HEIG–VD
Simulation et Optimisation

Bastien Clément
14 avril 2016

Rapport : Génération de variables aléatoires **Travail pratique 1**

Table des matières

1 Introduction

L'objectif de ce premier laboratoire est d'implémenter différents algorithmes de génération de variables aléatoires utilisant des fonctions affine-par-morceaux comme fonctions de densité. Ce TP reprend de nombreux points du dernier Travail Personnel et les algorithmes à implémenter sont entièrement basés sur les résultats obtenus lors du travail précédent.

Le langage et les choix d'implémentation sont laissés libres. Dans mon cas, j'ai opté pour une approche mixe orientée-objet/fonctionnelle avec le langage Scala. De cette façon, j'ai pu profiter du concept d'objets pour définir les éléments de base du programme (Fonctions, Tranches, Générateurs) et utiliser une approche fonctionnelle pour exprimer les opérations sur les résultats sous forme de *pipeline* de transformations à appliquer.

Les différents algorithmes ont ensuite été testés sur l'ensemble des fonctions fournies. Dans le cas des données présentées dans ce rapport, chaque test est composés de 10'000 mesures, elles mêmes composées de 1'000'000 générations de réalisation de la variable aléatoire.

Les tests ont été effectués sur une machine équipée d'un processeur *Intel Core i5-1234k* à 3.57 GHz avec 16 GB de RAM et exécutant Windows 10 et Java 8.0.

Ces tests étant relativement longs, j'ai choisi de tester chaque fois les 3 algorithmes en parallèle. Ceux-ci ne partageant aucune information, la concurrence n'était donc pas un problème. De plus, j'ai pris soin de définir la priorité maximale disponible pour le processus de test ainsi que de vérifier qu'aucun autre processus n'était susceptible d'utiliser le processeur le temps des tests. Ce dernier n'étant utilisé qu'à 75% (4 coeurs / 3 threads), je ne m'attends pas à des répercussions sur les mesures de temps.

2 Algorithmes implémentés

2.1 Acceptation-rejet (`HitMissGenerator`)

Un algorithme basé sur l'approche *bête et méchante* développée au point 2.*a* du TP1.

Un point (x, y) est généré uniformément dans le rectangle $[a, b] \times [0, y_{max}]$. S'il se trouve sous la courbe de f, x est retourné, sinon un nouveau point est généré jusqu'à ce que la condition soit vérifiée.

Chaque coup possède une chance $r = \frac{\int_a^b f(x)}{(b-a)y_{max}}$ égale au ratio de l'aire de la fonction par rapport à l'aire du rectangle encadrant de tomber sous la courbe de la fonction. Nous pouvons donc espérer, en moyenne, obtenir un résultat après $1/r$ essais.

2.2 Géométrique (`GeometricGenerator`)

Une approche basées sur la méthode des mélanges et une base d'acceptation-rejet. Dans le cas d'un rejet, les propriétés géométriques du problème peuvent être utilisées pour obtenir en temps constant une solution alternative valide. L'idée est développée dans le point 2.*b* du TP1.

Dans un premier temps, une tranche de la fonction est choisie aléatoirement avec une probabilité égale au rapport entre l'aire de la tranche et l'aire de la fonction dans son ensemble.

Par la suite, un point (x, y) est généré uniformément dans le rectangle $[a, b] \times [0, y_0 + y_1]$. Si le point tombe sous la courbe de la fonction, x est retourné, dans le cas contraire, une symétrie est appliquée pour obtenir un point sous la courbe.

Ici, nous avons une complexité en $O(1 + a)$, avec a la complexité de choisir une tranche aléatoirement. Dans le cadre du cours, nous avons vu un algorithme en $O(n)$ avec n le nombre de tranches.

Une méthode alternative (la *méthode de l'alias*), permettant de produire un choix en temps constant a été mentionnée à de nombreuses occasions. En recherchant d'avantage d'informations, j'ai pu trouver une très bonne explication accompagnée d'une implémentation Java de l'algorithme que j'ai pu réutiliser. Plus de détails sont disponibles dans le fichier `DiscreteGenerator.scala`.

Ce générateur produit donc des réalisations en temps $O(1)$.

2.3 Fonction inverse (`InverseGenerator`)

3 Compilation et exécution

4 Validation des implémentations

Dans un premier temps, nous nous intéresserons l'espérance d'une variable aléatoire générée en utilisant nos différentes fonctions comme fonctions de densité.

Pour chaque fonction, nous calculerons la densité attendue de façon purement mathématique. Nous effectuerons ensuite

5 Temps de génération

5.1 Résultats

Fonction	Algorithme	Moyenne	Écart-type	$IC_{95\%}$	Δ
Uniforme	Acceptation-rejet	53.552590	7.041156	[53.414584 ; 53.690597]	0.276013
	Géométrique	92.319710	8.157461	[92.159824 ; 92.479596]	0.319772
	Fonction inverse	61.869008	7.200549	[61.727877 ; 62.010139]	0.282262
Triangulaire	Acceptation-rejet	137.81593	7.751517	[137.66405 ; 137.96864]	0.303859
	Géométrique	96.373801	7.497934	[96.226841 ; 96.520760]	0.293919
	Fonction inverse	73.564317	7.159326	[73.423994 ; 73.704640]	0.280646
Plat	Acceptation-rejet	65.621054	7.580608	[65.472475 ; 65.769634]	0.297160
	Géométrique	94.995949	7.540267	[94.848160 ; 95.143738]	0.295578
	Fonction inverse	71.114322	7.598602	[70.965389 ; 71.263254]	0.297865
Accidenté	Acceptation-rejet	208.38447	8.445177	[208.21894 ; 208.54999]	0.331051
	Géométrique	95.905446	6.879117	[95.770616 ; 96.040277]	0.269661
	Fonction inverse	74.447486	6.724518	[74.315685 ; 74.579286]	0.263601