequence prend en paramètres un générateur congruentiel linéaire et un entier l, et va renvoyer la suite des l premiers nombres pseudo-aléatoires calculés par une copie locale du générateur placé en paramètre de la fonction. En effet, les calculs sont effectués avec une copie local du générateur initialisé avec la graine uniquement pour s'assurer que le premier terme de la suite X_0 soit bien la graine. Car supposons que le générateur placé en paramètre est déjà fait appel à sa méthode genNb(), alors son attribut X ne contient plus la graine et la suite renvoyée par la fonction ne débuterait donc pas par la graine. C'est pourquoi on place la graine dans l'attribut X de la copie locale du générateur à partir duquel on calcule la suite des nombres pseudo-aléatoires de taille l. A noter que la copie locale du générateur est créée automatiquement par l'appel de la fonction en c++.

La dernière fonction est getStats. Elle renvoie la liste de tous les nombres pseudo-aléatoires possibles, en partant de la graine, créés par le générateur congruentiel linéaire placé en paramètre, c'est-à-dire qu'elle renvoie une unique période de la suite de nombres pseudo-aléatoires générés par le générateur congruentiel linéaire. Cette méthode va principalement nous servir pour les tests statistiques.

Exemple Voici un exemple d'utilisation de la classe du générateur et des fonctions associées :

```
int main(){
  int taille_periode = 0;
  GeneCongruentiel gene(125,25,16,256);
  std::cout << gene.genNb() << std::endl;
  std::cout << gene.genNb() << std::endl;
  std::cout << gene.genNb() << std::endl;
  std::vector<int> suite = makeSequence(gene, 100);
  std::vector<int> stats = getStats(gene);
  for(int i : suite)
    std::cout << i << " ";
  std::ofstream data("data.csv");
  for(int i : stats) {
    data << i << std::endl;
    taille_periode++;
  }
  std::cout << std::endl;
  std::cout << "La taille de la période du générateur est : " <<
   taille_periode << std::endl;
  return 0;
}
```

Voici donc le résultat de l'exécution du programme :

```
69
205
21
125 69 205 21 29 229 109 181 189 133 13 85 93 37 173 245 253 197 77
    149 157 101 237 53 61 5 141 213 221 165 45 117 125 69 205 21 29
```

```
    229 109 181 189 133 13 85 93 37 173 245 253 197 77 149 157 101
   237 53 61 5 141 213 221 165 45 117 125 69 205 21 29 229 109 181
   189 133 13 85 93 37 173 245 253 197 77 149 157 101 237 53 61 5
   141 213 221 165 45 117 125 69 205 21
La taille de la période du générateur est : 32
```

Et le fichier data.csv contient les données suivantes :

```
125 69 205 21 29 229 109 181 189 133 13 85 93 37 173 245 253 197 77
    149 157 101 237 53 61 5 141 213 221 165 45 117
```

Cet exemple illustre bien les possibilités offertes par notre générateur congruentiel linéaire. Cependant, les paramètres a, c, m et la graine ont ici été choisis arbitrairement. Nous allons donc voir quels sont leurs impacts sur la suite des nombres pseudo-aléatoires générée et ainsi comment les choisir idéalement pour obtenir la suite la plus aléatoire possible.

3 Choix des paramètres

La qualité du générateur va dépendre des choix des paramètres a, c et m puisqu'on ne peut pas se satisfaire de choisir arbitrairement ces paramètres pour un X_0 donné, ce qui nous donnerait un générateur plus ou moins aléatoire. La longueur d'une période d'un générateur congruentiel linéaire est inférieure ou égale à son module. L'objectif est donc d'obtenir un générateur de période de longueur m, dit pleine période, afin qu'il puisse être le plus aléatoire possible. Mais comment choisir a, c et m tels que $\forall$ X_0, le générateur soit pleine période.

3.1 Choix du multiplicateur a et de l'incrément c

Pour $c \neq 0$, on obtient un générateur pleine période si et seulement si :
1. c et m sont premiers entre eux, soit $PGCD(c, m) = 1$
2. Pour chaque nombre premier p divisant m, $a - 1$ est un multiple de p
3. si m est un multiple de 4, alors $a - 1$ est un multiple de 4

Pour $c = 0$, on obtient un générateur pleine période si :
1. m est un nombre premier
2. a^{m-1}-1 est un multiple de m
3. $\forall k = 1, 2, ..., m - 2$, on a $a^k - 1$ n'est pas divisible par m

Remarque Si m est une puissance de 2, il suffit de choisir c impair et $a = 4n + 1$, $\forall n \in \mathbb{N}^*$.

3.2 Choix du module m

Le choix du module m va permettre d'augmenter ou de diminuer la taille de la période. En effet, le générateur congruentiel linéaire, de période pleine, génère des nombres pseudo-aléatoires compris entre 0 et m-1. Le module permet donc de définir la borne supérieure de l'intervalle dans lequel seront compris les nombres générés pseudo-aléatoirement.
Si a et c sont déjà fixés, il faut veiller à choisir le module de telle sorte que les conditions permettant d'obtenir un générateur pleine période soient toujours respectées. Mais on pourrait également fixer d'abord le module et chercher un multiplicateur et un incrément en fonction du m fixé afin d'obtenir un générateur de période pleine de taille voulue.

Remarque Le choix de X_0 pour un générateur de période pleine avec a, c et m fixés, sachant que $X_0 \in [\![0 \ ; \ m[\![$, va donc uniquement influer sur le premier terme à partir duquel la période va démarrer, mais la période générée sera identique.

4 Tests statistiques

4.1 Le potentiel

Le potentiel est une première approche qui permet d'écarter des générateurs peu aléatoires. On ne calculera le potentiel que pour des générateurs de période pleine. Le potentiel s est alors défini comme le plus petit entier tel que :

$$(a-1)^s \equiv 0 \mod m$$

Un potentiel inférieur ou égal à trois, apparaît immédiatement comme trop faible car la suite n'est pas suffisamment aléatoire. En fait un potentiel de 5 ou plus est conseillé.

Rappelons cependant que la notion de potentiel est uniquement là pour écarter quelques mauvais générateurs. Un générateur possédant un potentiel supérieur à 5 ne peut en aucun cas être considéré comme bon par cette simple conclusion, il doit d'abord subir quelques tests.

4.2 Test d'uniformité

Pour vérifier l'uniformité du générateur congruentiel, nous avons besoin réaliser un test du khi-2. Pour ce faire, on note les effectifs théoriques et observés dans k intervalles distincts allant de 0 à 1. Nous allons procéder à un exemple pour appuyer nos propos. On prendra $k = 100$ pour notre exemple. Afin de simplifier les choses, on utilise une suite $(U_n) = X_n/m$ pour avoir des valeurs contenues dans le segment $[0; 1[$.
Le test d'uniformité est important, car il est primordial d'obtenir des termes uniformément répartis dans $[0; 1[$ pour (U_n). En effet, il ne doit pas y avoir de segment privilégié dans $[0; 1[$, sinon, cela constituerait une faille du hasard qui pourrait être exploitée afin d'anticiper dans quel segment de $[0; 1[$ un nombre généré aurait le plus de chances de se trouver, et donc cela augmente aussi les chances de deviner exactement le nombre généré.

Pour notre exemple, on effectue le test avec comme paramètres : $X_0 = 30,\ a = 13,\ c = 5,\ m = 1024$ pour notre générateur.
Il faut ensuite utiliser la formule :

$$\chi^2 = \sum_i \frac{(O_i - T_i)^2}{T_i}$$

On a ici un générateur pleine période, cela implique que si l'on prend l'intégralité des 1024 valeurs pour le test du khi-2, on obtiendra $\chi^2 = 0$, tout simplement car on aura une fois chaque valeur comprise entre $[\![0\ ;\ 1024[\![$, cela donne alors une uniformité parfaite (en apparence seulement). Pour mesurer l'uniformité correctement on se limite donc (par exemple) aux 600 premières valeurs.

On obtient alors le tableau suivant :

intervalles	0-0.01	0.01-0.02	0.02-0.03	...	0.98-0.99	0.99-1.00
effectifs observés	8	4	4	...	3	7
effectifs théoriques	6	6	6	...	6	6

Table 1: Effectifs observés et théoriques en fonction des intervalles définis pour le test du Khi-2

A présent, $\chi^2 = 50.67$. Cette valeur seule ne signifie pas grand-chose, il nous faut la comparer à la valeur de la distance critique. Pour ce faire nous nous référons à la table de la loi de khi-2 (disponible en annexe), qui nous indique que pour un degré de liberté de 100 et un risque d'erreur fixé à 5% (par convention), la valeur critique est de 124.34. Le χ^2 obtenu est bien inférieur à cette valeur limite, ce qui confirme l'uniformité du générateur congruentiel linéaire que nous avons pris pour exemple. Si χ^2 est supérieur ou égal à la valeur critique, il faut alors écarter le générateur puisqu'il ne présente pas une uniformité convenable pour avoir un caractère aléatoire acceptable.

On peut effectuer ce test avec un générateur qui n'est pas pleine période en prenant cette fois toutes les valeurs. Pour un générateur pleine période, il faut se limiter à une partie de la période comme expliqué précédemment.

4.3 Test spectral

L'indépendance des nombres générés est aussi une partie essentielle de leur caractère aléatoire. Le but est de faire passer un test d'impédance à un générateur congruentiel linéaire afin de savoir s'il est possible ou non de deviner un nombre généré par celui à partir des précédents nombres déjà générés.
Le but du test spectral est d'étudier la distribution d'une sortie de nombres, afin de vérifier leur caractère aléatoire. Son fonctionnement consiste en la recherche de ligne, ou plan si on teste le générateur dans trois dimension, et à l'étude de l'écart entre les points et les lignes.

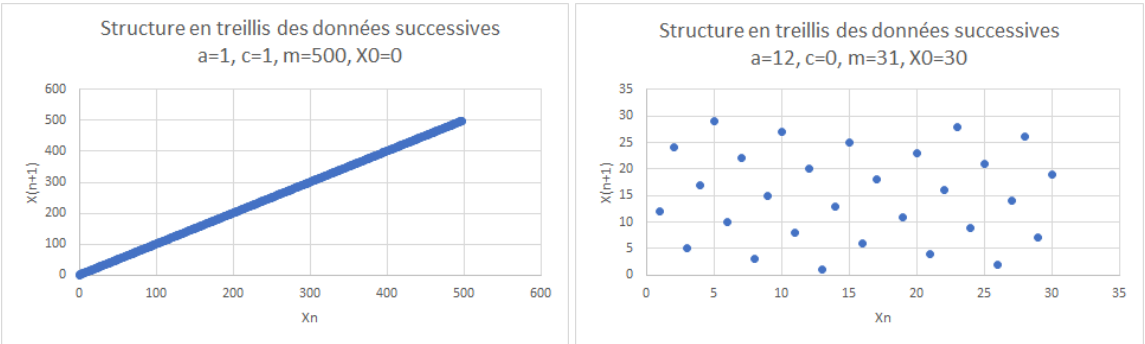

Figure 1: Exemples de structure en treillis pour le test spectral

On remarque que le premier générateur nous fournit une unique droite, on en déduit directement et visuellement que c'est générateur fournit une suite de nombres non aléatoire. En fait, le générateur congruentiel linéaire ayant pour paramètres a=1, c=1, m=500 et comme graine 0 génère simplement, avec un pas de 1, les entiers entre $[\![0\ ;\ 500[\![$.

Pour le deuxième générateur congruentiel linéaire choisi, on peut compter 7 droites qui contiennent à elles 7 tous les nombres générés. Ce générateur semble d'emblée plus aléatoire que le précédent.

Dans le test spectral, la qualité du générateur congruentiel linéaire est évaluée par le nombre de droites et l'espaces qui les séparent. Moins il y a de droites et plus elles sont éloignées, alors plus la qualité aléatoire du générateur est faible. On peut donc comparer facilement 2 générateurs congruentiels linéaires de même période mais ayant des paramètres différents, afin de savoir lequel est le plus intéressant à choisir pour une simulation, c'est-à-dire lequel est le plus favorable à proposer une suite de nombres imitant le plus possible le hasard.

5 Combinaison de plusieurs générateurs congruentiels linéaires

5.1 Description

Il apparaît que les générateurs congruentiels linéaires simples sont loin d'être parfaits : par exemple il est impossible d'obtenir 2 fois le même tirage avant d'avoir parcouru une période. Pour mitiger ce problème et créer un générateur plus complexe, nous avons décidé de créer un générateur combinant plusieurs générateurs congruentiels linéaires, que nous appellerons super générateur congruentiel linéaire (SGC).

Ce générateur est basé sur un premier générateur congruentiel, appelé générateur de choix, ainsi que sur un vecteur (une combinaison) d'autres générateurs congruentiels linéaires. À chaque tirage, le super générateur tire tout d'abord une valeur k avec le générateur de choix, puis tire une nouvelle valeur avec le générateur n du vecteur, avec

$$n = k \mod taille\_vecteur$$

5.2 Choix des paramètres

On utilise, en générateur de choix, un générateur à période pleine ayant un module supérieur à la taille totale du vecteur (sans quoi certains générateurs du vecteur seraient ignorés), et ayant obtenu de bons résultats aux test de χ^2 et au test spectral. Un module élevé assurera par ailleurs une grande période du SGC.

Pour les générateurs congruentiels qui composeront le vecteur, on choisit des générateurs à période pleine ayant des modules égaux afin de suivre une loi uniforme, et ayant obtenus de bons résultats au test de χ^2 et au test spectral.

5.3 Essai du générateur

On considère pour les tests qui suivent le SGC défini par la fonction $exempleSGC()$ présente dans *main.cpp*. On obtient ainsi tout d'abord une période de 258064, et des valeurs qui semblent aléatoires, ce qui évidemment n'est pas suffisant, nous allons donc à nouveau conduire des tests statistiques.

5.4 Test d'uniformité

En appliquant le test de χ^2 tel que décrit précédemment sur les 1000 premières valeurs obtenues avec le générateur, on obtient cette fois-ci $\chi^2 = 36,27$. En se référant à la même table que précédemment, la valeur critique (124,34) ne change pas. On observe aussi que le score de ce générateur est meilleur que score obtenu par le générateur testé précédemment, puisque plus proche de 0.

5.5 Test spectral

On réalise, pour juger de l'indépendance des valeurs, le même test spectral que défini précédemment, sur les 1000 premières valeurs générées (par soucis de visibilité).

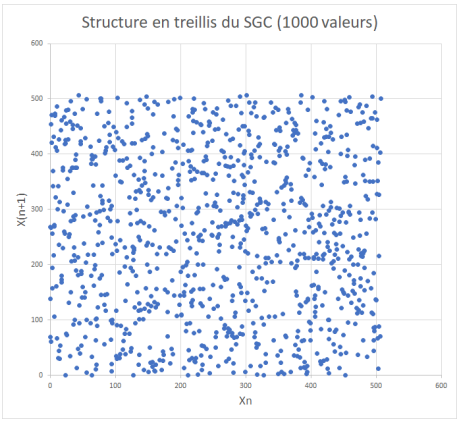

Figure 2: Structure en treillis des 1000 premières valeurs du SGC pour le test spectral

Aucune structure géométrique simple n'est discernable (il en est de même lorsque l'on augmente le nombre de valeurs, mais le graphique devient bien moins lisible). On va donc également appliquer un test spectral en 3 dimensions : on considère les 3 dernières valeurs, et non plus les 2 dernières, pour placer un point. On considère ici l'ensemble des 258064 valeurs.

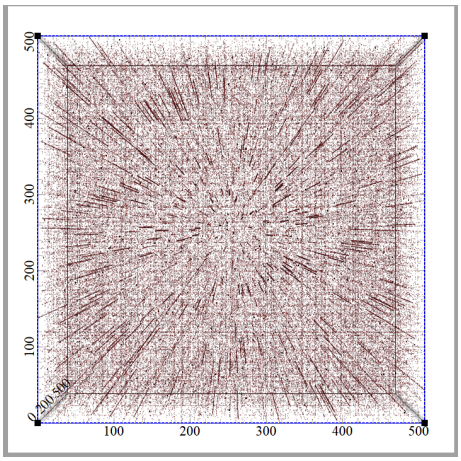

Figure 3: Structure 3D pour le test spectral

Cette fois-ci, des figures géométriques sont bien visibles : on distingue clairement des droites à travers le nuage de points. Pour autant, elles sont nombreuses, et aucun plan ne se distingue, quel que soit l'orientation du graphique. Le test spectral est donc très concluant pour ce générateur, tout comme le test d'uniformité.

5.6 Complexité temporelle et conclusion

La génération d'une valeur aléatoire, une fois le générateur initialisé, ne nécessite de générer que 2 valeurs aléatoires avec des générateurs congruentiels linéaires. Vu le gain en disparité des valeurs obtenues, ce gain vaut donc très largement le coût.

Enfin, on peut remarquer qu'il est possible de mixer, avec ce générateur, différents types de générateurs, pour obtenir des nombres semblants toujours plus aléatoires, voire même d'utiliser des super générateurs dans d'autres super générateurs.

6 Conclusion

Simuler des variables aléatoires informatiquement nécessite donc d'utiliser des générateurs de nombres pseudo-aléatoires, utilisant l'algèbre et permettant d'«imiter » le hasard. Le générateur congruentiel linéaire en est un parfait exemple, simple à implémenter et plutôt efficace, à condition de choisir convenablement les paramètres de celui-ci. Pour s'assurer de ne pas utiliser un générateur pseudo-aléatoire congruentiel linéaire mauvais, il est important de réaliser les tests que nous avons vu. Cependant, ils ne garantissent pas que le générateur est bon, mais ils éliminent les générateurs vraiment mauvais.

Bibliographie

[1] J. E. Gentle. *Random Number Generation and Monte Carlo Methods*. Springer, 1998.

[2] Wikipédia. Générateur de nombres pseudo-aléatoires, 2021. URL https://fr.wikipedia.org/wiki/Générateur_de_nombres_pseudo-aléatoires.

[3] Wikipédia. Générateur conguentiel linéaire, 2020. URL https://fr.wikipedia.org/wiki/Générateur_congruentiel_linéaire.

[4] Wikipédia. Test spectral, 2021. URL https://fr.wikipedia.org/wiki/Test_spectral.

[5] Lwh. Les générateurs de nombres pseudo-aléatoires, n.d. URL http://lwh.free.fr/pages/algo/crypto/prng.html.

[6] Gilles Dubois. Génération de nombres pseudo-aléatoires, n.d. URL http://gilles.dubois10.free.fr/probabilites/concept_simulation.html.

Annexe

```
/*
* @File:    geneCongruentiel.h
* @Project: Générateur de nombres pseuso-aléatoires - Mathématiques ESIREM 3A IT
* @Author:  Allan Coutarel, Valentin Cheny, Guillaume Imhoff
*/

#ifndef _GENECONGRU_
#define _GENECONGRU_

#include <iostream>
#include <vector>

class GeneCongruentiel {

public:
    GeneCongruentiel(int seed, int a, int c, int m);
    int getA() const;
    int getC() const;
    int getM() const;
    int getSeed() const;
    int genNb();
    friend std::vector<int> makeSequence(GeneCongruentiel gene, int l);

private:
    int _seed;
    int _X;
    int _a;
    int _c;
    int _m;
};

std::vector<int> getStats(GeneCongruentiel gene);

#endif
```

```cpp
/*
* @File:    geneCongruentiel.cpp
* @Project: Générateur de nombres pseuso-aléatoires - Mathématiques ESIREM 3A IT
* @Author:  Allan Coutarel, Valentin Cheny, Guillaume Imhoff
*/

#include <iostream>
#include "geneCongruentiel.h"

/**
 * Constructeur standard d'un générateur congruentiel
 * Initialise tous les paramètres du générateur
 * @param seed graine
 * @param a multiplicateur
 * @param c incrément
 * @param m module
 */
GeneCongruentiel::GeneCongruentiel(int seed, int a, int c, int m) : _seed(seed), _X(seed), _a(a), _c(c), _m(m){}


/**
 * @return multiplicateur du générateur
 */
int GeneCongruentiel::getA() const {
    return _a;
}

/**
 * @return incrément du générateur
 */
int GeneCongruentiel::getC() const {
    return _c;
}

/**
 * @return module du générateur
 */
int GeneCongruentiel::getM() const {
    return _m;
}

/**
 * @return graine du générateur
 */
int GeneCongruentiel::getSeed() const {
    return _seed;
}

/**
 * Renvoie un nombre pseudo aléatoire calculé à partir
 * du dernier nombre pseudo-aléatoire généré
 * @return nombre pseudo aléatoire
 */
int GeneCongruentiel::genNb() {
```

```
55      _X = (_a*_X+_c)%_m;
56      return _X;
57  }
58
59  /**
60   * Renvoie la suite des l premiers nombres pseudo-aléatoires généré
       s par :
61   * @param gene générateur congruentiel linéaire
62   * @param l nombres d'éléments de la suite
63   * @return suite de nombres pseudo-aléatoires
64   */
65  std::vector<int> makeSequence(GeneCongruentiel gene, int l) {
66      std::vector<int> sequence;
67      gene._X = gene.getSeed();
68      sequence.push_back(gene.getSeed());
69      for(int i=1; i<l; i++)
70          sequence.push_back(gene.genNb());
71      return sequence;
72  }
73
74  /**
75   * Renvoie la suite de tous les nombres pseudo-aléatoires diffé
       rents générés par :
76   * @param gene générateur congruentiel linéaire
77   * @return suite de nombres pseudo-aléatoires (1 période)
78   */
79  std::vector<int> getStats(GeneCongruentiel gene) {
80      std::vector<int> sequence = makeSequence(gene, gene.getM());
81      std::vector<int> res;
82      res.push_back(sequence.at(0));
83      res.push_back(sequence.at(1));
84      bool find = false;
85      int i=2;
86      do {
87          if(sequence.at(i)!=sequence.at(0) && sequence.at(i)!=
       sequence.at(1))
88              res.push_back(sequence.at(i));
89          else
90              find = true;
91          i++;
92      }while(!find && i<gene.getM());
93      return res;
94  }
```

```
/*
* @File:    superGeneCongruentiel.h
* @Project: Générateur de nombres pseuso-aléatoires - Mathématiques ESIREM 3A IT
* @Author:  Allan Coutarel, Valentin Cheny, Guillaume Imhoff
*/

#ifndef _SUPGENECONGRU_
#define _SUPGENECONGRU_

#include <iostream>
#include <vector>
#include "geneCongruentiel.h"

class SuperGeneCongruentiel {
    public:
        SuperGeneCongruentiel(GeneCongruentiel geneChoix);
        GeneCongruentiel getGeneChoix() const;
        void addGene(GeneCongruentiel nouveauGene);
        GeneCongruentiel getGene(int n) const;
        int genNb();
        friend std::vector<int> makeSequence(SuperGeneCongruentiel gene, int l);

    private:
        GeneCongruentiel _geneChoix;
        std::vector<GeneCongruentiel> _listGene;
    };

void periodeSGC(SuperGeneCongruentiel gene, int p_max);

#endif
```

```cpp
/*
* @File:    superGeneCongruentiel.cpp
* @Project: Générateur de nombres pseuso-aléatoires - Mathématiques ESIREM 3A IT
* @Author:  Allan Coutarel, Valentin Cheny, Guillaume Imhoff
*/

#include <iostream>
#include "superGeneCongruentiel.h"

/**
 * Constructeur standard d'un super générateur congruentiel
 * Initialise tous les paramétres du générateur
 * @param nouveauGene : définit le générateur congruentiel linéaire utilisé pour choisir entre les générateurs fournis
 */
SuperGeneCongruentiel::SuperGeneCongruentiel(GeneCongruentiel geneChoix) : _geneChoix(geneChoix) {}

/**
 * Retourne _geneChoix
 * @return générateur congruentiel linéaire permettant de choisir
 */
GeneCongruentiel SuperGeneCongruentiel::getGeneChoix() const {
    return _geneChoix;
}

/**
 * Ajouter un générateur supplémentaire dans la liste des générateurs utilisés
 * @param nouveauGene générateur congruentiel linéaire à ajouter
 */
void SuperGeneCongruentiel::addGene(GeneCongruentiel nouveauGene) {
    _listGene.push_back(nouveauGene);
}

/**
 * Retourne le n ième générateur utilisé
 * @param nouveauGene générateur congruentiel linéaire à ajouter
 * @return n ième générateur
 */
GeneCongruentiel SuperGeneCongruentiel::getGene(int n) const {
    return _listGene[n];
}

/**
 * Renvoie un nombre pseudo aléatoire calculé à partir
 * des derniers nombre pseudo-aléatoire généré par chaque sous-générateur
 * @return nombre pseudo aléatoire
 */
int SuperGeneCongruentiel::genNb() {
    return _listGene[_geneChoix.genNb() % _listGene.size()].genNb();
}

```

```cpp
/**
 * Renvoie la suite des l premiers nombres pseudo-aléatoires générés par :
 * @param gene super générateur congruentiel linéaire
 * @param l nombres d'éléments de la suite
 * @return suite de nombres pseudo-aléatoires
 */
std::vector<int> makeSequence(SuperGeneCongruentiel gene, int l) {
    std::vector<int> sequence;
  size_t L = l;
    for (size_t i = 1; i < L; i++)
        sequence.push_back(gene.genNb());
    return sequence;
}

bool presentHere(int n, std::vector<int> liste1, std::vector<int> liste2) {
  // test si la liste 1 est dans la liste 2 à la position n
  for (size_t i = 0; i < liste1.size(); i++) {
    if (liste1[i] != liste2[i + n])
      return false;
  }
  return true;
}

void periodeSGC(SuperGeneCongruentiel gene, int p_max) {

  std::vector<int> suite_debut = makeSequence(gene, 10);
  std::vector<int> suite = makeSequence(gene, p_max);

  for (size_t i = 0; i < suite.size(); i++) {
    if (presentHere(i, suite_debut, suite))
    {
      std::cout << i << std::endl;
    }
  }
}
```

```cpp
/*
* @File:    main.cpp
* @Project: Générateur de nombres pseuso-aléatoires - Mathématiques ESIREM 3A IT
* @Author:  Allan Coutarel, Valentin Cheny, Guillaume Imhoff
*/

#include <iostream>
#include "geneCongruentiel.h"
#include "superGeneCongruentiel.h"
#include <fstream>
#include <algorithm>
#include <vector>

//ICI IL S'AGIT D'UN CODE D'EXEMPLE D'IMPLEMENTATION
//ON PEUT BIEN SUR L'ADAPTER A NOS BESOINS

//Exemple d'utilisation du générateur congruentiel linéaire
void exImpGC() {
  int taille_periode = 0;
  GeneCongruentiel gene(30, 12, 0, 31);
  std::cout << gene.genNb() << std::endl;
  std::cout << gene.genNb() << std::endl;
  std::cout << gene.genNb() << std::endl;
  std::vector<int> suite = makeSequence(gene, 100);
  std::vector<int> stats = getStats(gene);
  for (int i : suite)
    std::cout << i << " ";
  std::ofstream data("data.csv");
  for (int i : stats) {
    data << i << std::endl;
    taille_periode++;
  }
  std::cout << std::endl;
  std::cout << "La taille de la période du générateur est : " << taille_periode << std::endl;
}

//Construction du super générateur
SuperGeneCongruentiel exempleSGC() {
  GeneCongruentiel geneChoix(34, 13, 349, 509);
  GeneCongruentiel gene1(122, 367, 127, 509);
  GeneCongruentiel gene2(347, 257, 97, 509);
  GeneCongruentiel gene3(502, 53, 211, 509);
  GeneCongruentiel gene4(107, 47, 17, 509);

  SuperGeneCongruentiel superGene(geneChoix);
  superGene.addGene(gene1);
  superGene.addGene(gene2);
  superGene.addGene(gene3);
  superGene.addGene(gene4);

  return superGene;
}

//Exemple d'utilisation du super générateur congruentiel linéaire
```

```
55 void testsSGC() {
56   auto SGC = exempleSGC();
57   // obtenir la période
58   periodeSGC(SGC, 300000);
59 
60   // exemples de valeurs
61   auto suite = makeSequence(SGC, 200);
62   for (auto x : suite)
63     std::cout << x << std::endl;
64 
65   //enregistrer les valeurs
66   std::ofstream data("data.csv");
67   for (int i : suite)
68     data << i << std::endl;
69 }
70 
71 int main(){
72   exImpGC();
73   //testsSGC();
74   return 0;
75 }
```

```
# Makefile with dependencies management (Q4)

CXX         = g++
CXXFLAGS    = -Wall  -std=c++17
DEPFLAGS    = -MMD
LDFLAGS     =
SRCS        = $(wildcard *.cpp)
OBJS        = $(SRCS:.cpp=.o)
TARGET      = app
DEPS      = $(OBJS:.o=.d)

all: $(TARGET)

$(TARGET): $(OBJS)
  $(CXX) $(CXXFLAGS) $(LDFLAGS) -o $(TARGET) $(OBJS)

%.o: %.cpp
  $(CXX) $(CXXFLAGS) $(DEPFLAGS) -c  $<

clean:
  rm -f *.o *.d

mrproper: clean
  rm -f $(TARGET)

exe: $(TARGET)
  ./$(TARGET)

-include $(DEPS)
```

Loi du khi-deux avec k degrés de liberté
Quantiles d'ordre $1-\gamma$

	γ										
k	0.995	0.990	0.975	0.950	0.900	0.500	0.100	0.050	0.025	0.010	0.005
1	0.00	0.00	0.00	0.00	0.02	0.45	2.71	3.84	5.02	6.63	7.88
2	0.01	0.02	0.05	0.10	0.21	1.39	4.61	5.99	7.38	9.21	10.60
3	0.07	0.11	0.22	0.35	0.58	2.37	6.25	7.81	9.35	11.34	12.84
4	0.21	0.30	0.48	0.71	1.06	3.36	7.78	9.94	11.14	13.28	14.86
5	0.41	0.55	0.83	1.15	1.61	4.35	9.24	11.07	12.83	15.09	16.75
6	0.68	0.87	1.24	1.64	2.20	5.35	10.65	12.59	14.45	16.81	18.55
7	0.99	1.24	1.69	2.17	2.83	6.35	12.02	14.07	16.01	18.48	20.28
8	1.34	1.65	2.18	2.73	3.49	7.34	13.36	15.51	17.53	20.09	21.96
9	1.73	2.09	2.70	3.33	4.17	8.34	14.68	16.92	19.02	21.67	23.59
10	2.16	2.56	3.25	3.94	4.87	9.34	15.99	18.31	20.48	23.21	25.19
11	2.60	3.05	3.82	4.57	5.58	10.34	17.28	19.68	21.92	24.72	26.76
12	3.07	3.57	4.40	5.23	6.30	11.34	18.55	21.03	23.34	26.22	28.30
13	3.57	4.11	5.01	5.89	7.04	12.34	19.81	22.36	24.74	27.69	29.82
14	4.07	4.66	5.63	6.57	7.79	13.34	21.06	23.68	26.12	29.14	31.32
15	4.60	5.23	6.27	7.26	8.55	14.34	22.31	25.00	27.49	30.58	32.80
16	5.14	5.81	6.91	7.96	9.31	15.34	23.54	26.30	28.85	32.00	34.27
17	5.70	6.41	7.56	8.67	10.09	16.34	24.77	27.59	30.19	33.41	35.72
18	6.26	7.01	8.23	9.39	10.87	17.34	25.99	28.87	31.53	34.81	37.16
19	6.84	7.63	8.81	10.12	11.65	18.34	27.20	30.14	32.85	36.19	38.58
20	7.43	8.26	9.59	10.85	12.44	19.34	28.41	31.41	34.17	37.57	40.00
21	8.03	8.90	10.28	11.59	13.24	20.34	29.62	32.67	35.48	38.93	41.40
22	8.64	9.54	10.98	12.34	14.04	21.34	30.81	33.92	36.78	40.29	42.80
23	9.26	10.20	11.69	13.09	14.85	22.34	32.01	35.17	38.08	41.64	44.18
24	9.89	10.86	12.40	13.85	15.66	23.34	33.20	36.42	39.36	42.98	45.56
25	10.52	11.52	13.12	14.61	16.47	24.34	34.28	37.65	40.65	44.31	46.93
26	11.16	12.20	13.84	15.38	17.29	25.34	35.56	38.89	41.92	45.64	48.29
27	11.81	12.88	14.57	16.15	18.11	26.34	36.74	40.11	43.19	46.96	49.65
28	12.46	13.57	15.31	16.93	18.94	27.34	37.92	41.34	44.46	48.28	50.99
29	13.12	14.26	16.05	17.71	19.77	28.34	39.09	42.56	45.72	49.59	52.34
30	13.79	14.95	16.79	18.49	20.60	29.34	40.26	43.77	46.98	50.89	53.67
40	20.71	22.16	24.43	26.51	29.05	39.34	51.81	55.76	59.34	63.69	66.77
50	27.99	29.71	32.36	34.76	37.69	49.33	63.17	67.50	71.42	76.15	79.49
60	35.53	37.48	40.48	43.19	46.46	59.33	74.40	79.08	83.30	88.38	91.95
70	43.28	45.44	48.76	51.74	55.33	69.33	85.53	90.53	95.02	100.42	104.22
80	51.17	53.54	57.15	60.39	64.28	79.33	96.58	101.88	106.63	112.33	116.32
90	59.20	61.75	65.65	69.13	73.29	89.33	107.57	113.14	118.14	124.12	128.30
100	67.33	70.06	74.22	77.93	82.36	99.33	118.50	124.34	129.56	135.81	140.17

Si k est entre 30 et 100 mais n'est pas un multiple de 10, on utilise la table ci-haut et on fait une interpolation linéaire. Si $k > 100$ on peut, grâce au théorème limite central, approximer la loi $\chi^2(k)$ par la loi $N(k, 2k)$.