RÈGLES POLYNOMIALES (REGPOLY)

Définition fonctionnelle et guide d'utilisation

Août 2000

François Panneton

Département d'I.R.O., Université de Montréal

Résumé

REGPOLY est un progiciel permettant de vérifier certains critères, dont l'équidistribution, de générateurs combinés utilisant une récurrence linéaire modulo 2.

Table des matières

1 Introduction

Le progiciel REGPOLY a été développé afin d'étudier les propriétés de générateurs de nombres aléatoires utilisant une récurrence linéaire modulo 2. Pour ces générateurs à récurrence linéaire modulo 2, l'espace des états est $\mathbb{F}_2^k$. Un élément de cet ensemble peut être représenté par un vecteur de k bits.

Les générateurs de ce type déjà existants sont les générateurs de Tausworthe [9, 2, 3], les TGFSR [5, 6], les GCL polynomiaux [4] et les Mersenne Twister [7]. Tous ces générateurs utilisent, sous des formes différentes, le même type de récurrence linéaire et peuvent tous être représentés sous une forme matricielle. Si on représente l'état courant par un vecteur colonne de k bits $\mathbf{x}_{n-1} = (x_{n-1}^{(0)}, \ldots, x_{n-1}^{(k-1)})^T$, alors le prochain état peut être obtenu par l'équation, dans $\mathbb{F}_2$,

$$\mathbf{x}_n = X\mathbf{x}_{n-1} \tag{1}$$

où X est une matrice inversible $k \times k$ dont chaque élément est un bit et les opérations implicites à la multiplication d'une matrice sont faites dans $\mathbb{F}_2$. On obtient la sortie par les équations suivantes :

$$\mathbf{z}_n = B\mathbf{x}_n, \tag{2}$$

$$\mathbf{y}_n = Y\mathbf{z}_n \tag{3}$$

et

$$u_n = \sum_{i=1}^{L} 2^{-i} y_n^{(i-1)} \tag{4}$$

où $\mathbf{y}_n = (y_n^{(0)}, \ldots, y_n^{(L-1)})^T$, B est une matrice $k \times k$, Y est une matrice $L \times k$ et L est la résolution que l'on veut avoir pour la sortie u_n qui est comprise dans l'intervalle [0,1). La matrice Y décide de la résolution de la sortie $\mathbf{y}_n$. Si $k \leq L$, alors la matrice Y est composée des L premières lignes de I_k, la matrice identité $k \times k$. Sinon elle est de la forme $(I_k \;\; C)^T$ où C est une matrice $(L-k) \times k$ qui définit les $L-k$ derniers bits de $\mathbf{y}_n$.

Dans les faits, la forme de la matrice X détermine le type de générateur et B représente une transformation linéaire appliquée à la sortie du générateur.

En observant les équations (1), (2), (3) et (4), on remarque que l'on peut obtenir différentes suites de nombres $u_0, u_1, \ldots$ avec la même matrice X en utilisant différentes matrices B. Cette observation se révèle importante lorsqu'on désire améliorer ou changer les propriétés des générateurs.

Le progiciel REGPOLY permet de vérifier différents critères sur des générateurs combinés qui utilisent une récurrence linéaire modulo 2. Pour combiner des générateurs de ce type, il s'agit de faire fonctionner les générateurs en parallèle et, à chaque itération, on produit une sortie à l'aide des sorties de chacun des générateurs.

Afin de bien définir les générateurs combinés, il faut introduire une certaine notation. Alors, définissons, à la n-ième itération, l'état de la j-ième composante $\mathbf{x}_{n,j}$ =

$(x_{n,j}^{(0)}, \ldots, x_{n,j}^{(k_j-1)})$, le vecteur $\mathbf{z}_{n,j} = (z_{n,j}^{(0)}, \ldots, z_{n,j}^{(k_j-1)})$ où k_j est le degré du polynôme caractéristique de la composante j et $\mathbf{y}_{n,j} = (y_{n,j}^{(0)}, \ldots, y_{n,j}^{(L_j-1)})$ où L_j est la résolution de la composante j. Pour la composante j, il faut également définir B_j, une matrice $k_j \times k_j$ de plein rang et Y_j, une matrice $L_j \times k_j$. On définit ensuite pour chacune des composantes

$$\mathbf{z}_{n,j} = B_j \mathbf{x}_{n,j}$$

et

$$\mathbf{y}_{n,j} = Y_j \mathbf{z}_{n,j}.$$

On obtient le vecteur de sortie du générateur combiné $\mathbf{y}_n$ par

$$\mathbf{y}_n = \mathbf{y}_{n,1} \oplus \cdots \oplus \mathbf{y}_{n,J}. \tag{5}$$

La valeur de L est définie par

$$L = \min \{L_j \mid 1 \leq j \leq J\}. \tag{6}$$

La sortie u_n est déterminée par la formule habituelle (4).

Supposons J générateurs distincts. La période du générateur combiné est

$$\text{ppcm}(\rho_1, \ldots, \rho_J). \tag{7}$$

Il est à remarquer que ces générateurs n'ont pas qu'un seul cycle. Afin de s'assurer que le générateur combiné progresse dans le cycle le plus long, où la période maximale est donnée par (7), il est nécessaire d'avoir $\mathbf{x}_{0,j} \neq \mathbf{0}$, pour $1 \leq j \leq J$, où $\mathbf{0}$ est le vecteur de bits ne contenant que des bits nuls. Dans le cas contraire, si pour la composante j' on a $\mathbf{x}_{0,j'} = \mathbf{0}$, alors on obtient le même générateur combiné avec les mêmes composantes sauf la composante j'.

On observe aussi que pour un générateur qui utilise k bits pour représenter son état, alors on pourrait s'attendre à ce que celui-ci ait une période maximale, pour le plus long cycle, de $2^k - 1$. Ce n'est pas le cas du générateur combiné, qui utilise $k = \sum_{j=1}^{J} k_j$ bits, puisque son cycle le plus long a une période de $(2^{k_1} - 1) \times \cdots \times (2^{k_J} - 1)$ qui n'est pas égale à $2^k - 1$.

Raison d'être de REGPOLY

Le progiciel REGPOLY a plusieurs fonctionnalités. En effet, il permet d'effectuer les tâches suivantes :

1. Vérifier l'équidistribution de n'importe quel type de générateur combiné à récurrence linéaire modulo 2 (pas seulement les générateurs de Tausworthe combinés et les TGFSR combinés).
2. Vérifier l'équidistribution des projections (voir la description du module tuplets pour plus de détails) pour n'importe quel type de générateur combiné.

3. Vérifier la propriété CF (voir la description du module mecf pour plus de détails) si le générateur est ME, pour n'importe quel type de générateur combiné.
4. Vérifier des générateurs combinés dont chaque composante peut être de type différent.
5. Appliquer des transformations linéaires sur la sortie de chacune des composantes du générateur combiné.
6. Utiliser un algorithme d'optimisation du tempering de Matsumoto-Kurita efficace pour n'importe quel type de générateur combiné.
7. Dans sa version présente, implanter trois types de transformations linéaires et trois types de générateurs de nombres aléatoires, mais permet d'ajouter de nouvelles transformations linéaires ou de nouveaux générateurs à cause du design modulaire du progiciel.
8. Le module qui s'occupe des générateurs de Tausworthe ne permet pas seulement des trinômes, mais aussi, des polynômes ayant plus que trois coefficients non nuls.
9. Permettre la création de nouveaux critères de sélection de générateur combiné.

Le progiciel REGPOLY est en fait un ensemble de facilités, codées à l'avance, qui permettent à quiconque d'écrire des programmes pour chercher de bons générateurs. Le programme recherche.c, que l'on peut retrouver dans ce guide d'utilisation, essaie de tirer au maximum les capacités présentes du progiciel. Les générateurs trouvés par le programme recherche.c peuvent être utilisés pour la simulation Monte Carlo et, avec de petits générateurs utilisés sur toute leur période, pour l'intégration quasi-Monte Carlo.

2 Une vue d'ensemble du progiciel

Le progiciel REGPOLY permet de vérifier différents critères sur des générateurs combinés à récurrence modulo 2. Ces différents critères sont expliqués en détail dans la description de leur module respectif.

Le langage de programmation de REGPOLY est le langage C. Étant donné que langage est connu par la plupart des programmeurs, ceci permet à plus de monde de l'utiliser sans les désavantages de l'apprentissage d'un nouveau langage de programmation. Aussi, le langage C permet une grande portabilité entre différentes plateforme. Le progiciel a été développé sur une plateforme Linux.

Le progiciel est divisé en modules. Il y a les modules btypes, etages, lintrans et baseopera qui constituent le coeur du progiciel. Il y a aussi des modules pour chacun des critères à vérifier (mecf,tuplets,..), pour chacun des types de générateurs (tgfsr, polynomial, tausworthe,...) et chacune des transformations linéaires (permut, temperMK, selft,...). Mais, avant d'entrer dans les détails de chacun des modules, il est important de comprendre l'algorithme pour lequel ce progiciel a été conçu.

Le progiciel permet de vérifier des critères sur des générateurs simples ou à J composantes. Celui-ci construit une structure à J étages où un étage contient toute l'information nécessaire sur les générateurs à tester comme composante j. Dans le cas des générateurs simples, alors on ne construit qu'un seul étage. Si N_j est le nombre de générateurs que l'on veut vérifier à l'étage j, alors

$$\mathcal{E}_j = \{G_{i,j} | 1 \leq i \leq N_j\} \ , \ 1 \leq j \leq J$$

est l'ensemble des générateurs qui seront testés comme composante j (ici $G_{i,j}$ est le i-ième générateur de la composante j à vérifier). Cet ensemble constitue l'étage j. On construit, à partir des ensembles $\mathcal{E}_j$, l'ensemble des générateurs combinés pour lesquels on veut vérifier un certain critère. Cet ensemble est

$$\mathcal{C} = \mathcal{E}_1 \times \cdots \times \mathcal{E}_J \ / \ \{(G_1, \ldots, G_J) | \text{Période}(G_i) = \text{Période}(G_j), i \neq j, 1 \leq i, j \leq J\}.$$

On n'accepte donc pas les générateurs combinés pour lesquels deux composantes ont la même période. Aussi, il est préférable que les périodes des générateurs n'aient pas de facteurs communs afin que la période du générateur combiné, soit $\text{ppcm}(\rho_1, \ldots, \rho_J)$, soit la plus grande possible (voir équation 7), mais le progiciel ne fait pas cela pour le moment.

Il est possible, avec le progiciel REGPOLY, d'appliquer des transformations linéaires B_j à la sortie de chacune des composantes. À l'étage j, on peut définir un certain nombre, M_j, de transformations linéaires à la sortie de la composante j. On définit

$$\mathcal{F}_j = \{B_{1,j}, B_{2,j}, \ldots, B_{M_j,j}\},$$

l'ensemble de toutes les transformations linéaires qui doivent être appliquées à la sortie de la composante j.

Voici donc l'algorithme de recherche général :

1. Définir les ensembles $\mathcal{E}_j$ pour $1 \leq j \leq J$.
2. Définir les ensembles $\mathcal{F}_j$ pour $1 \leq j \leq J$.
3. Construire l'ensemble $\mathcal{C}$.
4. $C \leftarrow$ le premier élément de $\mathcal{C}$.
5. Vérifier les critères désirés sur le générateur combiné C en prenant soin d'appliquer les transformations linéaires contenues dans $\mathcal{F}_j$ à la sortie de la j-ième composante pour $1 \leq j \leq J$.
6. $C \leftarrow$ le prochain élément de $\mathcal{C}$.
7. Retourner à l'étape 5

On remarque que dans le cadre de cet algorithme, on applique toujours les mêmes transformations linéaires pour chacun des générateurs combinés C. Une autre possibilité est d'essayer plusieurs transformations linéaires sur le même générateur combiné. Au lieu d'utiliser $\mathcal{F}_j$, on utilise

$$\mathcal{F}_j^{C,r},$$

l'ensemble des transformations linéaires qui sont appliquées à la j-ième composante du générateur combiné C au r-ième essai. Le nombre d'essais avec le même générateur combiné est R. L'algorithme devient alors le suivant :

1. Définir les ensembles $\mathcal{E}_j$ pour $1 \leq j \leq J$.
2. Définir les ensembles $\mathcal{F}_j^{C,r}$ pour $1 \leq j \leq J$, pour tout $C \in \mathcal{C}$ et $1 \leq r \leq R$.
3. Construire l'ensemble $\mathcal{C}$.
4. $C \leftarrow$ le premier élément de $\mathcal{C}$.
5. $r \leftarrow 1$.
6. $\mathcal{F}_j \leftarrow \mathcal{F}_j^{C,r}, 1 \leq j \leq J$.
7. Vérifier les critères désirés sur le générateur combiné C en prenant soin d'appliquer les transformations linéaires contenues dans $\mathcal{F}_j$ sur la sortie de la j-ième composante pour $1 \leq j \leq J$.
8. si $r < R$, alors $r \leftarrow r + 1$ et aller à l'étape 6.
9. $C \leftarrow$ le prochain élément de $\mathcal{C}$. Retourner à l'étape 5 si on n'a pas vérifié tous les générateurs combinés possibles.

À l'étape 2, on écrit qu'il faut définir les ensembles $\mathcal{F}_j^{C,r}$. Une façon d'y arriver, sans utiliser trop d'espace mémoire, est de produire ces ensembles à l'exécution du programme. Puisque chaque ensemble n'est utilisé qu'une seule fois, alors on peut les produire au besoin et les effacer aussitôt qu'on en a terminé. Les paramètres des transformations linéaires contenus dans $\mathcal{F}_j^{C,r}$ peuvent être choisis au hasard, ce qui simplifie la génération de $\mathcal{F}_j^{C,r}$.

On peut vouloir n'effectuer qu'une seule transformation linéaire sur le générateur combiné. Cela revient à faire la même transformation linéaire sur chacune des composantes du générateur. (Voir section 4.4 de [8] pour plus de détail)

Une description brève de chacun des modules

Voici maintenant une description de chacun des modules de REGPOLY. Chacun de ceux-ci contient des facilités pour la recherche de générateurs à récurrence linéaire modulo 2.

1. **btypes**. Dans ce module on définit certaines constantes et types de variables qui sont utilisés dans le progiciel REGPOLY. Il peut être nécessaire d'aller ajuster certains paramètres dans celui-ci afin d'augmenter la période maximale permise par le progiciel.
2. **baseopera**. Module permettant de faire les opérations de base sur les vecteurs de bits et les matrices de bits. Il contient également d'autres fonctions utilitaires diverses ainsi que la fonction `Init_REGPOLY()` qu'il est essentiel d'appeler avant d'utiliser n'importe quel module du progiciel.
3. **etages**. Ce module permet de gérer la structure à J étages qui contient les informations sur la recherche de paramètres des générateurs (J est le nombre de composantes du générateur combiné). Ce module contient les fonctions nécessaires afin de passer d'un générateur combiné à un autre.
4. **lintrans**. Ce module permet de gérer les transformations linéaires appliquées à la sortie des composantes du générateur combiné. Il contient également les fonctions nécessaires afin de passer d'une configuration de paramètres de transformations linéaires à une autre.
5. **mecf**. Module permettant de calculer l'équidistribution des générateurs combinés et la propriété "collision-free".
6. **tuplets**. Module permettant de calculer des critères concernant les projections.
7. **tausworthe**, **polynomial** et **tgfsr**. Ces modules contiennent des implantations génériques des générateurs TGFSR, GCL polynomial et Tausworthe, respectivement. Également, il y a des fonctions utilitaires se rapportant à chacun de ces types de générateurs à l'intérieur de ces modules.
8. **selft**, **permut** et **temperMK**. Ces modules contiennent des implantations génériques des transformations linéaires de types self-tempering, permutation et tempering de Matsumoto-Kurita. Également, il y a des fonctions utilitaires se rapportant à chacune de ces transformations linéaires à l'intérieur de ces modules.
9. **timer**. Module permettant de calculer le temps. Peut être utile si on désire calculer les performances en temps de CPU.

3 Un exemple de programme utilisant REGPOLY

Ici on présente un exemple d'un programme qui a été écrit grâce au progiciel REGPOLY. Celui-ci a été utilisé dans [8] afin de trouver de bons générateurs de type GCL polynomial ainsi que des TGFSR combinés à deux ou trois composantes. Afin de mieux comprendre cette partie, il est conseillé d'avoir lu préalablement [8]. Le programme s'appelle *recherche* et permet de rechercher des générateurs de nombres aléatoires qui ont les critères voulus d'équidistribution et/ou montrent une bonne équidistribution dans les projections.

Le code du programme principal est contenu dans le fichier recherche.c et le fichier util.c est nécessaire pour la lecture des données dans le fichier principal de données et dans les fichiers de transformations linéaires. Ces fichiers et le fichier de compilation (Makefile) sont disponibles à l'annexe B.

3.1 Fichier principal de données

Afin de faire fonctionner le programme correctement, il est important de comprendre la structure des fichiers d'entrée. Le premier fichier est le fichier principal de données. La structure de ce fichier est décrite dans la figure 1.

```
J                    # Nombre de composante au générateur combiné
seed1 [seed2]        # Racine du générateur pour les transformations linéaires
c1 L1 file1          # Type de générateur, L, et fichier pour la composante 1
t1 [trans1]          # Si t1=1,  fichier trans. lin. est trans1 pour comp. 1
..                   #
cJ LJ fileJ          # Type de générateur, L, et fichier pour la composante J
tJ [transJ]          # Si tJ=1, fichier trans. lin. est trans1 pour comp. J
nbessais             # Nombre d'essais de trans. lin. par générateur combiné
mse                  # Maximum sur la somme des delta_l pour l dans Psi_12
nb_lignes            # Nombre de lignes pour définir des valeurs max. de delta_l
[min max ecart]      # delta_l <= ecart pour l=min..max
..                   #
[min max ecart]      # delta_l <= ecart pour l=min..max
T [d t1 t2 .. td mDelta]   # On selectionne si Delta(t1, t2, .., td) <= mDelta
```

FIG. 1 – Fichier principal de données.

Voici la signification des symboles de la figure 1 (Les accolades signifient que les valeurs qu'elles contiennent ne sont pas toujours nécessaires) :

`J`	Nombre de composantes au générateur combiné.
`seed1` `seed2`	Racine du générateur de nombres aléatoires `CombLec88()`, définies dans le module baseopera, permettant de générer des paramètres de transformations linéaires au hasard. `seed1` doit être compris entre 1 et 2147483562 et `seed2` entre 1 et 2147483398. Si `seed1` est -1, alors la racine est déterminée par l'horloge de l'ordinateur et il est inutile

	de définir `seed2`.
`cj`	Type de générateur de la composante `j` si `cj` est “taus”, la composante `j` est un générateur de Tausworthe, si `cj` est “tgfsr”, la composante `j` est un TGFSR, si `cj` est “polynomial”, la composante `j` est un GCL polynomial.
`Lj`	Indique le nombre de bits de résolution de la composante `j` à la sortie.
`filej`	Nom du fichier contenant les détails du générateur de la composante `j`. Si `filej` est “same”, alors le fichier de la composante `j` est le même que celui de la composante `j`-1.
`tj`	Prend 0 ou 1 comme valeur. Si `tj` est 1, alors on effectue des transformations linéaires sur la composante `j`.
`transj`	À définir seulement si `tj` est 1. Il s'agit du nom du fichier qui contient les détails des transformations linéaires à faire sur la composante `j`. (Voir section 3.2 pour la description de ces fichiers)
`nbessais`	Est utilisé seulement s'il existe un `tj` égal à 1. Indique le nombre de fois que le programme essaie des transformations linéaires différentes avec le même générateur combiné.
`mse`	Valeur maximale que l'on peut tolérer pour $\sum_{\ell \in \Psi_{12}} \Delta_\ell$.(Voir module mecf) Si `mse` est -1, alors on ne vérifie pas l'équidistribution.
`nb_lignes`	Indique le nombre de lignes sur lesquelles on spécifie des tolérances δ_ℓ
`min`	
`max`	
`ecart`	Indique les valeurs de ℓ, telles que δ_ℓ=`ecart` (`min` $\leq \ell \leq$ `max`) (Voir module mecf)
`T`	Prend comme valeur 0 ou 1. Si `T` est 1, ceci indique qu'on vérifie le critère $\Delta_{t_1,\ldots,t_d}$.(Voir module tuplets)
`d t1 .. td`	À définir seulement si `T` est 1, paramètres du critère $\Delta_{\texttt{t1},..,\texttt{td}}$.
`mDelta`	À définir seulement si `T` est 1. Valeur maximum que l'on peut tolérer pour $\Delta_{\texttt{t1},..,\texttt{td}}$.

3.2 Fichier de transformations linéaires

Il existe un autre type de fichier qui doit être spécifié, si l'on désire appliquer des transformations linéaires sur une ou plusieurs composantes. Il s'agit des fichiers `transj` auxquels on fait référence dans l'explication du fichier principal de données. Ce type de fichier permet de déterminer les paramètres des transformations linéaires à appliquer sur **une** composante. La figure 2 décrit le format du fichier pour les transformations linéaires à appliquer sur une composante.

Voici maintenant des explications des symboles de la figure 2. L'entier `N` est le nombre de transformations linéaires que l'on désire appliquer à la composante, `ci` est une chaîne de caractères qui détermine la transformation linéaire à appliquer. Voici un tableau des différentes

```
N
c1 p11 p12 ..
c2 p21 p22 ..
..
cN pN1 pN2 ..
```

Fig. 2 – Fichier de transformations linéaires.

valeurs possibles de `ci` :

``permut''	permutation des coordonnées
``tempMK''	tempering de Matsumoto-Kurita
``tempMKopt''	tempering de Matsumoto-Kurita que l'on veut optimiser (permis seulement si c'est la dernière transformation linéaire à être effectuée sur cette composante)
``selft''	self-tempering

Les valeurs `p11`, `p12`,... à la figure 2 sont les paramètres de chacune des transformations. Voici une brève explication de la syntaxe à utiliser pour chacune des transformations linéaires possibles.

Permutation (voir module permut)
syntaxe :
`permut p q`

Met p=`p` et q=`q`. Si `p` est -1 et/ou `q` est -1, alors les valeurs de p et/ou q sont choisies au hasard pour la permutation.

Tempering de Matsumoto-Kurita (voir module temperMK)
syntaxe :
`tempMK eta mu alea`

μ=`mu`, η=`eta` et si `alea` est -1, alors les valeurs des vecteurs de bits $\mathbf{b}$ et $\mathbf{c}$ sont choisies au hasard. Sinon `alea` indique le nombre de mots de 32 bits permettant de définir les vecteurs de bits $\mathbf{b}$ et $\mathbf{c}$. Dans ce cas, on définit $\mathbf{b}$ sur une ligne et $\mathbf{c}$ sur la suivante en valeur hexadécimale. Par exemple, si `alea` est 2, alors on définit $\mathbf{b}$ et $\mathbf{c}$ par 2 mots de 32 bits.

exemple 1 :
`tempMK 7 15 -1`

Pour cet exemple, $\mathbf{b}$ et $\mathbf{c}$ sont choisis au hasard. On fixe η=7 et μ=15.

exemple 2 :
`tempMK 15 31 2`
`e56d66a1 8eff77f7`← (vecteur $\mathbf{b}$)
`7733344d 44d44ead`← (vecteur $\mathbf{c}$)

Tempering de Matsumoto-Kurita avec optimisation (voir module temperMK)

Si l'on désire optimiser le tempering de Matsumoto-Kurita, il suffit de changer le mot `tempMK` pour le mot `tempMKopt` et de rajouter deux paramètres : `disp` et `maxv`. Si `disp`$\neq 0$, alors il y aura un affichage indiquant où le programme est rendu dans l'optimisation. Le paramètre `maxv` indique que l'on veut optimiser de telle sorte que $\Delta_\ell \leq \delta_\ell$ pour $0 \leq \ell \leq$ `maxv`. Pour le générateur combiné, $L = \min\{L_1, \ldots, L_J\}$. La valeur de `maxv` sera la plus petite valeur entre les valeurs de `maxv` de chacune des composantes où on veut optimiser le tempering de Matsumoto-Kurita et la valeur de L.

syntaxe :
`tempMKopt eta mu alea disp maxv`

exemple 3 :
`tempMKopt 7 15 -1 1 32`

Pour cet exemple, on optimise jusqu'au 32^e bit (`maxv`=32) et on affiche à l'écran les progrès de l'optimisation (`disp`=1). On a également $\eta = 7$ et $\mu = 15$. Les valeurs de **b** et **c** sont fixées au hasard (`alea` est -1).
Self-tempering (voir module selft)
syntaxe :
`selft d`

Cette ligne permet d'effectuer un Self-tempering avec d=`d`. Si `d`$=-1$, alors le paramètre d est choisi au hasard.

3.3 Exemples de recherche

Afin de mieux comprendre la structure de ces fichiers, cette section donne quelques exemples de recherches de paramètres de générateurs.

Exemple 1. On désire obtenir des GCL polynomiaux à une seule composante qui ont un polynôme caractéristique de degré 32. Le fichier `32poly.dat` contient les informations sur les polynômes à vérifier (dans la description du module polynomial en annexe, le format de ce fichier y est expliqué). Pour cette recherche, on a que $L = k = 32$. On veut que $\sum_{\ell \in \Psi_{12}} \Delta_\ell \leq 3$. On applique une permutation des coordonnées avec p et q choisis au hasard et un tempering de Matsumoto-Kurita avec $\eta = 7$ et $\mu = 15$ (Le programme ne permet pas de mettre η et μ au hasard). On désire que le programme optimise le tempering de Matsumoto-Kurita jusqu'au 32^e bit. Sur chacun des GCL polynomiaux, on essaie 50 transformations linéaires différentes. Afin de pouvoir reproduire les résultats, la racine du générateur de nombres aléatoires qui produit les paramètres des transformations linéaires est `123456` et `123456`.

La figure 3 montre le contenu du fichier principal de données `exemple1`. On exécute le programme avec la commande

`% recherche exemple1.`

La figure 4 montre le contenu du fichier des transformations linéaires `trans32.dat`. La figure 5 donne le contenu du fichier `32poly.dat` qui contient tous les polynômes à vérifier, tandis que la figure 6 illustre l'affichage que le programme donne pour cet exemple. La première partie de cette figure comporte un résumé des paramètres de la recherche ainsi que le premier générateur trouvé. Tous les générateurs trouvés apparaissent à l'écran, lors de l'exécution du programme (à la figure 6, les trois points verticaux ($\vdots$) indiquent que les résultats ne sont pas tous représentés à la figure) et on peut rediriger les résultats dans un fichier du nom de `resultat` par la commande

`% recherche exemple1 > resultat .`

Dans l'affichage des paramètres du GCL polynomial, on retrouve le polynôme caractéristique sous deux formes : un vecteur de bits (en notation hexadécimale) ou la liste des degrés ayant des coefficients non nuls. On donne les paramètres de la permutation dans l'ordre p et q et pour le tempering de Kurita-Matsumoto, on donne les valeurs de η et μ entre parenthèses et les vecteurs $\mathbf{b}$ et $\mathbf{c}$ en notation hexadécimale. À la fin de l'exécution du programme, celui-ci donne un résumé des résultats de la recherche. La figure 7 montre ce résumé des résultats. Le programme donne le nombre de générateurs vérifiés, qui sont ME et CF-ME ainsi que le nombre de générateurs qui ont respecté les bornes supérieures sur les critères que l'on a définis. Le temps d'exécution du programme pour faire la recherche est également affiché. Pour cet exemple, 650 générateurs ont été vérifiés et 396 ont été retenus. Aucun générateur n'était ME.

```
1                      # Générateur à une seule composante
123456 123456          # Racine du générateur des paramètres de trans.lin.
poly 32 32poly.dat     # Générateur GCL polynomial, L=32, fichier 32poly.dat
1 trans32.dat          # fichier de trans.lin.=trans32.dat
50                     # 50 essais de trans.lin. par générateur
3                      # Somme Delta_l pour l dans Psi_12 <= 3
0                      # Aucune restriction sur bits particuliers
0                      # Ne pas vérifier les projections dans dim. éloignées
```

FIG. 3 – Fichier principal de données pour l'exemple 1.

```
2                           # 2 transformations linéaires
permut -1 -1                # permutation : p et q choisis au hasard
tempMKopt 7 15  -1 0 32     # temp. MK optimisé jusqu'au 32e bit. eta=7,mu=15
```

FIG. 4 – Fichier de transformations linéaires `trans32.dat`.

Exemple 2 On désire obtenir des générateurs de type Tausworthe à deux composantes qui ont des petites périodes (entre 2^7 et 2^{21}). Le fichier `trinomes.dat` contient les informations sur les polynômes (tous des trinômes) à vérifier (dans la description du module tausworthe en annexe, le format de ce fichier y est expliqué). On a $L_j = 32$ pour chacune

```
32 30 25 24 22 15  6  2  0
32 31 30 29 23 21 19 18 16 13 11 10  9  8  7  5  3  1  0
32 28 26 23 21 20 19 18 17 16 13 10  9  7  2  1  0
32 30 29 25 23 19 17 16 15 13 12 11  9  8  7  4  3  1  0
32 31 30 29 28 24 23 21 18 14 13  9  4  3  0
32 31 29 28 24 23 22 21 17 16 15 12 11  9  7  6  5  4  3  1  0
32 31 29 27 26 25 21 20 18 16 15 14 11 10  9  8  7  6  4  3  2  1  0
32 31 28 27 25 24 22 21 17 15 13 12 10  7  6  4  0
32 31 30 24 23 22 21 18 16 14 11 10  9  8  6  5  3  2  0
32 31 26 25 23 22 21 20 17 16 14 12 10  8  5  4  3  2  0
32 31 26 24 19 16 14 12 11 10  8  5  3  2  0
32 28 27 26 24 23 20 18 16 15 14 13 11 10  8  5  4  3  0
32 31 29 28 25 23 22 16 14 11 10  8  6  4  0
```

FIG. 5 – Fichier 32poly.dat.

des composantes. On ne désire pas nécessairement que le générateur soit ME : on ne va pas mettre de contrainte sur l'équidistribution des valeurs successives. Aussi, on ne désire appliquer à la sortie aucune transformation linéaire. Par contre, on ne désire conserver que les générateurs ayant $\Delta_{10,10,5,5} \leq 2$ (voir module tuplets).

La figure 8 montre comment devrait être le fichier principal de données. À noter que le mot `same` indique que les deux étages contiennent les mêmes générateurs et permet au programme de ne pas vérifier deux fois le même générateur combiné. La figure 9 contient le fichier trinomes.dat et la figure 10 montre l'affichage à l'écran pour le résumé des paramètres de recherche et le premier générateur trouvé. Les générateurs de Tausworthe sont donnés sous la forme de leur polynôme caractéristique et de leur paramètre s. Le premier tableau de la figure 10 donne les écarts en équidistribution pour les dimensions successives et le deuxième tableau indique les écarts maximaux obtenus dans la deuxième partie du critère $\Delta_{t_1,\ldots,t_s}$ et le pourcentage des projections qui ont un écart de 0. On voit pour le premier générateur trouvé que $\Delta_{10,10,5,5} = 2$ au bas de ce deuxième tableau. La figure 11 montre le résultat des recherches. Le programme a retenu 449 générateurs car ceux-ci correspondaient aux critères de recherche. Le temps de la recherche a été de 1.18 seconde.

Exemple 3 On désire avoir des générateurs de type TGFSR à 2 composantes. Les données sur chacune des composantes se trouvent dans `96_1.dat` et `155_2.dat`, ils sont affichés à la figure 13 (les détails sur le format de ces fichiers se trouvent en annexe dans l'explication du module tgfsr). Sur chacune de ces composantes, on désire appliquer un self-tempering (voir module selft) et un tempering de Matsumoto-Kurita (voir module temperMK) avec paramètres $\eta = 7$ et $\mu = 15$ que l'on désire optimiser jusqu'au 31^e bit (car on optimise jusqu'à L où $L = \min\{L_1, \ldots, L_J\}$, $J = 2$, $L_1 = 32$ et $L_2 = 31$). On veut $\sum_{\ell \in \Psi_{12}} \Delta_\ell \leq 20$, mais que $\Delta_\ell = 0$ pour $\ell = 1, \ldots, 9$ et $\Delta_\ell \leq 1$ pour $\ell = 10, \ldots, 32$. On désire essayer 25 transformations linéaires différentes sur chaque générateur combiné.

La Figure 12 montre comment devrait être le fichier principal de données et la figure 14 montre le fichier `trans1.dat`. Le résumé des paramètres de la recherche et le premier générateur trouvé sont à la figure 15. On peut remarquer que l'on affiche le paramètre **a** sous la forme hexadécimale tout comme les vecteurs **b** et **c** du tempering de Matsumoto-Kurita.

```
====================================================================
RESUME DE LA RECHERCHE DE GENERATEURS

Racine du générateur des trans. lin. = ( 123456, 123456 )

1 composante :
- Composante 1 : Polynomial
  Transformations :
   * Permutation
   * Tempering Matsumoto-Kurita
Nombre d'essais par générateur combiné : 50
Borne sup. pour la somme des écarts dans psi_12 : 3
====================================================================
==================================================================
Degré global :  32

GCL polynomial :
 32 30 25 24 22 15 6 2 0
sous forme hexadecimale :
 43408045
Permutation(3,3)
Tempering Matsumoto-Kurita (7,15)  b = 741e8200   c = 53158000
==================================================================

  Tableau des écarts en dimension pour chaque résolution
=======+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+
RESOL  |    1|    2|    3|    4|    5|    6|    7|    8|    9|   10|   11|   12|   13|   14|   15|   16|
-------+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----|
ECART  |     |     |     |    1|     |    1|     |    1|     |     |     |     |     |     |     |     |
-------+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----|
DIM    |   32|   16|   10|    7|    6|    4|    4|    3|    3|    3|    2|    2|    2|    2|    2|    2|
=======+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====
=======+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+
RESOL  |   17|   18|   19|   20|   21|   22|   23|   24|   25|   26|   27|   28|   29|   30|   31|   32|
-------+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----|
ECART  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
-------+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----|
DIM    |    1|    1|    1|    1|    1|    1|    1|    1|    1|    1|    1|    1|    1|    1|    1|    1|
=======+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+
---------------------------->SOMMES ECARTS (Psi_12) = 3


+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    .
    .
    .
```

FIG. 6 – Résumé des paramètres de la recherche et premier générateur trouvé pour l'exemple 1.

```
===========================
   total   =        650

     ME    =          0
   CF-ME   =          0
  retenus  =        396
---------------------------
# CPU (sec) =   2744.34
===========================
```

FIG. 7 – Résumé des résultats de la recherche pour l'exemple 1.

```
2                      # générateur à 2 composantes
123456 123456          # racine du générateur pour les params des trans.lin.
taus 32 trinomes.dat   # générateur de Tausworthe, L=32, fichier=trinomes.dat
0                      # aucune transformation linéaire composante 1
taus 32 same           # générateur de Tausworthe, L=32, même fich. :trinomes.dat
0                      # aucune transformation linéaire composante 2
0                      # aucune trans. lin. sur aucune composante
-1                     # Ne pas vérifier équidistribution
0                      # Aucune restriction sur bits particuliers
1 4 10 10 5 5 2        # Vérifier si  Delta(10,10,5,5) <= 2
```

FIG. 8 – Fichier principal de données pour l'exemple 2.

```
9
3 3 1 0
3 4 1 0
3 5 2 0
3 6 1 0
3 7 1 0
3 7 3 0
3 9 4 0
3 10 3 0
3 11 2 0
```

FIG. 9 – Fichier trinomes.dat.

```
=================================================================
RESUME DE LA RECHERCHE DE GENERATEURS

Racine du générateur des trans. lin. = ( 123456, 123456 )

2 composantes :
- Composante 1 : Tausworthe
- Composante 2 : Tausworthe
On vérifie le Critère DELTA( 10, 10, 5, 5)
On conserve quand DELTA(t_1,..,t_s)<=2
=================================================================
===============================================================
Degré global :   7

Generateur de Tausworthe :
 x^3 + x + 1                              s=1
---------------------------------------------------------------
Generateur de Tausworthe :
 x^4 + x + 1                              s=1
===============================================================

  Tableau des écarts en résolution (dimensions succesives)
=======+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+
DIM    |    1|    2|    3|    4|    5|    6|    7|    8|    9|   10|
-------+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----|
ECART  |     |    2|    1|     |     |     |     |     |     |     |
-------+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----|
RESOL  |    7|    1|    1|    1|    1|    1|    1|    0|    0|    0|
=======+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+


  Tableau des écarts maximaux en résolution (dimensions non-succesives)
==========+======+======+======+
Dimension  |     2|     3|     4|
-----------+------+------+------+
t_i        |    10|     5|     5|
-----------+------+------+------+
ECART      |     2|     1|     0|
-----------+------+------+------+
pourcentage| 50.00| 16.67|100.00|
===========+======+======+======+
Valeur de DELTA( 10, 10, 5, 5 ) = 2
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    .
    .
    .
```

Fig. 10 – Résumé des paramètres de recherche et premier générateur trouvé pour l'exemple 2.

```
===========================
   total   =        668

     ME    =          0
   CF-ME   =          0
  retenus  =        449
---------------------------
 CPU (sec) =    1.18
===========================
```

Fig. 11 – Résumé des résultats de la recherche pour l'exemple 2.

```
2                  # générateur à 2 composantes
123456 123456      # racine du générateur pour les trans.lin.
tgfsr 32 96_1.dt   # générateur TGFSR, L=32, fichier 96_1.dt
1 trans32.ex3.dat  # fichier de trans.lin. est trans32.ex3.dat
tgfsr 31 155_2.dt  # générateur TGFSR, L=32, fichier 155_2.dt
1 trans31.ex3.dat  # fichier de trans.lin. est trans31.ex3.dat
25                 # 25 trans.lin. par générateur combiné
20                 # Somme Delta_l dans Psi_12 <=20
2                  # sur 2 lignes, on met des restrictions sur Delta_l
1 9 0              # Delta_l<=0 pour l=1..9
10 32 1            # Delta_l<=1 pour l=10..32
0                  # On ne vérifie pas les projections dans dim. éloignées.
```

FIG. 12 – Fichier principal de données pour l'exemple 3.

```
32 3
a 5  9965523b b07c9b61 93da7639 89415f19 bd20f645
m 1 1
```

```
31 5
a 5 f45c111a ba693616 9be9c82a 9b6bf432 cd68fd62
m 1 2
```

FIG. 13 – Fichier `96_1.dat` et `155_2.dat` pour l'exemple 3.

```
1                     # 1 transformation linéaire
tempMKopt 7 15 -1 0 31 # temp.MK optimisé jusqu'à 31. eta=7, mu=15
```

```
1                     # 1 transformation linéaire
tempMKopt 7 15 -1 0 32 # temp.MK optimisé jusqu'à 32. eta=7, mu=15
```

FIG. 14 – Fichiers de transformations linéaires `trans31.ex3.dat` et `trans32.ex3.dat`.

```
=================================================================
RESUME DE LA RECHERCHE DE GENERATEURS

Racine du générateur des trans. lin. = ( 123456, 123456 )

2 composantes :
- Composante 1 : TGFSR
  Transformations :
   * Tempering Matsumoto-Kurita
- Composante 2 : TGFSR
  Transformations :
   * Tempering Matsumoto-Kurita
Nombre d'essais par générateur combiné : 25
Borne sup. pour la somme des écarts dans psi_12 : 20
delta pour bits particuliers :
   >>Bits 1 à 9 delta_i=0
   >>Bits 10 à 32 delta_i=1
=================================================================
===============================================================
Degré global : 251

TGFSR :
 w=  32  r=   3  m=   1  a=   9965523b
Tempering Matsumoto-Kurita (7,15)  b = 711a2101   c = 7dfa0001
---------------------------------------------------------------
TGFSR :
 w=  31  r=   5  m=   1  a=   f45c111a
Tempering Matsumoto-Kurita (7,15)  b = 63ee5402   c = 3bbd0000
===============================================================

  Tableau des écarts en dimension pour chaque résolution
======+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+
RESOL  |    1|    2|    3|    4|    5|    6|    7|    8|    9|   10|   11|   12|   13|   14|   15|   16|
-------+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----|
ECART  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
-------+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----|
DIM    |  251|  125|   83|   62|   50|   41|   35|   31|   27|   25|   22|   20|   19|   17|   16|   15|
======+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====
======+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+
RESOL  |   17|   18|   19|   20|   21|   22|   23|   24|   25|   26|   27|   28|   29|   30|   31|
-------+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----|
ECART  |     |     |    1|    1|     |     |     |     |    1|    1|    1|     |     |     |     |
-------+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----|
DIM    |   14|   13|   12|   11|   11|   11|   10|   10|    9|    8|    8|    8|    8|    8|    8|
======+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+
-------------------------->SOMMES ECARTS (Psi_12) = 4


+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    .
    .
    .
```

FIG. 15 – Résumé des paramètres de recherche et premier générateur trouvé pour l'exemple 3.

4 Ajout de nouveaux développements dans le progiciel

Cette section explique comment un programmeur peut ajouter de nouveaux générateurs, de nouvelles transformations linéaires et de nouveaux critères au progiciel. Pour chaque nouveau module, il est fortement conseillé d'étudier ceux qui existent déjà et leurs interactions avec les autres modules et de s'en inspirer.

4.1 Nouveau générateur

Supposons que le nouveau type de générateur s'appelle XXX. Alors, la première étape consite à créer un fichier XXX.c dans lequel on définira les éléments décrits par la Figure 16. Afin de développer ces fonctions, il est fortement recommandé de s'inspirer de celles déjà implantées dans les modules tgfsr, tausworthe et polynomial. La fonction à la Figure 17 est facultative.

4.2 Nouvelle transformation linéaire

Afin d'ajouter des transformations linéaires, il faut écrire les éléments décrits dans la Figure 18 dans le fichier YYY.c, en supposant que la transformation linéaire se nomme YYY. Il est préférable de s'inspirer des fonctions du même type déjà implantées dans les modules selft, temperMK et permut.

4.3 Nouveau critère

Pour définir un nouveau critère ZZZ, il suffit de créer un fichier ZZZ.c. Dans ce fichier, on définira un type structure du nom de `paramZZZ`. Cette structure contiendra tous les paramètres du critère. On pourra vérifier le critère en appelant la fonction `TestZZZ()`. Cette fonction aura comme paramètre une variable de type `Combinaison` et une de type `paramZZZ`. La fonction mettra à jour les champs de la variable de type `paramZZZ`. Il sera ensuite possible de vérifier certains aspects du critère avec des fonctions du type `estZZZ()`. Il est recommandé de s'inspirer des modules mecf et tuplets pour l'élaboration d'un nouveau module de ce genre.

```
typedef struct{
  ...
}paramXXX ;
```

La variable `paramXXX` contient les paramètres d'un générateur XXX ainsi que son état.

```
char *XXXNom ( void
             ) ;
```

Fonction qui affiche à l'écran le nom du type de générateur.

```
void InitParamXXX( void *params,
                   p1,
                   ..,
                   pN
                 ) ;
```

Permet d'initialiser les paramètres du générateur XXX défini par la variable pointée par `*params`. Le paramètre formel est de type `*void` (bien que le paramètre actuel devrait être de type `*paramXXX`) afin d'être compatible avec la structure de type `Generateur` définie au module etages. Les paramètres `p1,..,pN` représentent les paramètres du générateur.

```
void AjoutXXX ( Etage *E,
                paramXXX *params,
              ) ;
```

Cette fonction permet d'ajouter à l'étage `*E` le générateur pointé par `*paramXXX`.

```
void InitXXX ( void *params,
               int i,
               BitVect retour
             ) ;
```

Cette fonction met dans la racine du générateur XXX pointé par `*params` le (`i`+1)-ième vecteur de la base canonique. Une copie de cette racine est mise dans `retour`.

```
void XXX( void *params,
          BitVect retour
        ) ;
```

Retourne dans `retour` le prochain état du générateur XXX pointé par `*params`.

```
void AffXXX ( void *params
            ) ;
```

Fonction qui permet d'afficher à l'écran les paramètres du générateur XXX pointé par `*params`.

FIG. 16 – À écrire dans le fichier XXX.c

```
void LireDonneesXXX ( Etage *E,
                      char *filename,
                      boolean same,
                      int L
                    ) ;
```

Fonction facultative, mais très utile, qui permet de lire dans le fichier `*filename` tous les paramètres des générateurs de type XXX qui doivent être mis à l'étage `*E`. Le paramètre `same` indique que les générateurs de l'étage `*E` sont les mêmes que ceux de l'étage précédent, si cet étage est contenu dans une structure de type `Combinaison`. Le paramètre `L` indique le nombre de bits de résolution de la sortie des générateurs lus.

FIG. 17 – À écrire dans le fichier XXX.c (Facultatif)

```
typedef struct {
  ..
} paramYYY ;
```

Type contenant les paramètres de la transformation linéaire. Certains éléments de cette structure sont les paramètres effectifs de la transformation linéaire YYY. D'autres permettent de déterminer la façon de changer ces paramètres à l'aide de la fonction `SetYYY()`.

```
int DonnerYYYID ( void
                ) ;
```

Retourne le numéro de la transformation linéaire. Avant d'attribuer un numéro à cette transformation linéaire, il faut s'assurer qu'il est différent des autres transformations linéaires définies dans le progiciel.

```
void InitYYY( paramYYY *params,
              p1,
              ..
              pN
            ) ;
```

Fonction qui permet d'initialiser les paramètres de la transformation linéaire pointée par `*params`. Les paramètres `p1,..,pN` indiquent la façon dont `MAJYYY()` choisit les paramètres.

```
void AjoutYYY( Etage *E,
                paramYYY *params
              ) ;
```

Fonction qui ajoute une transformation linéaire de type YYY de paramètres `*params` à l'étage `*E`.

```
void MAJYYY( void *params
             int w
           ) ;
```

La fonction `MAJYYY()` permet de changer les paramètres effectifs de la transformation linéaire. Le paramètre `w` indique le nombre de bits sur lesquels la transformation linéaire s'effectue.

```
void YYY( BitVect A,
          void *params,
          int w
        ) ;
```

Fonction qui effectue la transformation linéaire dont les paramètres sont pointés par `*params` sur seulement, les `w` premiers bits du vecteur de bits `A`.

```
void AffYYY ( void *params,
               int w
             ) ;
```

Fonction qui affiche à l'écran les paramètres de la transformation linéaire pointée par `*params`. Le paramètre `w` indique le nombre de bits sur lesquels la transformation linéaire s'effectue.

FIG. 18 – À écrire dans le fichier YYY.c

5 Conclusion

Le progiciel REGPOLY permet de trouver de bons générateurs qui utilisent une récurrence linéaire modulo 2. Il a été conçu de manière à ce qu'il soit facile de vérifier d'autres types de générateurs que les générateurs déjà implantés (Tausworthe, TGFSR et GCL polynomial) et de nouvelles transformations linéaires en ajoutant de nouveaux modules. Par les équations 1, 2, 3 et 4, on observe que le cadre défini par ces équations permet un très grand nombre de générateurs différents. Parmi tous ces générateurs, il faut chercher des combinaisons de matrices X, B et Y qui donnent de bons générateurs et que l'implantation soit rapide. Beaucoup de bons générateurs ont été trouvés (voir [8, 2]), mais il reste du travail pour la recherche de bons générateurs qui sont également rapides.

Pour le développement futur de REGPOLY, le plus pressant est la construction d'un module qui permet la vérification de Mersenne Twister [7] et l'amélioration des algorithmes qui vérifient l'équidistribution. Comme il est vu dans la description du module mecf, pour vérifier la (t, ℓ)-équidistribution d'un générateur, on diagonalise la matrice $B_{t,\ell}$. Par la suite, pour le même générateur, on pourrait vouloir verifier la (t', ℓ')-équidistribution où $t > t'$ (ce qui se produit très souvent). Il serait avantageux de profiter du travail qui a été fait pour diagonaliser $B_{t,\ell}$ afin de diagonaliser $B_{t',\ell'}$ puisque ces deux matrices ont beaucoup de colonnes en commun. Dans sa version présente, REGPOLY n'utilise aucune stratégie de ce genre pour économiser du temps de calcul.

ANNEXES

A. Définition fonctionnelle des modules de REGPOLY

btypes

Ce module contient certains types et constantes de base du progiciel. On peut modifier également certaines constantes de ce module afin d'augmenter la période des générateurs admis par le progiciel.

```
#define FALSE    0
#define TRUE     1
```

FAUX et VRAI booléen.

```
#define WL      32
```

Longueur d'un mot machine.

```
#define     BV         5
#define     MAXL       (BV*WL)
#define     J_MAX      4
#define     K_MAX      1300
```

La constante `BV` est la taille d'un `BitVect` en mots machines.

La constante `MAXL` est la taille d'un `BitVect` en bits. C'est aussi le nombre maximum de bits que la sortie d'un générateur peut avoir.

La constante `J_MAX` est le nombre maximum de composantes d'un générateur combiné.

La constante `K_MAX` est le degré global maximum pour le générateur combiné.

Ces quatre macros peuvent être modifiées si les générateurs à tester ont un polynôme caractéristique avec un degré supérieur à `K_MAX`, si l'état des générateurs nécessite plus que `BVL` bits ou si on veut plus que `J_MAX` composantes pour le générateur combiné.

```
typedef unsigned char     boolean;
typedef          ulong    BitVect[BV];
```

Ces types sont des types qui sont utilisés dans tout le progiciel. Le type `boolean` est utilisé pour les variables booléennes. Le type `BitVect` permet de manipuler des vecteurs de bits de dimensions plus grandes que `WL`. Ce type est en fait un tableau unidimensionnel de `BV` objets de type `ulong`.

```
typedef BitVect * Vecteur;
typedef struct{
  Vecteur   lignes[K_MAX];
  int nblignes;
  int nbcolonnes;
} Matrice;
```

Le type `Vecteur` est un pointeur à une variable de type `BitVect`. La variable `lignes` de type `Vecteur` est un tableau à deux dimensions de valeurs de type `BitVect`. Ce tableau aura `K_MAX` lignes et les colonnes seront allouées dynamiquement avec la fonction `AllouerMat()` définie dans le module baseopera. Les variables `nblignes` et `nbcolonnes` représentent le nombre de lignes et le nombre de colonnes qui ont été allouées.

etages

Cette section du progiciel sert à administrer toute l'information reliée aux générateurs combinés. Le type `Combinaison` sert à gérer la recherche de générateurs combinés. Ce type est composé d'autant d'étages que de composantes des générateurs combinés recherchés. Sur le premier étage, on retrouve tous les générateurs que l'on veut vérifier comme première composante du générateur combiné et de même pour les autres étages. Une fois tous les étages saturés, alors on est prêt à vérifier différents critères sur tous les différents générateurs combinés possibles contenus dans la variable de type `Combinaison`.

Afin d'allouer l'espace nécessaire pour vérifier un générateur à J composantes, on doit tout d'abord déclarer une variable de type `Combinaison`. Ensuite, il faut appeler la fonction `AllouerEtagesDansCombinaison()` dont les paramètres sont le nombre de composantes du générateur combiné. Ceci permet de réserver l'espace mémoire nécessaire pour J étages.

Une fois `AllouerEtagesDansCombinaison()` appelé, il faut réserver l'espace nécessaire pour chacun des générateurs que l'on veut mettre à chaque étage. Pour un étage, il est nécessaire de savoir combien de générateurs on désire mettre dans celui-ci. On y arrive en appelant la fonction `AllouerGensDansEtage()` avec comme paramètre le nombre de générateurs à essayer à cet étage. Le paramètre `meme` permet de ne pas tester plusieurs fois le même générateur combiné, si deux étages consécutifs contiennent les mêmes générateurs et dans le même ordre.

Si on décide d'appliquer des transformations linéaires sur une ou plusieurs composantes du générateur combiné, alors il faut réserver l'espace mémoire nécessaire pour pouvoir les effectuer. Ceci est accompli en appelant la fonction `AllouerTransDansEtage()` du module translin et d'autres fonctions décrites dans les modules de transformations linéaires. Voir la description de ces modules pour plus de détails sur les transformations linéaires.

Une fois la mémoire réservée, il est maintenant temps d'ajouter les générateurs que l'on veut mettre dans chacun des étages. Les fonctions qui permettent cela pour chacun des types de générateurs sont décrites dans les modules spécifiques à chacun.

Afin de tester les générateurs combinés, il est important de vérifier tous les générateurs combinés possibles étant donné la variable de type `Combinaison` que l'on a définie. Pour ce faire, la fonction `PremierFirstGen()` prend le premier générateur combiné valide et lui donne le statut de générateur combiné courant. Un générateur combiné valide est un générateur pour lequel deux composantes ne peuvent avoir la même période et le générateur combiné courant est celui sur lequel on peut vérifier des critères (équidistribution, équidistribution des projections, etc...). La fonction `ProchainGen()` permet de prendre le prochain générateur combiné valide et le rendre courant. Ces 2 fonctions permettent de passer à travers tous les générateurs combinés possibles compris dans une variable de type `Combinaison`.

Enfin, la fonction `AffCombCourante()` permet d'afficher à l'écran le générateur combiné courant, (c'est-à-dire le générateur courant de chacun des étages) et les transformations linéaires appliquées à la sortie de chacune des composantes du générateur combiné courant.

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

#include "btypes.h"
#ifndef INCLUDELINTRANS_H
#define INCLUDELINTRANS_H
#include "lintrans.h"
#endif


typedef struct {
  void* ParamGen ;
  void (*AffGen)(void *params) ;
  char* (*AffNom)(void) ;
  void (*InitGen)(void *params, int i, BitVect retour) ;
  void (*Iteration)(void *params, BitVect B) ;
  int k ;
  int Saut ;
  int L ;
  int w ;
} Generateur ;
```

Type permettant d'emmagasiner les paramètres d'un générateur, peu importe le type. Le pointeur `ParamGen` pointe vers une variable qui contient les paramètres du générateur. Ce pointeur est de type `void*` afin d'être compatible avec tous les types de générateurs. Le pointeur `AffGen` permet d'appeler la fonction d'affichage du générateur. Le pointeur `AffNom` permet d'afficher le nom du type de générateur. Le pointeur `InitGen` met dans l'état initial du générateur le (`i`+1)-ième vecteur unitaire et retourne dans `retour` l'état du générateur plus, s'il y a lieu, des bits supplémentaires venant de la récurrence (par exemple, dans le cas des générateurs de Tausworthe, `retour` contient l'état plus d'autres bits de la récurrence). La fonction pointée par `Iteration` fait une itération du générateur et met l'état résultant dans `B`. La variable `k` indique le nombre de bits sur lesquels l'état du générateur est défini. La variable `Saut` est un paramètre du générateur implanté tel que pgcd($2^{\mathtt{k}}-1$,`Saut`) doit être 1, si on désire le générateur de période $2^{\mathtt{k}}-1$ (par exemple, pour les générateurs de type Tausworthe, c'est le paramètre s). S'il n'y a pas de tels paramètres, `Saut` est mis à 1. La variable `L` est le nombre de bits de la sortie générés par le générateur et `w` est le nombre de bits de l'état qui sont utilisés par les transformations linéaires appliquées à ce générateur. Pour les pointeurs aux fonctions, le paramètre `*params` représente un pointeur aux paramètres du générateur.

```
typedef struct {
  Generateur* Gen ;
  struct TransType* Trans ;
  int GenCourant ;
  int NbTrans ;
  int NbTransAlloc ;
  int NbGen ;
  int NbGenAlloc ;
  boolean meme ;
} Etage ;
```

Type permettant d'emmagasiner l'information de tous les générateurs à explorer pour une composante du générateur combiné. Le tableau `*Gen` contient l'information de tous les générateurs à explorer dans sa composante correspondante. Le tableau `*Trans` contient l'information concernant les transformations linéaires appliquées à la sortie de cette composante à explorer. L'espace mémoire des composantes des tableaux `*Trans` et `*Gen` est allouée dynamiquement par les fonctions `AllouerGensDansEtage()` et `AllouerTransDansEtage()`. La variable `GenCourant` correspond à un pointeur du générateur courant de cet `Etage`. Supposons une variable `E` de type `Etage`, alors `E.Gen[E.GenCourant]` est le générateur courant de l'`Etage`. La variable `NbTrans` indique le nombre de transformations linéaires à effectuer à la sortie du générateur de cette composante. La variable `NbTransAlloc` indique le nombre de transformations linéaire pour lesquelles de l'espace mémoire a été réservé. La variable `NbGen` indique le nombre de générateurs que l'on veut explorer à cet étage. La variable `NbGenAlloc` indique le nombre de générateurs pour lesquels de l'espace mémoire à cet étage a été réservé. La variable `meme` indique que cet étage est pareil au précédent, si celui-ci se trouve dans une structure de type `Combinaison`.

```
typedef struct{
  Etage* Etages ;
  int J ;
  int k_g ;
  int L ;
} Combinaison ;
```

Type permettant d'emmagasiner toute l'information nécessaire sur les générateurs combinés à explorer. Le tableau `Etages` contient tous les `Etages` de la combinaison. L'espace mémoire pour ce tableau est alloué dynamiquement avec la fonction `AllouerEtagesDansCombinaison`. La variable `J` indique le nombre d'étages que `Combinaison` contient. La variable `k_g` est le degré du générateur combiné courant. La variable `L` est la résolution de la sortie du générateur combiné courant.

```
void AllouerEtagesDansCombinaison ( Combinaison *C,
                                    int J
                                  ) ;
```

Fonction qui permet de réserver l'espace nécessaire afin de tester des générateurs combinés à `J` composantes à l'aide de la variable `*C`.

```
void EffacerEtagesDansCombinaison ( Combinaison *C
                                  ) ;
```

Fonction qui libère l'espace mémoire occupé par les `Etages` d'une variable de type `Combinaison`.

```
void AllouerGensDansEtage ( Etage *E,
                            int nbgen,
                            boolean meme
                          ) ;
```

Fonction qui permet de réserver l'espace nécessaire afin de tester `nbgen` générateurs à l'étage `*E`. La variable `meme` indique que l'étage `*E` est pareil à l'étage précédent, si celle-ci se trouve dans une structure de type `Combinaison`.

```
void EffacerGensDansEtage ( Etage *E
                          ) ;
```

Fonction qui libère l'espace mémoire occupé par les générateurs de l'étage `E`.

```
boolean PremierGen ( Combinaison *C
                   ) ;
```

Une fois tous les étages bien définis de `*C`, cette fonction permet de mettre le pointeur du générateur combiné courant sur le premier valide. Retourne `TRUE` si elle réussit, `FALSE` sinon.

```
boolean ProchainGen ( Combinaison *C
                    ) ;
```

Fonction qui permet de mettre le pointeur du générateur combiné courant sur le prochain valide. Retourne `TRUE` si elle réussit, `FALSE` sinon. Il est important d'appeler `PremierGen()` avant celle-ci.

```
void AffCombCourante ( Combinaison *C
                     ) ;
```

Fonction qui affiche à l'écran le générateur combiné courant (générateurs et transformations linéaires).

lintrans

Si on décide d'appliquer des transformations linéaires sur une ou plusieurs composantes du générateur combiné, alors il faut réserver l'espace mémoire nécessaire pour pouvoir les effectuer. Ceci est accompli en appelant la fonction `AllouerTransDansEtage()` avec comme paramètres l'étage où l'on veut appliquer des transformations linéaires ainsi que le nombre de transformations linéaires.

Pour ajouter une transformation linéaire spécifique sur un étage, voir la description du module correspondant.

Les fonctions `MAJTrans()` et `MAJToutesTrans()` permettent de mettre à jour les paramètres effectifs des transformations linéaires.

```
#include <time.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "btypes.h"
#include "etages.h"
```

```
typedef struct TransType {
  void* ParamTrans ;
  void (*AffLinTrans)(void *params, int w ) ;
  void (*SpecifiqueTrans)(BitVect A,void *params,int) ;
  void (*ChangeParamTrans)(void *params,int w) ;
  int (*DonnerTransID)(void) ;
} Transformation ;
```

Type permettant d'emmagasiner les paramètres d'une transformation linéaire, peu importe le type. Le pointeur `ParamTrans` pointe vers une variable qui contient les paramètres de la transformation linéaire et est de type `void*` afin d'être compatible avec tous les types de transformation linéaire. Le pointeur `AffLinTran` permet d'appeler la fonction d'affichage des paramètres de la transformation linéaire. Le pointeur `SpecifiqueTrans` permet d'appeler la fonction qui transforme le vecteur de bits `A` selon la transformation linéaire. La fonction pointée par `ChangeParamTrans` permet de mettre de nouvelles valeurs aux paramètres effectifs de la transformation linéaire. La fonction `DonnerTransID` indique le type de la transformation linéaire par son numéro d'identification. Pour les paramètres des fonctions, `*params` représente un pointeur vers les paramètres de la transformation linéaire et `w` représente le nombre de bits sur lesquels la transformation linéaire s'effectue.

```
void AffTrans ( Etage *E,
                int w
              ) ;
```

Fonction qui permet d'afficher les transformations qui sont appliquées sur le générateur courant de l'étage `*E`. Le paramètre `w` indique le nombre de bits sur lesquels cette transformation est effectuée.

```
void Transforme ( Transformation *T,
                  int NbTrans,
                  BitVect A,
                  int w
                );
```

Fonction qui applique sur `A` les transformations linéaires contenues dans le tableau `*T`. La variable `NbTrans` indique le nombre de transformations linéaires qui sont contenues dans le tableau `*T`. Le paramètre `w` indique le nombre de bits sur lesquels cette transformation est effectuée.

```
void MAJToutesTrans ( Combinaison *C
                    );
```

Fonction qui change les paramètres effectifs des transformations linéaires de tous les étages contenues dans `*C`.

```
void MAJTrans ( Etage *E,
                int w
              );
```

Fonction qui change les paramètres effectifs des transformations linéaires de l'étage `*E`. Le paramètre `w` indique le nombre de bits sur lesquels cette transformation est effectuée.

```
void AllouerTransDansEtage ( Etage *E,
                             int nbtrans
                           );
```

Fonction qui permet de réserver l'espace nécessaire afin d'effectuer `nbtrans` transformations linéaires sur l'étage `*E`.

```
void EffacerTransDansEtage ( Etage *E
                           );
```

Fonction qui libère l'espace mémoire occupé par les transformations linéaires à l'étage `*E`.

baseopera

Ce module contient la fonction `Init_REGPOLY()` qu'il est essentiel d'appeler avant d'utiliser les fonctions définies dans n'importe quel module du progiciel REGPOLY. Il y a également plusieurs fonctions qui agissent sur le type `BitVect` ainsi que d'autres fonctions qui sont utilisées par d'autres modules et qui peuvent être utilisées. Ce module contient également les fonctions `AllouerMat()` et `EffacerMat()` qui permettent de réserver et libérer l'espace mémoire pour les matrices de bits.

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include "btypes.h"
```

```
BitVect B_C[MAXL] ;
```

Tableau de `BitVect` où l'élément `B_C[`t`]` contient le $(t+1)$-ième vecteur unitaire.

```
BitVect TousUns ;
```

Vecteur de bits (1, 1, 1, 1,.. , 1)

```
BitVect  masque[MAXL],
         invmasque[MAXL] ;
```

Les variables `masque` et `invmasque` sont des matrices de bits. Le vecteur de bits `masque[`r`]` est constitué de $r+1$ uns suivis de zéros tandis que `invmasque[`r`]` est constitué de r zéros suivit de uns.

```
void Init_REGPOLY ( void
                  ) ;
```

Fonction qu'il est essentiel d'appeler avant d'utiliser n'importe quelle fonction du progiciel REGPOLY. Cette fonction initialise les vecteurs de bits `B_C[]`, `TousUns`, `masque[]` et `invmasque[]`. Ces vecteurs de bits sont utilisés dans d'autres modules.

```
boolean Diag ( Matrice *m,
               int kg,
               int t,
               int l,
               int *gr
             ) ;
```

Évalue si la matrice pointée par `*m` sur `kg` lignes est de plein rang. On procède sur `t` `BitVect` en considérant les `l` premiers bits de chacun. La fonction retourne `TRUE` si la matrice `*m` est de plein rang `t*l` et `*gr` est inchangé. Sinon, la fonction retourne `FALSE` et `*gr` prend pour valeur le numéro du `BitVect` où il y a eu échec moins un (= dimension pour laquelle on a résolution `l`).

```
int ValBitBV ( BitVect A,
               int noBit
             ) ;
```

Retourne la valeur du `noBit`-ième bit (l'indexation va de gauche vers la droite et commence au bit 0).

```
void MetBitBV ( BitVect A,
                int noBit,
                int valBit
              ) ;
```

Met la valeur du `noBit`-ième bit à `valBit` (l'indexation va de gauche à droite et commence au bit 0) où `valBit` peut être 0 ou 1.

```
int intmin ( int A,
             int B
           ) ;
```

Fonction qui retourne la valeur minimale entre `A` et `B`.

```
int intmax ( int A,
             int B
           ) ;
```

Fonction qui retourne la valeur maximale entre `A` et `B`.

```
int pgcd ( int x,
           int y
         ) ;
```

Calcule le plus grand commun diviseur de `x` et `y`.

```
void MetBVaZero ( BitVect A
                ) ;
```

Initialise le vecteur de bit `A` à zéro.

```
void ulongaBV ( ulong l,
                BitVect A,
                int precision_w
              ) ;
```

Cette fonction est utilisée par `tgfsr.c`. Elle permet de transformer une valeur de type `ulong` en un type `BitVect`. Cette fonction copie les bits de `l` dans les `WL` premiers bits de `A` sauf dans l'ordre inverse. C'est à dire que le bit le plus significatif de `l` sera à la position 0 du BitVect `A` et le moins significatif sera à la position `precision_w`-1.

```
void CopieBV ( BitVect A,
               BitVect B
             ) ;
```

Copie le contenu de `B` dans `A` (`A=B`).

```
boolean EgalBV ( BitVect A,
                 BitVect B
               ) ;
```

Compare le contenu de `B` avec celui de `A`. Retourne `TRUE` si les deux contiennent la même information.

```
void XORBV ( BitVect A,
             BitVect B,
             BitVect C
           ) ;
```

Cette fonction effectue `A` = `B ^ C`.

```
void XOR2BV ( BitVect A,
              BitVect B,
              BitVect C,
              BitVect D
            ) ;
```

Cette fonction effectue `A` = `B ^ C ^ D`.

```
void ANDBV ( BitVect A,
             BitVect B,
             BitVect C
           ) ;
```

Cette fonction effectue `A` = `B` & `C`.

```
void ANDBVSelf ( BitVect A,
                 BitVect B
               ) ;
```

Cette fonction effectue `A` = `A` & `B`.

```
void XORBVSelf ( BitVect A,
                 BitVect B
               ) ;
```

Cette fonction effectue `A` = `A ^ B`.

```
void BVLShift ( BitVect R,
                BitVect A,
                int n
              ) ;
```

Cette fonction effectue `R` = `A` ≪ `n`.

```
void BVRShift ( BitVect R,
                BitVect A,
                int n
              ) ;
```

Cette fonction effectue `R` = `A` ≫ `n`.

```
void BVLShiftSelf ( BitVect R,
                    int n
                  ) ;
```

Cette fonction effectue `R` = `R` ≪ `n`.

```
void BVLS1Self ( BitVect R
               );
```

Cette fonction effectue `R` $=$ `R` $\ll$ `1`. C'est une version spécialisée de `BVLShiftSelf()` pour utilisation fréquente.

```
void BVRShiftSelf ( BitVect R,
                    int n
                  );
```

Cette fonction effectue `R` $=$ `R` $\gg$ `n`.

```
void InverseBV ( BitVect A
               );
```

Inverse la valeur de tous les bits de `A` (fait `A` = `~A`)

```
boolean VerifBitsCommuns ( BitVect ds1,
                           BitVect ds2
                         );
```

Vérifie si les `BitVect ds1` et `ds2` ont des bits communs qui ont comme valeur 1. Si c'est le cas, elle retourne `TRUE`, sinon elle retourne `FALSE`.

```
void AllouerMat ( Matrice *m,
                  int nl ,
                  int indice_max
                );
```

Alloue l'espace de `nl` lignes pour la matrice pointée par `*m`. Interrompt le programme si la mémoire disponible est insuffisante. Le paramètre `indice_max` indique le nombre de `BitVect` par lignes.

```
void EffacerMat ( Matrice *m
                );
```

Libère l'espace occupé par la matrice `m`.

```
void CopieMat ( Matrice *m,
                Matrice *ms,
                int nl,
                int t
              );
```

Copie les `t` premiers `BitVect` des `nl` premières lignes de la matrice `*ms` dans la matrice de travail `*m`.

```
void CopieNTupleMat ( Matrice *m,
                      Matrice *ms,
                      int nl,
                      int *colonnes,
                      int t
                    );
```

Copie les `t`-1 `BitVect` indiqués par le tableau `*colonnes`, plus la colonne 0, des `nl` premières lignes de la matrice `*ms` dans la matrice `*m`.

```
void EchangeVect ( Matrice *m,
                   int i,
                   int j
                 ) ;
```

Interchange les lignes `i` et `j` de la matrice `*m`.

```
void XorVect ( Matrice *m,
               int r,
               int s,
               int min,
               int max
             ) ;
```

Effectue un ou-exclusif entre la `s`-ième et la `r`-ième ligne de la matrice `*m` pour les colonnes (`BitVect`) `min` à `max`-1 seulement. Le résultat est mis dans la `r`-ième ligne (`m[r] = m[r] ^ m[s]`).

```
void AffMat ( Matrice *m,
              int t,
              int l,
              int k_g
            ) ;
```

Affiche la matrice `*m` sur `k_g` lignes par `t`×`l` colonnes.

```
void AffBitVect ( BitVect A,
                  int l
                ) ;
```

Affiche le `BitVect A` sur `l` bits seulement.

```
void ReadLn ( FILE *f
            ) ;
```

Fonction qui permet d'ignorer le reste de la ligne courante du fichier `f`.

```
long CombLec88 ( void
               ) ;
```

Générateur de nombre aléatoire de période d'environ 2^{88} qui retourne un mot de 32 bits de type `long`.

```
void SetCombLec88 ( long S1,
                    long S2
                  ) ;
```

Initialise le germe du générateur de nombres aléatoires `CombLec88()`. Les valeurs de `S1` et `S2` doivent pas être telles que $1 \leq$ `S1` ≤ 2147483562 et $1 \leq$ `S1` ≤ 2147483398.

mecf

L'article de Pierre l'Écuyer [2] ainsi que la référence [8] sont excellents afin de comprendre le module mecf. Ce module permet de vérifier l'équidistribution du générateur combiné courant.

Définition de l'équidistribution

Considérons l'ensemble de points Ω_t de tous les points à t dimensions produit par les valeurs successives d'un générateur, à partir de tous les états initiaux possibles $\mathbf{x}_0$, défini par

$$\Omega_t = \{\mathbf{u}_{0,t} = (u_0, \ldots, u_{t-1}) : \mathbf{x}_0 \in \mathbb{F}_2^k\}.$$

Maintenant, supposons que l'on partionne l'hypercube $[0,1)^t$ en $2^{t\ell}$ petits hypercubes de mêmes grandeurs. La plus grande valeur de ℓ pour laquelle chacun des petits hypercubes contient le même nombre de points est appelée la *résolution* en dimension t et est notée ℓ_t. On dit que Ω_t (et par le fait même, le générateur) est *(t,ℓ)-équidistribué* si tous les petits hypercubes contiennent le même nombre de points, soit $2^{k-t\ell}$ points. Ceci est possible seulement si $k \geq t\ell$, parce que la cardinalité de Ω_t est au plus 2^k. L'ensemble Ω_t est dit *équidistribué au maximum (ME)* si ℓ_t atteint sa borne supérieure pour toutes les valeurs de t possibles, c'est à dire $\ell_t = \min(L, \lfloor k/t \rfloor)$ pour $t = 1, .., k$. Un ensemble de nombres complémentaires sont les $t_\ell, 1 \leq \ell \leq L$, où t_ℓ représente la plus grande valeur de t pour laquelle Ω_t est (t,ℓ)-équidistribué. On a donc les bornes supérieures

$$\ell_t \leq \ell_t^* \stackrel{\text{def}}{=} \min(L, \lfloor k/t \rfloor) \quad \text{pour } t = 1, .., k$$

et

$$t_\ell \leq t_\ell^* \stackrel{\text{def}}{=} \lfloor k/\ell \rfloor \quad \text{pour } l = 1, .., L.$$

On définit aussi *l'écart en dimension* t par $\Lambda_t \stackrel{\text{def}}{=} \ell_t^* - \ell_t$ et *l'écart en résolution* ℓ par $\Delta_l \stackrel{\text{def}}{=} t_\ell^* - t_\ell$. On dit que le générateur a une *résolution maximale en dimension* t, si $\Lambda_t = 0$ et est de *dimension maximale en résolution* ℓ si $\Delta_\ell = 0$. Suivant ces définitions, on peut dire que le générateur est ME, si $\Lambda_t = 0$ pour $t = 1, .., k$ ou, de façon équivalente, si $\Delta_\ell = 0$ pour $\ell = 1, .., \min(k, L)$. L'équivalence peut être vue en constatant que si on a $\ell_t < \ell_t^*$ pour une valeur de t, alors ceci implique que $\Delta_{\ell_t^*} > 0$ et que, si on a $t_\ell < t_\ell^*$ pour une valeur de ℓ, alors $\Lambda_{t_\ell^*} > 0$.

En fait, il n'est pas nécessaire de calculer Δ_ℓ et Λ_t pour $\ell = 1, \ldots, \min(k, L)$ et $t = 1, \ldots, k$. Dans [2], on démontre que si $\Delta_\ell = 0$ pour tout $\ell \in \Psi_{12}$ ou si $\Lambda_t = 0$ pour tous $t \in \Phi_{12}$, alors le générateur est ME. Les ensembles Ψ_{12} et Φ_{12} sont définis par

$$\begin{aligned}
\Phi_{12} &= \{\max(2, \lfloor k/L \rfloor), \ldots, \lfloor \sqrt{k} \rfloor\} \cup \{t = \lfloor k/\ell \rfloor \mid \ell = 1, \ldots, \lfloor \sqrt{k-1} \rfloor\} \\
\Psi_{12} &= \{1, \ldots, \lfloor \sqrt{k} \rfloor\} \cup \{\ell = \lfloor k/t \rfloor \mid t = \max(2, \lfloor k/L \rfloor), \ldots, \lfloor \sqrt{k-1} \rfloor\}.
\end{aligned}$$

Calcul de l'équidistribution

Dans ce qui suit, on montre la façon dont le module vérifie l'équidistribution pour les générateurs utilisant des récurrences linéaires modulo 2. Dans cette section, on montre que le calcul du rang d'une matrice permet de trouver les valeurs de ℓ_t et de t_ℓ. Tous les calculs se font dans $\mathbb{F}_2$.

À partir des vecteurs de sortie du générateur $\mathbf{y}_n$, construisons le vecteur ligne de $t\ell$ bits :

$$\mathbf{s}_{t,\ell} = (\mathrm{trunc}_\ell(\mathbf{y}_0)^T, .., \mathrm{trunc}_\ell(\mathbf{y}_{t-1})^T)$$

Des équations (1),(2) et (3), on peut observer que chaque valeur de $y_n^{(i)}$ peut être exprimée comme une combinaison linéaire de $x_0^{(0)}, .., x_0^{(k-1)}$. On obtient

$$y_n^{(i)} = \sum_{g=0}^{k-1} b_{n,i,g} x_0^{(g)} \tag{8}$$

où $b_{n,i,g} \in \mathbb{F}_2$. Le vecteur $\mathbf{s}_{t,\ell}$ peut être exprimé par

$$\mathbf{s}_{t,\ell} = \mathbf{x}_0 B_{t,\ell}, \tag{9}$$

où $B_{t,\ell}$ est une matrice $k \times t\ell$, avec éléments binaires, qui peut être définie comme suit : si $y_n^{(i)}$ est le g-ième élément de $\mathbf{s}_{t,\ell}$, alors la g-ième colonne de la matrice est $(b_{n,i,0}, .., b_{n,i,k-1})^T$. La proposition suivante donne une condition nécessaire pour vérifier si le générateur est (t, ℓ)-équidistribué. Le module utilise l'algorithme donné dans [2] pour construire la matrice $B_{t,\ell}$.

Proposition A.1 *(L'Écuyer[2])*
Le générateur est (t, ℓ)-équidistribué si et seulement si la matrice $B_{t,\ell}$ est de plein rang.

Vérifier l'équidistribution pour des générateurs combinés

On peut combiner plusieurs générateurs afin d'améliorer l'équidistribution comparativement à l'équidistribution des composantes individuelles. On peut ainsi combiner J générateurs avec l'équation (5). De l'équation (8), on obtient pour chaque composante

$$y_{n,j}^{(i)} = \sum_{g=0}^{k_j-1} b_{n,j,i,g} x_{0,j}^{(g)}$$

où $b_{n,j,i,g} \in \{0, 1\}$. Le vecteur de bits

$$\mathbf{s}_{j,t,\ell} = (\mathrm{trunc}_\ell(\mathbf{y}_{0,j})^T, \ldots, \mathrm{trunc}_\ell(\mathbf{y}_{t-1,j})^T)$$

peut être exprimé par

$$\mathbf{s}_{j,t,\ell} = \mathbf{x}_{0,j} B_{j,t,\ell}$$

où $B_{j,t,\ell}$ est une matrice à éléments binaires de dimension $k_j \times t\ell$. Ses éléments peuvent être calculés avec les méthodes décrites dans la section précédente en considérant chaque composante indépendamment.

On peut observer que

$$y_n^{(i)} = y_{n,1}^{(i)} + \cdots + y_{n,J}^{(i)} = \sum_{j=1}^{J} \sum_{g=0}^{k_j-1} b_{n,j,i,g} y_{0,j}^{(i)}$$

pour chaque valeur de n et i possibles. On obtient

$$\mathbf{s}_{t,\ell} = \sum_{j=1}^{J} \mathbf{s}_{j,t,\ell} = \sum_{j=1}^{J} \mathbf{x}_{0,j} B_{j,t,\ell}$$

que l'on peut réécrire de la façon suivante :

$$\mathbf{s}_{t,\ell} \equiv \tilde{\mathbf{x}}_0 \tilde{B}_{t,\ell}$$

où $\tilde{B}_{t,\ell}$ est une matrice $k \times t\ell$, qui est définie comme la juxtaposition verticale des B_{j,t,ℓ,s_j}, $k = k_1 + \cdots + k_J$ et $\tilde{\mathbf{x}}_0$, est obtenue en juxtaposant les $\mathbf{x}_{0,j}$ horizontalement. On obtient une proposition similaire à celle de la section précédente, sauf que celle-ci est plus générale. Comme dans le cas du générateur simple, il est possible de déterminer si le générateur est (t,ℓ)-équidistribué en trouvant le rang d'une matrice.

Proposition A.2 *(L'Écuyer[2]) Un générateur combiné est (t,ℓ)-équidistribué si et seulement si la matrice $\tilde{B}_{t,\ell}$ a plein rang $t\ell$.*

Vérifier la propriété "sans-collision" ou CF

Cette propriété a été introduite dans [2] et une extension à la propriété ME. Si on obtient un générateur qui est ME, alors, pour les dimensions t qui divisent k, chacun des 2^k petits cubes en dimension t et en résolution $\ell_t = k/\ell < L$ contient exactement un point de Ω_t. Mais, si t ne divise pas k et $\ell_t < L$, alors chacun des petits cubes contient exactement 2^{λ_t} points où $\lambda_t = k - t\ell_t$. Si $\ell_t + 1 \leq L$ et si on partionne l'hypercube unitaire à t dimensions en $2^{t(\ell_t+1)}$ petits cubes, alors si les petits cubes non vides contiennent exactement un point, alors on dit que la séquence est sans-collision en dimension t, ou CF(t). Si cela tient pour toutes les dimensions t pour lesquelles $\ell_t < L$, alors on dit que la séquence est sans-collision, ou CF (de l'anglais "collision-free").

Afin de vérifier la propriété CF, il faut définir les ensembles suivants :

$$\Phi_3 = \{t \mid 2 \leq t \leq ;\ \ell_t = \lfloor k/t \rfloor < k/t \leq L\},$$
$$\Phi_4 = \{t \in \Phi_3 \mid k \mod (\ell_t + 1) \neq 0 ;\ \ell_{t-1} > \ell_t\}.$$

Les deux prochaines propositions nous permettent de vérifier la propriété CF.

Proposition A.3 *(L'Écuyer[2])*
Un générateur ME est aussi CF si et seulement si, pour tout $t \in \Phi_4$, il est CF(t).

Proposition A.4 *(L'Écuyer[2])*
Si $\ell_t = \lfloor k/t \rfloor < k/t \leq L$, alors le générateur est CF(t) si et seulement si la matrice $\tilde{B}_{t,\ell_t+1}$ est de rang k.

On définit l'écart CF en dimension t par la différence entre k et le rang de la matrice $\tilde{B}_{t,\ell_t+1}$, soit $\epsilon_t = k - \mathrm{rang}(\tilde{B}_{t,\ell_t+1})$.

Ce que le module mecf permet de faire

La fonction `PrepareMat()` construit la matrice $\tilde{B}_{t,\ell}$ pour le générateur combiné courant de la variable de type `Combinaison` mise en paramètre. Lors de l'appel de `TestME()`, cette fonction calcule les valeurs de Δ_ℓ pour $\ell \in \Psi_{12}$. Les valeurs de Δ_ℓ sont contenues dans le tableau `ecart` de la structure de type `paramMecf`.

La fonction `TestCF()` permet de vérifier si un générateur qui est ME est aussi CF. La fonction calcule le rang de $\tilde{B}_{t,\ell_t+1}$ pour toutes les dimensions $t \in \Phi_4$. Dans la structure `paramMecf`, il y a la variable `EcartCF` dans la structure de type `paramMecf` qui contient les valeurs de ϵ_t pour chacune des dimensions.

Une fois `TestME()` ou `TestCF()` appelé, il est possible de vérifier certains aspects de l'équidistribution. Ainsi, on dit qu'un générateur est quasi-ME(S), si $\sum_{\ell \in \Psi_{12}} \Delta_\ell < S$ où S est une borne supérieure que l'on peut accepter sur cette quantité. On dit également que le générateur est presque-ME(S,δ), si le générateur est quasi-ME(S) et si $\Delta_\ell < \delta_\ell$ où les δ_ℓ sont des bornes supérieures sur les valeurs de Δ_ℓ que l'on peut accepter. Les fonctions qui permettent de vérifier ces propriétés sont les fonctions `estQuasiME()` et `estPresqueME()`. La fonction `estME()` indique si le générateur est ME ou non.

Pour la propriété CF, on dit que le générateur est quasi-CF(T), si $\sum_{t \in \Phi_4} \epsilon_t < T$ où T est une borne supérieure que l'on peut accepter sur cette quantité. La fonction `estQuasiCF()` permet de vérifier cette propriété. La fonction `estCF()` permet de vérifier si le générateur est CF ou non.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "btypes.h"
#include "etages.h"
#include "lintrans.h"
#include "baseopera.h"
```

```
typedef boolean   Ensemble[K_MAX+1] ;
```

Permet de définir les ensembles Ψ_{12}, Φ_{12} et Φ_4. Si on déclare une variable `S` de ce type pour représenter un ensemble S, alors si $x \in S$, ceci implique que `S[`x`]==TRUE`.

```
typedef struct{
  int k ;
  int L ;
  int maxdim ;
  Ensemble phi12 ;
  Ensemble phi4 ;
  Ensemble psi12 ;
  int ecart[MAXL+1] ;
  int delta[MAXL+1] ;
  int Lambda[K_MAX+1] ;
  int ecartCF[K_MAX+1] ;
  int se ;
  int secf ;
  int mse ;
  int msecf ;
} paramMecf ;
```

Type définissant les paramètres de l'évaluation de l'équidistribution et de la propriété CF. Les variable `k` et `L` sont le degré et la résolution à la sortie du générateur combiné sur lequel on vérifie l'équidistribution. La variable `maxdim` représente la dimension maximale pour laquelle on a vérifié l'équidistribution. La variable `phi12` représente l'ensemble Φ_{12}. La variable `phi4` représente l'ensemble Φ_4. La variable `psi12` représente l'ensemble Ψ_{12}. La variable `ecart` est un tableau qui contient Δ_ℓ dans `ecart[`ℓ`]`. La variable `delta` est un tableau qui contient δ_ℓ dans `delta[`ℓ`]`. La variable `Lambda` contient les valeurs de Λ_t. La variable `ecartCF` contient les écarts ϵ_t pour la propriété CF. La variable `se` est la somme des `ecart[i]`, c'est-à-dire `se` $= \sum_{\ell \in \Psi_{12}} \Delta_\ell$. La variable `secf` est la somme des `ecartCF[i]`, c'est-à-dire `se` $= \sum_{t \in \Phi_4} \epsilon_t$. La variable `mse` est la valeur de S et la variable `msecf` est la valeur de T.

```
boolean estPresqueME ( paramMecf *params
                     ) ;
```

Indique si le générateur pour lequel on a vérifié l'équidistribution est presque-ME(S,δ). Retourne `TRUE` si c'est le cas, `FALSE` sinon.

```
boolean estME ( paramMecf *params
              ) ;
```

Indique si le générateur pour lequel on a vérifié l'équidistribution est ME. Retourne `TRUE` si c'est le cas, `FALSE` sinon.

```
boolean estQuasiCF ( paramMecf *params
                   ) ;
```

Indique si le générateur pour lequel on a vérifié l'équidistribution est quasi-CF(T). Retourne `TRUE` si c'est le cas, `FALSE` sinon.

```
boolean estCF ( paramMecf *params
              ) ;
```

Indique si le générateur pour lequel on a vérifié l'équidistribution est CF. Retourne `TRUE` si c'est le cas, `FALSE` sinon.

```
boolean estQuasiME ( paramMecf *params
                   ) ;
```

Indique si le générateur pour lequel on a vérifié l'équidistribution est quasi-ME(S). Retourne `TRUE` si c'est le cas, `FALSE` sinon.

```
void InitParamMecf ( paramMecf *params,
                     int delta[MAXL+1],
                     int mse,
                     int msecf
                   ) ;
```

Fonction qui initialise une variable de type `paramMecf`. Il permet d'initialiser le tableau des écarts acceptables δ_ℓ avec la variable `delta`. Les variables `mse` et `msecf` indiquent les valeurs maximales de $\sum \ell \in \Psi_{12}$ et de la somme des écarts CF.

```
void TestME ( Combinaison *C,
              paramMecf *params,
              int indmax
            ) ;
```

Vérifie l'équidistribution du générateur courant de `*C`. La valeurs de Δ_ℓ sont calculées pour $\ell \in \Psi_{12}$. La variable pointée par `*params` est mise à jour par cette fonction. La variable `indmax` indique la dimension maximale pour laquelle on désire vérifier l'équidistribution.

```
void TestCF ( Combinaison *C,
              paramMecf *params
            ) ;
```

Fonction qui calcule les valeurs de ϵ_t du générateur combiné courant de `*C`. Le paramètre `*params` est mis à jour par cette fonction.

```
void PrepareMat ( Combinaison *C,
                  Matrice *Mat,
                  int indice_max
                ) ;
```

Fonction qui permet de préparer la matrice de travail `*Mat` selon le générateur combiné courant de `*C`. Le paramètre `indice_max` indique la dimension maximum pour laquelle on désire construire la matrice.

```
void AffTable ( paramMecf *params,
                 char type
              ) ;
```

Fonction qui permet d'afficher, soit les écarts Δ_ℓ de la variable `*params` si `type` contient le caractère 'l'. S'il contient le caractère 't', alors la fonction affiche les écarts Λ_t.

```
void CalculEnsembles ( Combinaison *C,
                       paramMecf *params,
                       int indmax
                     ) ;
```

Fonction qui calcule les ensembles Φ_{12}, Φ_4 et Ψ_{12} pour la vérification de l'équidistribution et de la propriété CF. Cette fonction est appelée automatiquement par `TestME()` et `TestCF()`. Peut être utile si l'on décide de construire d'autres fonctions spécialisées de `TestME()` (par exemple, `TemperMKTestME()` dans le module temperMK). La variable `indmax` indique la dimension maximale pour laquelle on désire vérifier l'équidistribution.

```
void ConvEcarts ( paramMecf *params
                );
```

Fonction qui, une fois `TestME()` appelée, permet de calculer les valeurs de Λ_t à partir des valeurs de Δ_ℓ. Cette fonction est nécessaire si l'on veut afficher les valeurs de Λ_t avec la fonction `DispTable()`.

```
void AffME ( paramMecf *params
           );
```

Fonction qui affiche, si l'équidistribution montre la propriété ME, un message indiquant que le générateur est ME.

```
void AffCF ( paramMecf *params
           );
```

Fonction qui affiche, si l'équidistribution montre la propriété CF, un message indiquant que le générateur est CF.

tuplets

Dans le module mecf, on présente l'ensemble de points Ω_t et la notion d'équidistribution du générateur qui génère cet ensemble de points. Cet ensemble de points est généré par les valeurs successives produites par le générateur. Pour la simulation et l'intégration quasi-Monte Carlo (afin de réduire l'erreur d'intégration) il est nécessaire de regarder l'équidistribution dans plusieurs ensembles de points formés par des t-tuplets de valeurs non successives du générateur. Dans [1], on donne le critère suivant qui est une généralisation du critère d'équidistribution.

On définit

$$\Delta_{t_1,\ldots,t_d} = \max\left(\max_{I=\{1,\ldots,s\},s=1,\ldots,t_1}(\ell^*_{|I|}(n)-\ell_I), \max_{I\in S(t_s,s),s=2,\ldots,d}(\ell^*_{|I|}(n)-\ell_I)\right), \tag{10}$$

où $S(t_s,s)=\{\{i_1,\ldots,i_s\}, 1=i_1<\ldots<i_s\leq t_s\}$. La quantité $\ell^*_{|I|}(n)$ est donnée par $\lfloor k/|I|\rfloor$ et correspond à la résolution maximale pour un ensemble de $n=2^k$ points dans $[0,1)^{|I|}$. L'ensemble I est un ensemble d'indices de dimensions.

La première partie du critère, soit

$$\max_{I=\{1,\ldots,s\},s=1,\ldots,t_1}(\ell^*_{|I|}(n)-\ell_I),$$

détermine l'écart maximal entre la résolution maximale, $\ell^*_{|I|}(n)$, et la résolution obtenue, ℓ_I, en prenant les ensembles de points produits par des s-tuplets de valeurs successives du générateur, soit

$$\{\mathbf{u}_s=(u_0,\ldots,u_{s-1}) : \mathbf{x}_0\in \mathbb{F}_2^k\}.$$

pour $s=1,\ldots,t_1$.

La deuxième partie du critère, soit

$$\max_{I\in S(t_s,s),s=2,\ldots,d}(\ell^*_{|I|}(n)-\ell_I),$$

détermine également l'écart maximal entre la résolution maximale et la résolution obtenue mais en prenant des ensembles de points de valeurs non successives. Ces ensembles de points sont

$$\{\mathbf{u}=(u_{i_1},\ldots,u_{i_s}) : \mathbf{x}_0\in \mathbb{F}_2^k\},$$

pour $s=2,\ldots,d$ et $1=i_1<\ldots<i_s\leq t_s$. Le paramètre t_s permet de choisir, pour s indices, la valeur maximale de i_s et, par le fait même, de contrôler le nombre d'ensembles de points à vérifier. La valeur $\Delta_{t_1,\ldots,t_d}$ est la valeur maximale de $\ell^*_{|I|}(n)-\ell_I$ qu'on obtient dans les deux parties.

Il est important de voir que ce critère $\Delta_{t_1,\ldots,t_d}$ généralise la propriété ME parce que le générateur est ME si et seulement si $\Delta_k=0$. Dans ce cas, on obtient

$$\Delta_k = \max\left(\max_{I=\{1,\ldots,s\},s=1,\ldots,k}(\ell^*_{|I|}(n)-\ell_I)\right).$$

Dans la définition que l'on a donnée d'un générateur ME, on dit qu'il est ME si $\Lambda_s = 0$ pour $s = 1, \ldots, k$. Dans ce cas-ci, $\Lambda_s = \ell^*_{\{1,\ldots,s\}}(n) - \ell_{\{1,\ldots,s\}}$, d'où l'équivalence entre $\Delta_k = 0$ et la propriété ME.

Ce module permet de vérifier le critère $\Delta_{t_1,\ldots,t_d}$ et d'afficher les écarts obtenus pour la deuxième partie du critère. Les écarts obtenus pour la première partie du critère peuvent être affichés avec la fonction `AffTable()` du module mecf.

```
#include <stdlib.h>
#include <stdio.h>
#include "btypes.h"
#include "etages.h"
#include "lintrans.h"
#include "baseopera.h"
#include "mecf.h"


typedef struct{
  paramMecf pmecf ;
  int tupd ;
  int tupt[K_MAX+1] ;
  int DELTA[K_MAX+1] ;
  float pourcentage[K_MAX+1] ;
  int DD ;
  int max ;
} paramTuplets ;
```

La variable `tupd` représente d. La variable `tupt` est un tableau qui contient dans `tupt[`$i-1$`]` la valeur de t_i. Le tableau `DELTA` contient les écarts de résolution maximaux qu'on a obtenus pour chacune des dimensions dans la deuxième partie du critère et `pourcentage[]` contient le pourcentage des projections pour lesquelles l'écart de résolution était 0. La variable `DD` représente la valeur du critère $\Delta_{t_1,\ldots,t_d}$ pour le générateur courant et `max` est la valeur maximale que l'on peut accepter pour ce critère.

```
void InitParamTuplets ( paramTuplets *params,
                        int d,
                        int t[K_MAX+1],
                        int max
                      ) ;
```

Fonction qui initialise les paramètres du critère à vérifier.

```
boolean estTuplets ( paramTuplets *params
                   ) ;
```

Fonction qui indique que la valeur du critère est plus petite ou égale au maximum acceptable. Retourne `TRUE` si c'est le cas, `FALSE` sinon. Si le critère n'a pas été vérifié, alors la fonction retourne `TRUE`.

```
void tuplets ( Combinaison *C,
               paramTuplets *params
             ) ;
```

Fonction qui vérifie le critère $\Delta_{t_1,\ldots,t_d}$ pour le générateur courant de `*C` avec les paramètres indiqués par `*params`. Afin de calculer la première partie du critère la fonction `TestME()` du module mecf est utilisée.

```
void AffTuplets ( paramTuplets *params
                );
```

Fonction qui affiche les écarts obtenus par la première partie du critère avec la fonction `AffTable()` du module mecf. Elle affiche aussi les écarts obtenus par la deuxième partie du critère dans un autre tableau. Dans ce deuxième tableau, elle affiche le pourcentage des projections qui ont eu des écarts en résolution de 0 et l'écart maximum pour les projections dans chacune des dimensions.

```
paramMecf* PointeurMecf ( paramTuplets *params
                        );
```

Fonction qui retourne un pointeur vers les paramètres d'équidistribution de type `*paramMecf`.

tausworthe

Ce module s'occupe de tout ce qui se rapporte aux générateurs de type Tausworthe. Ces générateurs ont deux paramètres : un entier s et un polynôme caractéristique

$$P(z) = z^{q_{n-1}} + z^{q_{n-2}} + \ldots + z^{q_0}$$

où n est le nombre de coefficients non nuls et $q_0 = 0$.

La récurrence employée est

$$x_j = a_1 x_{j-1} + \cdots + a_k x_{j-k} \tag{11}$$

où $a_{k-p} = 1$ si $p \in \{q_0, \ldots, q_{n-1}\}$, sinon $a_{k-p} = 0$.

L'état du générateur à la n-ième itération est

$$\mathbf{x}_n = (x_{ns}, x_{ns+1}, \ldots, x_{ns+k-1})$$

où s est un paramètre entier.

Le vecteur de bits de la sortie associé à cet état du générateur est

$$\mathbf{y}_n = (x_{ns}, x_{ns+1}, \ldots, x_{ns+L-1}).$$

Si on a que

$$q_{n-i} > q_{n-i-1} \text{ pour } i = 1, \ldots, n-1,$$

alors l'implantation de ce type de générateurs dans ce module a les restrictions suivantes :

$$0 < 2q_{n-1} < q_{n-2}, \;\; 0 < s \leq q_{n-1} - q_{n-2} < q_{n-1} \leq L.$$

Afin d'obtenir une période maximale, il est important que pgcd$(s, 2^{q_{n-1}})\neq 1$, mais ceci est pris en charge par la fonction `LireDonneesTaus()`, lors de l'initialisation de l'étage. Afin de représenter le polynôme caractéristique, on utilise un tableau unidimensionnel `Q[]` qui contient dans l'ordre $q_0, q_1, \ldots, q_{n-1}$. L'implantation utilisée dans ce module est inspirée de l'algorithme QuickTaus[2] et est décrite dans [8].

La fonction `InitParamTaus()` permet d'initialiser une implantation du générateur. On n'a qu'à mettre comme paramètres à cette fonction les paramètres du générateur. La fonction `InitTaus()` permet d'initialiser la racine du générateur. La fonction `Taus()` est l'implantation même du générateur et retourne la prochaine sortie du générateur.

La fonction `LireDonneesTaus()` permet de lire des données dans un fichier et de remplir un étage avec plusieurs générateurs de type Tausworthe. Le format de ce fichier est décrit dans la figure 19.

Voici l'explication des symboles de la figure 19 :

`N`	Nombre de générateurs définis
`ni`	Nombre de coefficients non nuls pour le polynôme de la `i`-ième ligne
`qij`	`j`-ième coefficient non nul du `i`-ième polynôme. Ceux-ci doivent être mis en ordre décroissant.

```
N
n1 q10 q11 q12 ..
..
nN qN0 qN1 qN2 ..
```

FIG. 19 – Fichier de données pour les générateurs de Tausworthe.

Il est à remarquer que le paramètre s n'est pas spécifié dans le fichier puisque la fonction `LireDonneesTaus()` trouve les valeurs de s valides et ajoute à l'étage un générateur pour chacune de ces valeurs de s.

REMARQUE IMPORTANTE : 2 fichiers sont essentiels pour le bon fonctionnement de cette partie du logiciel. Le fichier 'seektaus.dt1' contient, à la ligne n, les facteurs de n. Ce fichier contient `NBLIGNES` (constante définie dans btypes.h) lignes. Le fichier seektaus.dt2 contient, à la ligne n, les facteurs de $2^n - 1$, qui sont inférieurs à `NBLIGNES`. La valeur de `NBLIGNES` doit être inférieure ou égale à `MAXL`. Si on décide de modifier un des deux fichiers, alors il faudra modifier la valeur de `NBLIGNES` et de `MAXL`. Présentement, `NBLIGNES` est fixé à 256, ce qui veut dire que l'on ne peut avoir des composantes de générateurs avec polynôme caractéristique ayant un degré supérieur à 256 (k <`MAXL`). Ces fichiers permettent de trouver les valeurs du paramètre s valide.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "btypes.h"
#include "etages.h"
#include "baseopera.h"


#define MAXNBCOEFF 100
```

Nombre maximum de coefficients des polynômes que l'on veut vérifier pour les générateurs de type Tausworthe.

```
typedef struct {
  int k ;
  int Q[MAXNBCOEFF] ;
  int s ;
  int NbCoeff ;
  int gen_kms ;
  BitVect A ;
  BitVect gen_masque ;
} paramTaus ;
```

Type définissant une structure qui permet d'emmagasiner les paramètres d'un générateur de type Tausworthe. Le polynôme que définit cette structure est $x^{\mathtt{k}} + x^{\mathtt{Q[NbCcoeff-2]}} + .. + x^{\mathtt{Q[1]}} + x^{\mathtt{Q[0]}}$ et `s` est le paramètre s de la récurrence. A noter que `Q[NbCoeff-1]` a la même valeur que `k` et qu'on devrait avoir `Q[0]`$==0$. Le vecteur de bits `A` est l'état du générateur plus les autres bits de la récurrence jusqu'à concurrence de `MAXL` bits au total. Le vecteur de bits `gen_masque` est nécessaire pour l'exécution de `Taus`. La variable `gen_kms` contient le résultat de `k`$-$`s`.

```
char* TausNom ( void
              ) ;
```

Fonction qui retourne une chaîne de caractère contenant le nom du type de générateur.

```
void InitParamTaus ( paramTaus *params,
                     int k,
                     int Q[MAXNBCOEFF],
                     int s,
                     int NbCoeff
                   ) ;
```

Sert à initialiser les paramètres du générateur pointé par `*params`.

```
void AjoutTaus ( Etage *E,
                 paramTaus *params,
                 int L
               ) ;
```

Fonction qui ajoute à l'étage `E` un générateur de type Tausworthe avec les paramètres donnés par `params`. Le paramètre `L` indique le nombre de bits de résolution de la composante à la sortie.

```
void InitTaus ( void *params,
                int i,
                BitVect retour
              ) ;
```

Cette fonction initialise l'état du générateur `*params` avec le vecteur de bits `B_C[i]`. Le vecteur de bits `retour` contient `B_C[i]`, mais les `MAXL-k` prochains bits sont déterminés par la récurrence du générateur.

```
void Taus ( void *params,
            BitVect gen_etat
          ) ;
```

Implantation du générateur de type Tausworthe. Retourne dans `gen_state` le prochain état du générateur `*params` suivi des `MAXL-k` bits suivants de la récurrence.

```
void AffTaus ( void *params
             ) ;
```

Affiche les paramètres du générateur.

```
void LireDonneesTaus ( Etage *E,
                       char *nomfich,
                       boolean meme,
                       int L
                     ) ;
```

Permet de lire les données relatives aux générateurs de type Tausworthe dans le fichier `nomfich` et remplit en conséquence l'étage `*E`. La variable `meme` indique que le fichier de données est le même que pour l'étage précédent si l'étage `E` fait partie d'une structure de type `Combinaison`. Voir dans le préambule de la description de ce module le format que devrait avoir le fichier `nomfich`. Le paramètre `L` indique le nombre de bits de résolution de la composante à la sortie.

polynomial

Ce module s'occupe des générateurs de type polynomiaux. Les générateurs de ce type n'ont qu'un seul paramètre : un polynôme caractéristique

$$P(z) = z^k + a_1 z^{k-1} + ... + a_k$$

où les $a_i \in \mathbb{F}_2$.

La récurrence de ce type de générateur est

$$\mathbf{x}_n = X\mathbf{x}_{n-1} \tag{12}$$

où

$$X = \begin{pmatrix} a_1 & 1 & 0 & 0 & \dots & 0 \\ a_2 & 0 & 1 & 0 & \dots & 0 \\ a_3 & 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{k-1} & 0 & 0 & 0 & \dots & 1 \\ a_k & 0 & 0 & 0 & \dots & 0 \end{pmatrix}. \tag{13}$$

Dans ce module, on représente le polynôme caractéristique par le vecteur de bits $\mathbf{a} = (a_1, \dots, a_k)$. Voir [8] et [4] pour plus de détails sur ces générateurs.

La fonction `InitParamPoly()` initialise les paramètres du générateur tandis que `InitPoly()` initialise l'état du générateur. La fonction `Poly()` retourne le prochain état du générateur courant tandis que `AffPoly()` affiche à l'écran les coefficients non nuls du générateur.

La fonction `LireDonneesPoly()` lit dans un fichier les données de générateurs polynômiaux à mettre dans un étage. Le format du fichier est décrit dans la figure 20.

```
p11 p12 ..  0
p21 p22 ..  0
..
pN1 pN2 ..  0
```

FIG. 20 – Fichier de données pour les GCL polynomiaux.

Chaque ligne de la figure 20 représente un polynôme

$$x^{\mathtt{pi1}} + x^{\mathtt{pi2}} + .. + 1$$

Sur chaque ligne, on retrouve le degré des coefficients non nuls en terminant par le degré 0. Le programme ne permet pas de lire plus que `MAXNBPOLY` polynômes à la fois.

```
#include <stdio.h>
#include <stdlib.h>
#include "btypes.h"
#include "etages.h"
#include "baseopera.h"
```

```
#define MAXNBPOLY 100
```

Nombre maximum de polynome défini dans le fichier de données lu par `LireDonneesPoly()`.

```
typedef struct {
  BitVect Polynome ;
  int Deg ;
  BitVect gen_etat ;
  BitVect gen_masque ;
}paramPoly ;
```

Type définissant une structure qui permet d'emmagasiner les paramètres d'un générateur de type polynômial. Le vecteur de bits `Polynome` contient le polynôme caractéristique de la récurrence et `Deg` contient le degré de ce polynôme. Le vecteur de bit `gen_state` contient l'état courant du générateur et `gen_mask` contient un masque nécessaire pour l'exécution de `Poly()`.

```
char* PolyNom ( void
              ) ;
```

Fonction qui retourne une chaîne de caractères contenant le nom du type de générateur.

```
void InitParamPoly ( paramPoly *params,
                     int degre,
                     BitVect Polynome
                   ) ;
```

Sert à initialiser les paramètres du générateur pointé par `params`.

```
void AjoutPoly ( Etage *E,
                 paramPoly *params,
                 int L
               ) ;
```

Fonction qui ajoute à l'étage `*E` un générateur de type GCL polynomial avec les paramètres donnés par `*params`. Le paramètre `L` indique le nombre de bits de résolution de la composante à la sortie.

```
void InitPoly ( void *params,
                int i,
                BitVect retour
              ) ;
```

Cette fonction initialise l'état du générateur `*params` avec le vecteur de bits `B_C[i]`. Le vecteur de bits `retour` devient une copie de `B_C[i]`.

```
void Poly ( void *params,
            BitVect g_etat
          ) ;
```

Implantation du générateur de type GCL polynomial. Effectue une itération sur le générateur pointé par `params`. Retourne dans `g_etat` le prochain état du générateur.

```
void AffPoly ( void *params
             ) ;
```

Affiche les paramètres du générateur.

```
void LireDonneesPoly ( Etage *E,
                       char *nomfich,
                       boolean meme,
                       int L
                     ) ;
```

Permet de lire les données relatives aux générateurs de type GCL polynomial dans le fichier `nomfich` et remplit en conséquence l'étage `*E`. La variable `meme` indique que le fichier de données est le même que pour l'étage précédent si l'étage `*E` fait partie d'une structure de type `Combinaison`. Voir dans le préambule de la description de ce module le format que devrait avoir le fichier `nomfich`. Le paramètre `L` indique le nombre de bits de résolution de la composante à la sortie.

tgfsr

Ce module permet de gérer tout ce qui touche aux générateurs de type TGFSR. Ces générateurs ont les paramètres suivants : r, m, w et $\mathbf{a}$. Les trois premiers paramètres sont des entiers tandis que $\mathbf{a}$ est un vecteur de w bits.

Ce générateur est basé sur la récurrence

$$\mathbf{v}_n = \mathbf{v}_{n+m-r} \oplus A\mathbf{v}_{n-r}, \tag{14}$$

où A est une matrice binaire de dimension $w \times w$ et les $\mathbf{v}_i$ sont vus comme des vecteurs-colonnes. La matrice A, pour l'implantation de ce module, est

$$\begin{pmatrix} & & & & a_0 \\ 1 & & & & a_1 \\ & 1 & & & a_2 \\ & & \ddots & & \\ & & & 1 & a_{w-1} \end{pmatrix}.$$

On définit, pour ce module, le vecteur de bits $\mathbf{a} = (a_0, \ldots, a_{w-1})$. Ce vecteur définit pleinement la matrice A.

L'état du générateur à la n-ième itération est $\mathbf{x}_n = (\mathbf{v}_n^T, \mathbf{v}_{n-1}^T, \ldots, \mathbf{v}_{n-r+1}^T)$. Voir [5], [6] et [8] pour plus de détails sur ce type de générateurs.

Les paramètres sont initialisés par la fonction `InitParamTGFSR()`. La fonction `TGFSR()` retourne la prochaine sortie du générateur. La fonction `AffTGFSR()` permet d'imprimer les paramètres du TGFSR. La fonction `LireDonneesTGFSR()` lit dans un fichier les données des générateurs à explorer. Le format du fichier est donné à la figure 21.

```
w r
'a' na a1 a2 ..
['A' na a1 a2 ..]
'm' nm m1 m2 ..
```

FIG. 21 – Fichier de données pour les TGFSR.

Voici une explication des symboles de la figure 21 :

`w`	Paramètre w
`r`	Paramètre r
`'a'`	Indique que les valeurs de $\mathbf{a}$ seront données sous forme hexadécimale
`'A'`	Indique que les valeurs de $\mathbf{a}$ seront données sous forme décimale
`na`	Nombre de valeurs de $\mathbf{a}$ spécifiées
`ai`	i-ième valeur de $\mathbf{a}$ spécifiée
`nm`	Nombre de valeurs de m spécifiées
`mi`	i-ième valeur de m spécifiée

Il est à remarquer que ce fichier n'a que 3 lignes et que l'on doit commencer la deuxième ligne par un 'a' ou un 'A' qui détermine la forme sous laquelle les valeurs de **a** sont données. Il est fortement recommandé d'utiliser la forme hexadécimale. Si l'on utilise la forme décimale, alors il est important de savoir que l'ordre des bits est inversé, par rapport au même chiffre mis en notation hexadécimale. Ceci permet d'être compatible avec les paramètres trouvés par Armand Nganou. La troisième ligne doit commencer par un 'm'. Dans le fichier tgfsr.c, il y a la constante `Max_r` qui indique la valeur maximale que r peut prendre. Pour l'instant la valeur maximale de w est de 32 pour l'implantation du TGFSR de ce module.

```
#include <stdio.h>
#include <stdlib.h>
#include "btypes.h"
#include "etages.h"
#include "baseopera.h"


#define MAX_R 32
```

Valeur maximal de r dans l'implantation du TGFSR.

```
typedef BitVect TGEtat[MAX_R] ;
```

Type contenant l'état d'un TGFSR. Ainsi, une variable `etat` déclarée comme de type `TabSeed` contient dans `etat[`n`]` le vecteur de bits $\mathbf{v}_n$.

```
typedef struct {
  int w ;
  int r ;
  int m ;
  BitVect a ;
  BitVect gen_masque ;
  TGEtat gen_etat ;
  int  l ;
}paramTGFSR ;
```

Type définissant une structure qui permet d'emmagasiner les paramètres d'un générateur de type TGFSR. Le vecteur de bits `gen_masque` est nécessaire pour l'exécution de `TGFSR()`. La variable `gen_etat` contient l'état du générateur. La variable `l` contient le numero de l'état modulo `gen_r`.

```
char* TGFSRNom ( void
               ) ;
```

Fonction qui retourne une chaîne de caractères contenant le nom du type de générateur.

```
void InitParamTGFSR ( paramTGFSR *params,
                      int w,
                      int r,
                      int m,
                      BitVect A
                    ) ;
```

Sert à initialiser les paramètres du TGFSR.

```
void AjoutTGFSR ( Etage *E,
                  paramTGFSR *params,
                  int L
                );
```

Fonction qui ajoute à l'étage `*E` un générateur de type TGFSR avec les paramètres donnés par `*params`. Le paramètre `L` indique le nombre de bits de résolution de la composante à la sortie.

```
void InitTGFSR ( void *params,
                 int i,
                 BitVect retour
               );
```

Cette fonction initialise l'état du générateur `*params` avec le vecteur de bits `B_C[i]`. Les premiers w bits de l'état sont copiés dans `retour`.

```
void  TGFSR ( void *params,
              BitVect RETOUR
            );
```

Implantation du générateur `*params` de type TGFSR. Retourne dans `RETOUR` les `w` premiers bits du prochain état du générateur.

```
void AffTGFSR ( void *params
              );
```

Affiche les paramètres du générateur.

```
void LireDonneesTGFSR ( Etage *E,
                        char *nomfich,
                        boolean meme,
                        int L
                      );
```

Permet de lire les données, relatives aux générateurs de type TGFSR, dans le fichier `*nomfich` et remplit en conséquence l'étage `*E`. La variable `meme` indique que le fichier de données est le même que pour l'étage précédent si l'étage `E` fait partie d'une structure de type `Combinaison`. Voir dans le préambule de la description de ce module le format que devrait avoir le fichier `nomfich`. Le paramètre `L` indique le nombre de bits de résolution de la composante à la sortie.

temperMK

Le tempering de Matsumoto-Kurita a 5 paramètres w, η, μ, $\mathbf{b}$ et $\mathbf{c}$. Les paramètres w, η et μ sont des entiers et $\mathbf{b}$ et $\mathbf{c}$ sont des vecteurs de w bits.

Le tempering de Matsumoto-Kurita s'effectue sur les w premiers bits de $\mathbf{x}_n$ pour obtenir le vecteur de bits $\mathbf{z}_n$. On procède à l'aide des opérations suivantes :

1. $\overline{\mathbf{x}}_n = \mathrm{trunc}_w(\mathbf{x}_n)$
2. $\mathbf{r}_n = \overline{\mathbf{x}}_n \oplus ((\overline{\mathbf{x}}_n \ll \eta) \,\&\, \mathbf{b})$
3. $\mathbf{z}_n = \mathbf{r}_n \oplus ((\mathbf{r}_n \ll \mu) \,\&\, \mathbf{c})$

La fonction trunc_w ne retourne que les w premiers bits du vecteur de bits mis en argument. L'opération $\oplus$ est un ou-exclusif bit par bit, & est le 'ET'-logique bit par bit et $\ll \eta$ est un décalage de η bits vers la gauche. Voir [5], [6] et [8] pour plus de détails sur cette transformation linéaire.

Afin de créer un ensemble de paramètres de type `paramTempMK`, il suffit de déclarer une variable de ce type. Pour initialiser les champs de ce type, alors on utilise la fonction `InitParamTemperMK()`. Il faut choisir explicitement les paramètres η et μ. Les paramètres $\mathbf{b}$ et $\mathbf{c}$ peuvent être aléatoires, en mettant -1 dans le paramètre `Alea_bc`. Dans ce cas, il faut cependant appeler la fonction `MAJTemperMK()` pour les fixer. Les fonctions `MAJTrans()` et `MAJToutesTrans()` du module lintrans fixent les paramètres aléatoires, si cette transformation linéaire se trouve dans un `Etage`.

Afin d'effectuer ce type de transformation linéaire sur les composantes d'un étage, alors il suffit d'appeler la fonction `AjoutTemperMK()` avec l'étage concerné et les paramètres de la transformation linéaire en paramètres.

La fonction `TemperMKTestME()` est semblable à la fonction `TestME()` du module mecf. Comme cette dernière, elle permet de vérifier l'équidistribution, mais utilise l'algorithme d'optimisation du tempering de Matsumoto-Kurita pour générateurs combinés décrit dans [8]. Cet algorithme essaie de trouver des vecteurs de bits $\mathbf{b}$ et $\mathbf{c}$ afin que le générateur résultant soit tel que $\Delta_\ell \leq \delta_\ell$ pour $\ell = 1, \ldots, \min(k, L)$.

```
#include <time.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "btypes.h"
#include "etages.h"
#include "lintrans.h"
#include "baseopera.h"
#include "mecf.h"

#define TEMPMKID 1
#define TEMPNAME "Tempering MK"
```

```
typedef struct {
  int eta;
  int mu;
  int Alea_bc;
  BitVect b;
  BitVect b_original;
  BitVect c;
  BitVect c_original;
  BitVect BestB;
  BitVect BestC;
  boolean Optimise;
} paramTempMK;
```

Type définissant une structure qui permet d'emmagasiner les paramètres d'une transformation linéaire de type tempering de Matsumoto-Kurita. Les variables `eta` et `mu` définissent les paramètres η et μ de la transformation linéaire. Si la variable `Alea_bc` est mise à -1, alors ceci indique que les paramètres **b** et **c** de la transformation linéaire sont choisis au hasard. Les variables `b_original` et `c_original` sont les paramètres `b` et `c` de la fonction `InitTemperMK()` lors de l'appel de celle-ci. Par la suite, `b_original` et `c_original` ne sont plus changés. Les vecteurs de bits `b` et `c` de cette structure sont les paramètres **b** et **c** effectifs de la transformation. Ceux-ci sont modifiés (changés à une valeur aléatoire) à chaque appel de `MAJTemperMK()` si `Alea_bc` est -1. Si on met `Optimise` à `TRUE`, alors les valeurs de `b` et `c` de cette composante seront optimisées si l'on appelle la fonction `TemperMKTestME()`. Sinon, celles-ci ne sont pas affectées par la fonction. Les vecteurs de bits `BestB` et `BestC` sont utilisés seulement si on a fait appel à `TemperMKTestME()` et que `Optimise` est mis à `TRUE`. Il s'agit des meilleurs vecteurs **b** et et **c** rencontrés lors de l'optimisation du tempering de Matsumoto-Kurita. Pour les variables de ce type, il est important d'utiliser `InitTemperMK()` avant de l'utiliser avec des fonctions de ce module.

```
int DonnerTemperMKID ( void
                     );
```

Fonction qui donne le numéro distinctif du tempering de Matsumoto-Kurita. Le numéro pour cette transformation linéaire est le 1 (`TEMPMKID`).

```
void InitTemperMK  ( paramTempMK *params,
                     int eta,
                     int mu,
                     int Alea_bc,
                     BitVect b,
                     BitVect c,
                     boolean Optimise
                   );
```

Fonction qui initialise tous les champs de `*params`. Les champs `b_original`, `b` et `BestB` sont initialisés à la valeur du paramètre `b` de cette fonction, tandis que les champs `b_original`, `b` et `BestB` sont initialisés à la valeur du paramètre `c`. Les autres champs sont initialisés aux valeurs des paramètres de mêmes noms.

```
void AjoutTemperMK ( Etage *E,
                     paramTempMK *params
                   );
```

Fonction qui ajoute une transformation linéaire de type Tempering de Matsumoto-Kurita de paramètres spécifiés par `*params` à l'étage `*E`. Avant d'appeler cette fonction, il est important de spécifier les paramètres de `*params` à l'aide de la fonction `InitTemperMK()`.

```
void MAJTemperMK ( void *params,
                   int w
                 ) ;
```

Fonction qui met à jour la transformation linéaire spécifiée par `*params`. Lors de l'appel de `InitTemperMK()`, si `alea` est -1, alors `MAJTemper()` met dans `b` et `c` des vecteurs de bits aléatoires, sinon on prend les valeurs spécifiées par `b` et `c` qui ont été mises en paramètres dans la fonction `InitTemperMK()`. Le paramètre `w` indique le nombre de bits sur lesquels la transformation linéaire se fait.

```
void TemperingMK ( BitVect RES,
                   void *params,
                   int w
                 ) ;
```

Fonction qui effectue un tempering de Matsumoto-Kurita sur `w` premiers bits de `RES`. Les paramètres sont spécifiés par `params`.

```
void AffTemperMK ( void *params,
                    int w
                 ) ;
```

Fonction qui affiche les paramètres du tempering de Matsumoto-Kurita qui sont spécifiés par `*params`. Le paramètre `w` indique le nombre de bits sur lesquels est effectué le tempering de Matsumoto-Kurita.

```
void TemperMKTestME ( Combinaison *C,
                      paramMecf *params,
                      int max_v,
                      int affprog
                    ) ;
```

Même fonction que `TestME()` dans le module mecf sauf qu'on lance l'algorithme d'optimisation du tempering de Matsumoto-Kurita afin d'améliorer la valeur des écarts Δ_ℓ. L'algorithme d'optimisation est expliqué dans [8]. La variable `max_v` indique la valeur de ℓ jusqu'où la fonction `TempTestME()` doit optimiser le tempering de Matsumoto-Kurita. Si le paramètre `affprog` est mis à `TRUE`, alors la fonction `TempTestME()` affiche la progression de l'optimisation du tempering avec le numéro du bit v que l'algorithme est en train d'optimiser et la somme des écarts Δ_ℓ pour $\ell = 1, \ldots, v$.

REMARQUE IMPORTANTE : pour que l'optimisation fonctionne, il faut que le tempering de Matsumoto-Kurita soit la dernière transformation linéaire appliquée sur chacune des composantes. Également, l'algorithme ne fonctionne que sur les tempering Matsumoto-Kurita pour lesquels le paramètre `Optimise` est mis à `TRUE`.

selft

Le self-tempering a été introduit, pour la première fois, dans [8]. Le self-tempering implanté dans ce module n'a qu'un seul paramètre d (le paramère c est fixé à 32). Afin de représenter un mot de k bits $\mathbf{x}_n$, on a besoin de $K = \lceil k/32 \rceil$ mots de 32 bits : $\mathbf{x}_n^1, \mathbf{x}_n^2, .., \mathbf{x}_n^K$. Donc $\mathbf{x}_n$ est la juxtaposition des $\mathbf{x}_n^j$. La transformation se fait de la manière suivante :

1. $\mathbf{e} = (\bigoplus_{j=1}^{K} \mathbf{x}_n^j) \ll d$
2. $\mathbf{z}_n^j = \mathbf{x}_n^j \oplus \mathbf{e}$ pour $j = 1, \ldots, K-1$.
3. $\mathbf{z}_n^K = (\mathbf{x}_n^K \oplus \mathbf{e})$ & $\mathbf{m}_h$

où $h = k - 32(K-1)$, $\mathbf{e}$ est un vecteur de 32 bits et le résultat de la transformation sur $\mathbf{x}_n$ est le vecteur de bits $\mathbf{z}_n$, qui est la juxtaposition horizontale des $\mathbf{z}_n^j$ $(1 \leq j \leq K)$. L'opération $\oplus$ est un ou-exclusif bit par bit, & est le 'ET'-logique bit par bit et $\ll d$ est un décalage de d bits vers la gauche. Le symbole $\mathbf{m}_h$ représente un vecteur de bits composé de h uns suivis de zéros. Voir [8] et [4] pour plus de détails sur cette transformation linéaire.

Afin de créer un ensemble de paramètres de type `paramselft`, il faut déclarer une variable de ce type. Pour initialiser les champs de ce type, on utilise la fonction `InitParamPermut()`. On peut choisir explicitement le paramètre d ou mettre -1 pour ce paramètre si on le veut aléatoire. Dans ce cas, il faut cependant appeler la fonction `SetSelfT()` pour le fixer. Les fonctions `MAJTrans()` et `MAJToutesTrans()` du module lintrans fixent ce paramètre aléatoire si cette transformation linéaire se trouve dans un `Etage`.

Afin d'effectuer ce type de transformation linéaire sur les composantes d'un étage, alors il suffit d'appeler la fonction `AjoutSelfT()` avec l'étage concerné et les paramètres de la transformation linéaire en paramètres.

```
#include <time.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "btypes.h"
#include "etages.h"
#include "lintrans.h"
#include "baseopera.h"

#define SELFID 2
#define SELFNAME "Self-tempering"

typedef struct {
  int d;
  int d_original;
} paramselft;
```

Type définissant une structure qui permet d'emmagasiner les paramètres d'une transformation linéaire de type self-tempering. La variable `d_original` est le paramètre `d` de la fonction `InitSelfT()` lors de l'appel de celle-ci et la valeur de celle-ci n'est plus changée par la suite. La variable `d` de cette structure est le paramètre d effectif de la transformation. Celui-ci est modifié à chaque appel de `MAJSelfT()`, si `d_original`est -1.

```
int DonnerSelfTID ( void
                  );
```

Fonction qui retourne le numéro d'identification de la transformation linéaire self-tempering. Le numéro pour le self-tempering est 2 (SELFID).

```
void InitSelfT ( paramselft *params,
                 int d
               );
```

Fonction qui indique si les paramètres de la transformation linéaire seront choisis au hasard. Si d est -1, alors le paramètre d de la transformation linéaire sera choisi au hasard. Dans ce cas, avant d'utiliser *params, il est important d'utiliser MAJSelfT() avec *params comme paramètre afin de fixer la valeur de d. Si d est différent de -1, alors on peut utiliser *params sans avoir recours à MAJSelfT().

```
void AjoutSelfT ( Etage *E,
                  paramselft *params
                );
```

Fonction qui ajoute une transformation linéaire de type self-tempering avec paramètres définis par *params à l'étage *E. Avant d'appeler cette fonction, il est important de bien spécifier *params à l'aide de la fonction InitSelfT()

```
void MAJSelfT ( void *params,
                int w
              );
```

Fonction qui met de nouveaux paramètres dans *params pour les paramètres qui doivent être choisis au hasard. Le paramètre w indique le nombre de bits sur lesquels la transformation linéaire s'effectue.

```
void SelfT ( BitVect A,
             void *params,
             int w
           );
```

Fonction qui permet de faire un self-tempering avec les paramètres définis par *params sur le vecteur de bits A. Le paramètre w indique le nombre de bits sur lesquels la transformation linéaire se fait.

```
void AffSelfT ( void *params,
                 int w
              );
```

Fonction qui affiche les paramètres *params de la transformation linéaire effectuée sur w bits.

permut

Cette transformation linéaire consiste à changer la position des w premiers bits du vecteur $\mathbf{x}_n$ pour obtenir $\mathbf{z}_n$ de la façon suivante :

$$z_n^{(i)} = x_n^{(\pi(i))}, \quad 0 \le i < L-1 \tag{15}$$
$$\tag{16}$$

où $\pi : \mathbb{Z}_w \to \mathbb{Z}_w$ est bijectif. La fonction qui est utilisée est

$$\pi(i) = pi + q \mod w \text{ pour } 0 \le i < w.$$

Les deux paramètres de cette transformation linéaire sont p et q.

Afin de créer un ensemble de paramètres de type `parampermut`, il faut déclarer une variable de ce type. Pour initialiser les champs de ce type, on utilise la fonction `InitParamPermut()`. On peut choisir explicitement les paramètres p et q ou ne pas les définir en mettant -1 dans les paramètres que l'on veut aléatoires. Dans ce cas, il faut cependant appeler la fonction `MAJPermut()` pour les fixer. Les fonctions `MAJTrans()` et `MAJToutesTrans()` du module lintrans fixent les paramètres aléatoires si cette transformation linéaire se trouve dans un `Etage`.

Afin d'effectuer ce type de transformation linéaire sur les composantes d'un étage, il suffit d'appeler la fonction `AjoutPermut()` avec l'étage concerné et les paramètres de la transformation linéaire en paramètres.

```
#include <time.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "btypes.h"
#include "etages.h"
#include "lintrans.h"
#include "baseopera.h"

#define PERMID 3
#define PERMNAME "Permutation"

typedef struct {
  int p ;
  int p_original ;
  int q ;
  int q_original ;
} parampermut ;
```

Type définissant une structure qui permet d'emmagasiner les paramètres d'une transformation linéaire de type permutation. Les variables `p_original` et `q_original` sont les paramètres `p` et `q` de la fonction `InitPermut()` lors de l'appel de celle-ci. Les variables `p_original` et `q_original` ne sont plus changées par la suite. Les variables `p` et `q` de cette structure sont les paramètres p et q effectifs de la transformation. Celles-ci sont modifiées (changées à des valeurs aléatoires) à chaque appel de `MAJPermut()` si `p_original` est -1 et/ou `q_original` est -1.

```
int DonnerPermutID ( void
                   );
```

Fonction qui retourne le numéro d'identification de la transformation linéaire permutation. Le numéro de la permutation est 3 (`PERMID`).

```
void InitPermut ( parampermut *params,
                  int p,
                  int q
                );
```

Fonction qui indique si les paramètres de la transformation linéaire sont choisis au hasard. Si `p` est -1, alors le paramètre p de la transformation linéaire sera choisi au hasard. De la même manière, si `q` est -1, alors le paramètre q de la transformation linéaire est choisi au hasard. Dans ce cas, avant d'utiliser les paramètres définis par `*params`, il est important d'utiliser `MAJpermut()` avec `*params` comme paramètre afin de fixer les valeurs. Si `p` et `q` sont différents de -1, alors on peut utiliser `*params` sans avoir recours à `MAJpermut()`.

```
void AjoutPermut ( Etage *E,
                   parampermut *params
                 );
```

Fonction qui ajoute une transformation linéaire de type permutation avec paramètres définis par `*params` à l'étage `*E`. Avant d'appeler cette fonction, il est important de bien spécifier `*params` à l'aide de la fonction `InitPermut()`

```
void MAJPermut ( void *params,
                 int w
               );
```

Fonction qui met de nouveaux paramètres dans `*params` pour les paramètres qui doivent être choisis au hasard. Le paramètre `w` indique le nombre de bits sur lesquels la transformation linéaire s'effectue.

```
void Permut ( BitVect A,
              void *params,
              int w
            );
```

Fonction qui permet de faire une permutation avec les paramètres définis par `*params` sur le vecteur de bits `A`. Le paramètre `w` indique le nombre de bits sur lesquels la transformation linéaire s'effectue.

```
void AffPermut ( void *params,
                  int w
               );
```

Fonction qui affiche les paramètres `*params` de la transformation linéaire effectuée sur `w` bits.

timer

Les facilités incluses dans ce module se retrouvent dans le module `chrono` de `mylib` disponible au laboratoire de simulation.

`timer` est un petit module qui fait l'interface avec l'horloge de l'ordinateur, et sert à chronométrer l'exécution de certaines parties d'un programme.

On peut faire fonctionner plusieurs chronomètres simultanément. Pour chaque chronomètre que l'on veut utiliser, on doit déclarer une variable de type `Chrono`. La procédure `InitChrono` permet d'initialiser ou de réinitialiser un chronomètre à zéro. Les types `TimeFormat` et `TimeUnit`, prédéfinis dans le module, permettent de choisir les unités de temps utilisées.

La valeur du temps de CPU consommé par le programme depuis son dernier appel à `InitChrono`, pour un chronomètre donné, peut être retournée à l'usager par la procédure `ValChrono`. Evidemment, la valeur retournée inclut une partie du temps d'exécution des procédures du module `chrono`.

```
typedef struct { unsigned long microsec, second ; } Chrono ;
```

Un chronomètre. Pour chaque chronomètre qu'il veut utiliser, l'usager doir déclarer une variable de ce type, puis l'initialiser en appelant `InitChrono`.

```
enum TimeFormat {sec, min, hours, days, hms} ;
typedef enum TimeFormat TimeFormat ;
```

Lors de chaque consultation de chronomètre, l'usager spécifie s'il désire obtenir une mesure en secondes (sec), en minutes (min), en heures (hours) ou en jours (days). Lorsqu'on fait imprimer la valeur du chronomètre, on peut aussi demander le format `hms`, et le temps apparaîtra sous la forme `HH :MM :SS.xx` indiquant les heures, minutes, secondes et centièmes de secondes.

```
void InitChrono (Chrono * C) ;
```

Initializes the clock `C` to zero (0).

```
double ValChrono (Chrono C, TimeFormat Unit) ;
```

Retourne, selon l'unité `Unit`, le temps CPU consommé par le programme appelant depuis son dernier appel à `InitChrono` pour le chronomètre `C`. Si `hms` est passé en paramètre, un message d'erreur sera retourné.

ANNEXES

B. Code du programme recherche

Code du programme recherche

Makefile

```
### Makefile pour le programme recherche

objects = mecf.o lintrans.o etages.o tuplets.o baseopera.o timer.o
polynomial.o tgfsr.o tausworthe.o temperMK.o selft.o permut.o util.o

C = cc

default : recherche.c  $(objects)
$C recherche.c $(objects) -O3 -lm -o recherche

util.o : util.c
$C -c -O3 -Wall util.c
```

recherche.c

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include "btypes.h"
#include "etages.h"
#include "lintrans.h"
#include "baseopera.h"
#include "mecf.h"
#include "tuplets.h"
#include "temperMK.h"
#include "timer.h"
#define SEPARATION printf("\n\n+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++..+\n")

void Seek(char *, char *);
/* ===================================================================== */
/*   MAIN                                                                */
/* ===================================================================== */
int main (int argc, char *argv[] ){
   FILE *input;

   if( argc < 2 || argc > 3 )
   {
     printf("\nUSAGE :  POL <data_file> [outfile] \n\n");
     exit(1);
   }
   if(!(input = fopen(argv[1],"r")) )
   {
     printf("\n Fichier de donnees %s introuvable.\n\n", argv[1]);
     exit(1);
   }
   fclose(input);

   Seek (argv[1], argv[2]);


   return 0;
}

/* ********************************************************************* */
/* void Seek( char *in, char *out )                                      */
/* Coeur du programme de recherche de paramètres.  *in pointe vers le    */
/* fichier qui contient les données permettant d'orienter la recherche.  */
/* *out n'est pas utilisé(pour l'instant   */
/* ********************************************************************* */
void readdata(Combinaison *C, char *datafile);

void Seek (char * in, char * out)
{
  Combinaison Combin;    /* Contient toute l'information sur les géné-    */
                         /* rateurs à vérifier                            */
  int nbgen,             /* nombre de générateurs vérifiés                */
      nbsel,             /* nombre de générateurs sélectionnés            */
      nbME,              /* nombre de générateurs ME                      */
      nbCF;              /* nombre de générateurs CF                      */

   boolean Continue;     /* indique s'il existe un prochain générateur    */
                         /* valide                                        */
   Chrono timer;         /* Enregistre le temps de recherche              */
   int no_essai;         /* Numero de l'essai en cours                    */
```

```
extern int limitv,affprogress,nbessais, execTestME,temperingMKopt;
extern paramMecf pmecf;
extern paramTuplets ptuplets;
extern boolean tupletsverif;

nbgen = nbsel = nbME = nbCF = 0;
                  /* IMPORTANT- Voir guide, module baseopera   */
Init_REGPOLY();   /* Fonction qui initialise certains vecteurs */
/* Lecture des données dans le fichier de données principal    */
readdata(&Combin,in);

if(!PremierGen(&Combin)){
  printf("Combinaison non-valide");
  exit(1);
}

no_essai=1;
InitChrono(&timer); /* Initialise le chronomètre               */
do {
  MAJToutesTrans(&Combin); /* Mise à jour des paramètres des   */
                           /* transformations linéaires        */
  nbgen++;
  if(temperingMKopt){ /* si on optimise le tempering M-K       */
    TemperMKTestME(&Combin,&pmecf,limitv,affprogress);
  }
  else if(execTestME){
    TestME (&Combin, &pmecf,-1);
  }
  if(tupletsverif){ /* Si on vérfie les projections éloignées */
    tuplets(&Combin,&ptuplets);
  }
  /* Affichage des résultats                                  */
  if (execTestME && estME(&pmecf)) {
    nbME++;
    TestCF (&Combin, &pmecf);
    if (estCF(&pmecf)) {
      nbCF++;
    }
    fflush (stdout);
  }
  if(execTestME && estPresqueME(&pmecf) ){
    if(tupletsverif && estTuplets(&ptuplets)){
      AffCombCourante(&Combin);
      if(!estME(&pmecf)){
        printf("\n  Tableau des écarts en dimension pour chaque résolution");
        AffTable(&pmecf,'l');
      }
      else{
        AffME(&pmecf);
        AffCF(&pmecf);
      }
      AffTuplets(&ptuplets);
      SEPARATION;
      nbsel++;
    }
    if(!tupletsverif){
      AffCombCourante(&Combin);
      if(!estME(&pmecf)){
        printf("\n  Tableau des écarts en dimension pour chaque résolution");
        AffTable(&pmecf,'l');
      }
```

```
        else{
          AffME(&pmecf);
          AffCF(&pmecf);
        }
        SEPARATION;
        nbsel++;
      }

    }
    else if(!execTestME && tupletsverif && estTuplets(&ptuplets)){
      AffCombCourante(&Combin);
      AffTuplets(&ptuplets);
      nbsel++;
      SEPARATION;
    }

    /* Vérifie s'il y a un autre générateur combiné valide à vérifier    */
    if(nbessais==no_essai){
      Continue=ProchainGen(&Combin);
      no_essai=1;
    }else{
      Continue=TRUE;
      no_essai++;
    }
  } while (Continue);

  /* Affichage du résumé de la recherche                       */
  printf("\n===========================\n");
  printf("   total   =  %10u  \n", nbgen);
  printf("\n");
  printf("     ME    =  %10u  \n", nbME);
  printf("   CF-ME   =  %10u  \n", nbCF);
  printf("  retenus  =  %10u  \n", nbsel);
  printf("---------------------------\n");
  printf(" CPU (sec) =   %5.2f       \n",ValChrono(timer,sec) );
  printf("===========================\n");
}
```

util.c

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <limits.h>
#include "btypes.h"
#include "etages.h"
#include "lintrans.h"
#include "baseopera.h"
#include "mecf.h"
#include "tuplets.h"
#include "tausworthe.h"
#include "tgfsr.h"
#include "polynomial.h"
#include "permut.h"
#include "temperMK.h"
#include "selft.h"

int  execTestME=TRUE,        /* TRUE= On verifie l'equidistribution            */
     temperingMKopt=FALSE;  /* TRUE= On optimise le tempering                 */
int nbessais;               /* nombre d'essais de trans.lin par géné. combiné  */
paramMecf pmecf;            /* Paramètres de l'équidistribution à vérifier     */
paramTuplets ptuplets;      /* Paramètres de l'équidistribution des projections*/
int tupletsverif=FALSE;     /* Indique si on vérifie l'équidis. des projections*/
int mDelta;                 /* Valeur maximale de Delta(t_1,..t_d)             */
void ReadTrans(Etage *E, char *filename);
int limitv=INT_MAX;         /* Dans l'optimisation du tempering de Matsumoto-  */
                           /* Kurita, la résolution maximale jusqu'où on      */
                           /* optimise                                        */
int affprogress=FALSE;      /* Indique si on affiche la progression de l'opti- */
                           /* misation du tempering de Matsumoto-Kurita       */
/* ********************************************************************** */
/* void readdata(char *datafile)                                          */
/* Fonction qui lit les données relié à la recherche de générateur        */
/* C'est dans le fichier, spécifié par datafile, que l'on trouvera l'information*/
/* concernant le type de générateur auquel on est intéressé (Tausworthe,TGFSR,  */
/* ou polynomial). Tous les renseignements qui touche les parametres de la     */
/* recherche sont lus ici                                                 */
/* ********************************************************************** */
void readdata(Combinaison *C, char *datafile){

  extern int nbessais;
  extern paramMecf pmecf;
  extern paramTuplets ptuplets;
  int mse,mDD,d;
  int delta[K_MAX+1], s[K_MAX+2];
  int i,j,jmin,jmax,nb_lignes_ecarts,ec,NbComp,L;
  char gen_type[25],genfile[50],transfile[50],old_gen[50];
  int deftrans;
  long seed1,seed2;         /* Racine du generateur de nombres aleatoires pour */
                           /* la recherche de parametres.                     */
  boolean same, Readnbessais=FALSE;
  FILE *f;

  /* Ouverture du fichier de données principal                */
  f=fopen(datafile,"r");
  if(f==NULL){
    printf("Fichier %s inexistant\n",datafile);
    exit(1);
  }
```

```
/* Nombre de composantes au générateurs combinés       */
fscanf(f,"%d",&NbComp);
if( NbComp > J_MAX )
{
  printf("\n Donnees invalides : trop de generateurs!\n\n");
  exit(1);
}
ReadLn(f);

/* Lecture de la racine du générateur de nombres aléatoires */
/* pour les transformations linéaires                       */
fscanf(f,"%ld",&seed1);
if(seed1==-1){
  seed1=((ulong)time(NULL))%2147483562;
  seed2=seed1-1000;
}
else{
  fscanf(f,"%ld",&seed2);
}
SetCombLec88 (seed1,seed2);

ReadLn(f);
/* Allocation de la mémoire pour avoir NbComp ‘Etage‘s dans */
/* la structure C de type Combinaison.                      */
AllouerEtagesDansCombinaison(C,NbComp);
printf("====================================================================\n");
printf("RESUME DE LA RECHERCHE DE GENERATEURS\n\n");
printf("Racine du générateur des trans. lin. = ( %ld, %ld )\n\n",seed1,seed2);
if(NbComp==1)
  printf("%d composante :\n",NbComp);
else
  printf("%d composantes :\n",NbComp);
for(j=0;j<NbComp;j++){
  /* Lecture des données pour la composante j               */
  fscanf(f,"%s %d",gen_type,&L);
                              /* L = resolution du generateur            */
  fscanf(f,"%s",genfile);     /* Nom du fichier contenant les données du */
                              /* generateur choisi                       */
  /* Si genfile est "same", alors ca indique que fichier de */
  /* la composante j est le meme que celle de la composante */
  /* j-1.                                                   */
  if(strncmp(genfile,"same",4)==0){
     same=TRUE;
     strcpy(genfile,old_gen);
  }
  else{
    same=FALSE;
    strcpy(old_gen,genfile);
  }
  ReadLn(f);
  fscanf(f,"%d",&deftrans);
  /* si deftrans==1, alors on fait des trans sur cet etage  */
  if(deftrans==TRUE){
    fscanf(f,"%s",transfile);
    Readnbessais=TRUE;
  }
  ReadLn(f);
```

```
  /* On lit les données pour les générateurs à vérifier     */
  if(strcmp("taus",gen_type)==0){
    LireDonneesTaus(&(C->Etages[j]),genfile,same,L);
    printf("- Composante %d : Tausworthe \n",j+1);
  }
  else if(strcmp("poly",gen_type)==0){
    LireDonneesPoly(&(C->Etages[j]),genfile,same,L);
    printf("- Composante %d : Polynomial \n",j+1);
  }
  else if(strcmp("tgfsr",gen_type)==0){
    LireDonneesTGFSR(&(C->Etages[j]),genfile,same,L);
    printf("- Composante %d : TGFSR \n",j+1);

  }
  else{
    printf("Générateur non-supporte= %s\n",gen_type);
    exit(1);
  }
  if(deftrans==1)
    ReadTrans(&(C->Etages[j]),transfile);
}
nbessais=1;
fscanf(f,"%d",&nbessais);
ReadLn(f);
if(Readnbessais){
  printf("Nombre d'essais par générateur combiné : %d\n",nbessais);
}

fscanf(f,"%d",&mse); /* Valeur maximal de la somme des ecart[] pour laquelle */
                      /* on affiche le générateur à J composantes courant */
if(mse==-1){
  execTestME=FALSE;
  mse=INT_MAX;         /* Dans ce cas, le maximum sur la sommes des Delta_l    */
                      /* pour l dans Psi_12 est à la plus grande valeur      */
                      /* possible                                            */
}
if(execTestME){
  printf("Borne sup. pour la somme des écarts dans psi_12 : %d\n",mse);
}
ReadLn(f);
fscanf(f,"%d",&nb_lignes_ecarts);
                      /* Nombre de lignes qui suivent qui permettront de     */
                      /* déterminer les écarts admissible pour une           */
                      /* résolution donnée.                                  */
ReadLn(f);
if(nb_lignes_ecarts>0){
  printf("delta pour bits particuliers :\n");
  for(i=0;i<nb_lignes_ecarts;i++){
    fscanf(f,"%d %d %d",&jmin,&jmax,&ec);
    printf("   >>Bits %d à %d delta_i=%d\n",jmin,jmax,ec);
    ReadLn(f);
    for(j=jmin;j<=jmax;j++)
      delta[j]=ec;
  }
}
else{
  for(i=0;i<=MAXL;i++){
    delta[i]=10000000;
  }
}
```

```
  InitParamMecf(&pmecf,delta,mse,0);
  mDD=0;
  fscanf(f,"%d",&tupletsverif); /* tupletsverif == 1 , alors on calcule la      */
                                /* valeur de Delta(t_1,..,t_d) (voir tuplets.c) */
  if(tupletsverif==1){
    fscanf(f,"%d",&d);         /* valeur de d dans Delta(t_1,..,t_d)           */
    for(j=1;j<=d;j++)
      fscanf(f,"%d",&s[j]);     /* valeur de t[j] dans Delta(t_1,..,t_d)        */
    fscanf(f,"%d",&mDD); /* valeur maximale de DD pour qu'on affiche le  */
                                /* générateur à J composantes courant          */
    printf("On vérifie le Critère DELTA( ");
    for(j=1;j<d;j++)
      printf("%d, ",s[j]);
    printf("%d)\n",s[d]);
    printf("On conserve quand DELTA(t_1,..,t_s)<=%d\n",mDD);
    InitParamTuplets(&ptuplets,d,s,mDD);
  }
  else{
    for(j=0;j<=K_MAX;j++)
      s[j]=0;
    InitParamTuplets(&ptuplets,0,s,0);
  }
  ReadLn(f);
  printf("===================================================================\n");
  fclose(f);
}


/* ************************************************************************ */
/* void ReadTrans(Etage *E, char *filename)                                 */
/* Fonction qui lit du fichier filename les transformations linéaires qui seront*/
/* appliquées à la sortie de la composante courante de l'Etage *E.          */
/* ************************************************************************ */
void ReadTrans(Etage *E, char *filename){

  int i,k,param1,param2;
  FILE *f;
  int NbTrans,aleatoire,M;
  char lintrans[25];
  BitVect init_b,init_c;
  boolean Temp_opt,D;

  /* Ouverture du fichier de transformation linéaire */
  f=fopen(filename,"r");
  if(f==NULL){
    printf("fichier %s inexistant\n",filename);
    exit(1);
  }
  fscanf(f,"%d",&NbTrans);
  /* Allocation de la mémoire pour NbTrans transformations linéaires à         */
  /* l'étage *E.                                                               */
  AllouerTransDansEtage(E,NbTrans);
  ReadLn(f);
  if(NbTrans>0){
    printf("  Transformations :\n");
```

```
    for(i=0;i<NbTrans;i++){
      /* lecture de la i-ieme transformation linéaire                          */
      fscanf(f,"%s %d",lintrans,&param1);
      if(strcmp("permut",lintrans)==0){
        parampermut permut;
        printf ("   * Permutation\n");
        fscanf(f,"%d",&param2);
        ReadLn(f);
        InitPermut(&permut,param1,param2);
        /* Ajout d'une permutation à l'Etage *E                                */
        AjoutPermut(E,&permut);
      }
      else if(strcmp("selft",lintrans)==0){
        paramselft selft;
        printf("   * Self-Tempering\n");
        ReadLn(f);
        InitSelfT(&selft,param1);
        /* Ajout d'un self-tempering à l'Etage *E                              */
        AjoutSelfT(E,&selft);
      }
      else if(strcmp("tempMK",lintrans)==0 || strcmp("tempMKopt",lintrans)==0){
        paramTempMK temper;
        printf("   * Tempering Matsumoto-Kurita\n");
        fscanf(f,"%d",&param2);
        if(param1==-1 || param2==-1){
          printf("On doit absolument definir s et t pour le tempering M-K\n");
          exit(1);
        }
        fscanf(f,"%d",&aleatoire);
        Temp_opt=FALSE;
        if(strcmp("tempMKopt",lintrans)==0){
          temperingMKopt=TRUE;
          Temp_opt=TRUE;
          fscanf(f,"%d %d",&D,&M);
          affprogress|=D;
          if(M<limitv){
            limitv=M;
          }
        }
        ReadLn(f);
        if(aleatoire!=-1){
          for(k=0;k<aleatoire;k++)
            fscanf(f,"%lx",&init_b[k]);
          ReadLn(f);
          for(k=0;k<aleatoire;k++)
            fscanf(f,"%lx",&init_c[k]);
          ReadLn(f);
        }
        else{
          MetBVaZero(init_b);
          MetBVaZero(init_c);
        }
        InitTemperMK(&temper,param1,param2,aleatoire,init_b,init_c,Temp_opt);
        /* Ajout d'un tempering de Matsumoto-Kurita à l'Etage *E               */
        AjoutTemperMK(E,&temper);
      }
      else {
        printf("Transformation inconnue : %s ",lintrans);
        exit(1);
      }
    }
  }
  fclose(f);
}
```

Références

[1] R. Couture, P. L'Ecuyer, and C. Lemieux. Polynomial lattice rules. in preparation, 1999.

[2] P. L'Ecuyer. Maximally equidistributed combined Tausworthe generators. *Mathematics of Computation*, 65(213) :203–213, 1996.

[3] P. L'Ecuyer. Tables of maximally equidistributed combined LFSR generators. *Mathematics of Computation*, 68(225) :261–269, 1999.

[4] P. L'Ecuyer and F. Panneton. A new class of linear feedback shift register generators. In J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, editors, *Proceedings of the 2000 Winter Simulation Conference*, pages 690–696, Pistacaway, NJ, 2000. IEEE Press.

[5] M. Matsumoto and Y. Kurita. Twisted GFSR generators. *ACM Transactions on Modeling and Computer Simulation*, 2(3) :179–194, 1992.

[6] M. Matsumoto and Y. Kurita. Twisted GFSR generators II. *ACM Transactions on Modeling and Computer Simulation*, 4(3) :254–266, 1994.

[7] M. Matsumoto and T. Nishimura. Mersenne twister : A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1) :3–30, 1998.

[8] F. Panneton. Générateurs de nombres aléatoires utilisant des récurrences linéaires modulo 2. Master's thesis, Département d'informatique et de recherche opérationnelle, Université de Montréal, 2000.

[9] R. C. Tausworthe. Random numbers generated by linear recurrence modulo two. *Mathematics of Computation*, 19 :201–209, 1965.