Rapport TP : Générateurs Aléatoires

Bayard Guillaume, Pierucci Dimitri

Classe Dvector

On veut étudier les différences entre les deux écritures suivantes :
Dvector x;
x=Dvector(3,1.0);

et

Dvector x=Dvector(3,1.0);

Dans la première écriture, x est construit par défaut (taille nulle). La commande qui suit détruit le vecteur x et le reconstruit avec le constructeur Dvector(int, double).
La première écriture fait donc appel deux fois à un constructeur.

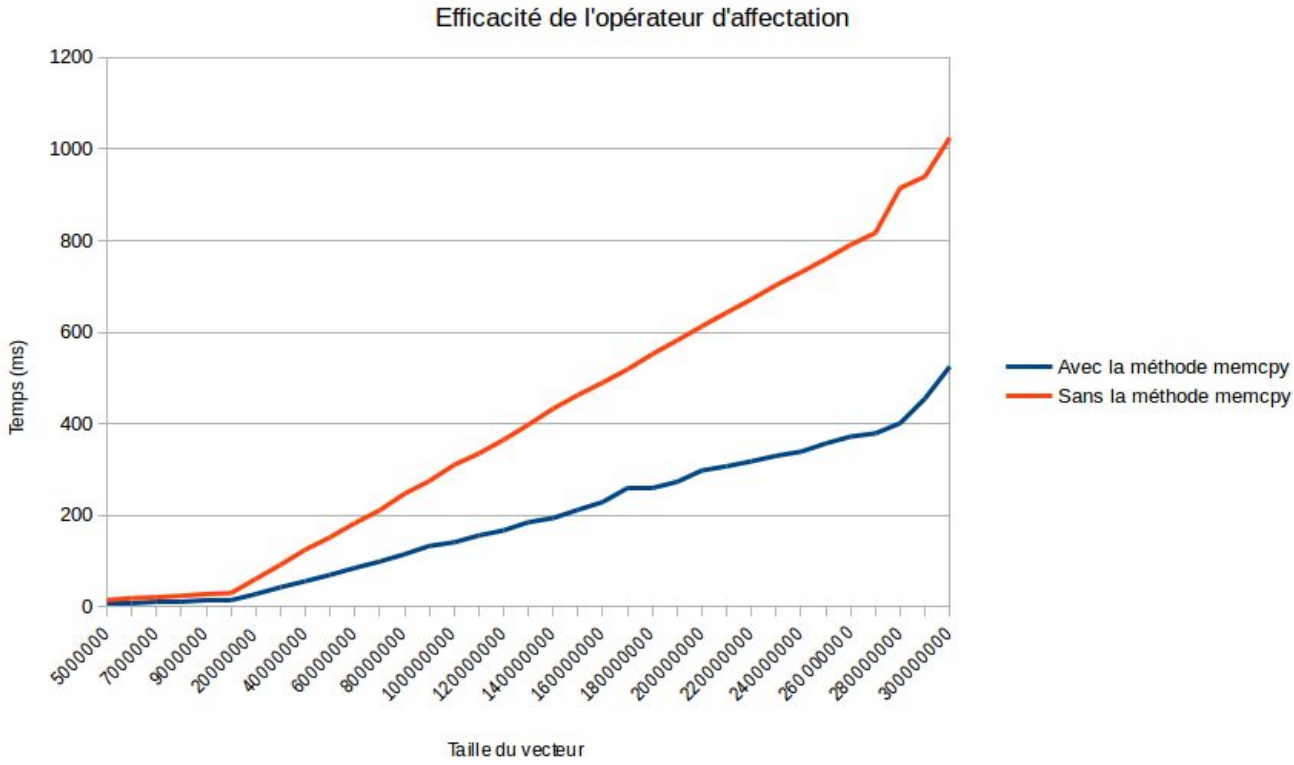
Dans la deuxième écriture, le vecteur x est directement créé par le constructeur Dvector(int, double).

Pour un même résultat, la deuxième écriture ne construit qu'une seule fois le vecteur x. Cette écriture est donc à privilégier.

Deux implémentations sont possibles pour surcharger l'opérateur d'affectation = :

- on peut utiliser la commande *memcpy* qui copie la zone mémoire dans laquelle est stockée le vecteur entrée en paramètre et la copie dans le vecteur qu'on veut modifier.
- on peut se passer de la commande *memcpy* et copier manuellement toutes les valeurs du vecteurs.

On effectue donc des tests de performance sur ces deux méthodes :



On voit très clairement que pour des vecteurs de grande taille (plusieurs millions d'éléments), l'opérateur d'affectation faisant appel à la méthode *memcpy* est bien plus rapide.

Générateurs Aléatoires :

Comme demandé dans le sujet nous avons implémenté deux générateurs aléatoires d'entiers : le Park-Miller et le XorShift. Ces deux classes sont héritées de la classe abstraite GénérateurNombreAléatoire.

Avantages et inconvénients des deux générateurs :

Park Miller

- Avantages : Simplicité d'implémentation
- Inconvénients : La taille des entiers est majorée par m (on ne peut tirer d'entier supérieur au paramètre m)

XorShift

- Avantageca : Simplicité d'implémentation, permet de générer des nombres a priori sans contrainte de taille, opérations bit à bit peu couteuses.

- Inconvénients : Nécessite de choisir un bon triplet d'entier (propriétés statistiques …)

Distributions :

Nous avons implémenté deux types de distributions : une loi uniforme de paramètres a et b (loi uniforme sur l'intervalle [a,b]) et une loi normale de paramètres (m,sigma).

Le choix de l'algorithme pour la générer des tirages de loi uniformes étant simple, nous détaillerons seulement celui utilisé pour la distribution normale.

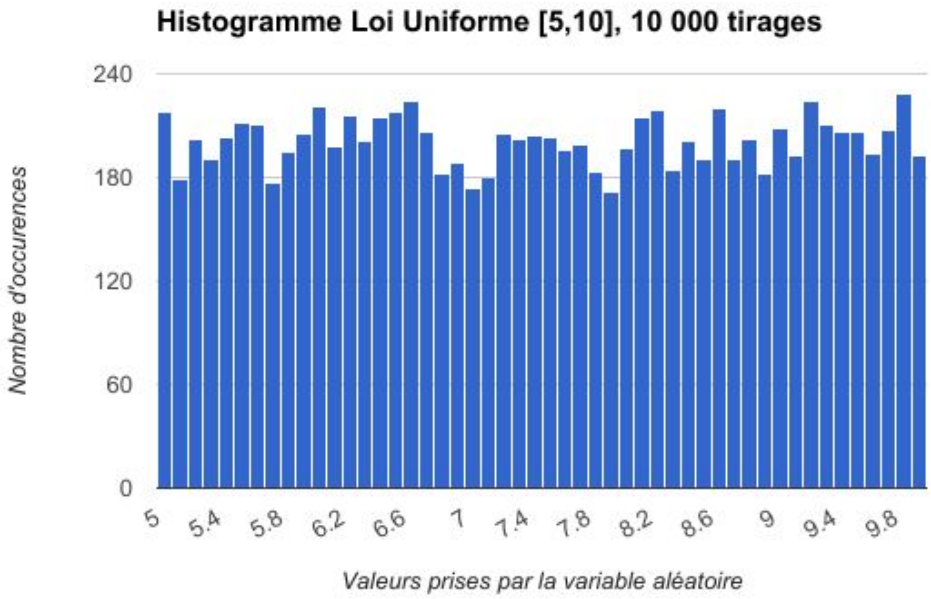
Nous avons fait le choix de l'algorithme de Box-Muller :

$$X_1 = \sqrt{-2\log U_1}\cos(2\pi U_2),$$

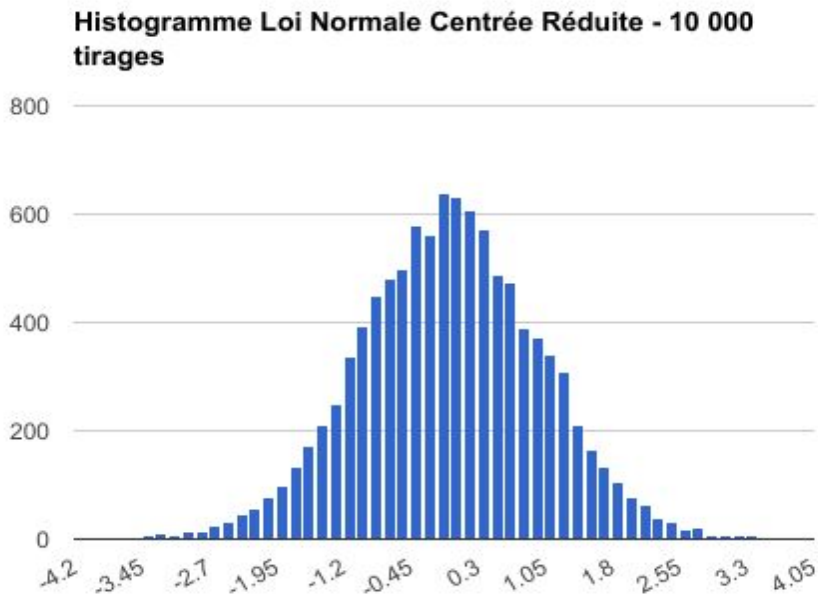
Avec U1 et U2 deux lois uniformes indépendantes sur [0,1]
X1 suit alors une loi normale centrée réduite.
On utilise dans cet algorithme le générateur d'entiers XorShift, à cause de sa plus grande rapidité.

Pour visualiser les lois que nous avons simulé, nous avons tracé des histogrammes pour une loi uniforme et une loi normale.

Loi Uniforme [5,10] - 10 000 tirages



Loi Normale Centrée Réduite - 10 000 tirages



Tests de performances :

Park-Miller - Temps pour générer 100 000 000 entiers : 1,357s
XorShift - Temps pour générer 100 000 000 entiers : 0.988s
Distribution Normale (0,1) - Temps pour générer 100 000 000 tirages : 13,251s
Loi Uniforme (0,1) - Temps pour générer 100 000 000 tirages : 2,460s

*Mesures effectuée avec la commande **time** dans le terminal*
Machine utilisée : Intel i5, 8Go de RAM.