HEIG–VD
Simulation et Optimisation

Bastien Clément
13 mai 2016

Rapport : Méthodes de Monte-Carlo et réduction de variance **TP 2**

Table des matières

1 Introduction

L'objectif de ce second laboratoire et d'estimer l'intégrale définie d'une fonction mathématique (trop complexe pour appliquer les méthodes usuelles d'intégration) en utilisant différentes méthodes de Monte-Carlo et de comparer leur performance.

Plus spécifiquement, nous nous intéressons à la fonction

$$g(x) = \left(25 + \frac{x(x-6)(x-8)(x-14)}{25}\right) e^{\sqrt{1+cos(x^2/10)}}$$

et à l'intégrale sur l'intervalle $[0, 15]$

$$\int_0^{15} g(x)\,dx$$

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 mixte 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 composé de 10'000 mesures, elles-mêmes composées de 1'000'000 de réalisations. L'exécution des tests utilisés dans ce rapport nécessite un peu plus d'une heure et demie. L'archive ZIP fournie contient en plus du fichier `output.txt` un fichier `output.fast.txt` proposant des résultats bien plus rapides à générer dans le cas où l'exécution devrait être réitérée.

Les tests ont été effectués sur une machine équipée d'un processeur *Intel Core i5-3570* à 3.57 GHz avec 16 GB de RAM et exécutant Windows 10 et Java 8u77.

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 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 au cours des tests. Ce dernier n'étant utilisé qu'à 75% (4 coeurs / 3 threads), je ne m'attends pas à ce que le parallélisme implique des répercussions significatives sur les mesures de temps individuelles.

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 trou-

ver 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`)

Cette dernière approche réutilise le concept de sélection de tranche de l'algorithme précédent mais utilise la méthode développée au point 1.*e* du TP1 : une fois la tranche choisie, l'inverse de la fonction de répartition associée à la variable aléatoire définie par la tranche est utilisée pour produire une réalisation en temps constant.

Comme expliqué dans le TP1, un cas particulier se présente lorsque la section de la fonction correspond à une tranche uniforme. Dans ce cas, l'algorithme équivaut à la génération d'une uniforme entre les deux bornes de la tranche.

3 Compilation et exécution

Le choix d'un langage un peu particulier impose une section sur la méthode de compilation et d'exécution du logiciel. La compilation d'un programme Scala produit du bytecode Java exécutable sur une JVM standard. Une archive JAR exécutable `labo1.jar` est disponible à la racine de l'archive ZIP et propose d'exécuter une version précompilée du logiciel sans autre dépendances que Java.

```
? java -jar labo1.jar <nb iter> <nb gen/iter> <1|0 exécution parallèle> <graine>
$ java -jar labo1.jar 1000 100000 1 42
```

FIGURE 1 – Exemple de lancement avec `labo1.jar`

Les paramètres donnés en exemple correspondent aux valeurs par défaut utilisées si le paramètre correspondant est omis. Utiliser l'exécution parallèle divise en théorie le temps d'exécution total par 3.

Dans le cas où des modifications devaient être apportées, il est nécessaire de recompiler le logiciel. L'utilisation de sbt, le compilateur de Scala est très simple puisqu'il se chargera de télécharger et mettre en place toutes les dépendances nécessaires pour compiler le code. Une version automatique de sbt (`sbt.jar`) est disponible dans l'archive ZIP et peut être lancée directement.

Lorsque sbt est lancé, celui-ci propose un shell interactif permettant de donner des commandes au compilateur. Il permet également d'exécuter directement une application avec la commande `run`. Dans le cas où les sources ont été modifiées depuis la dernière compilation, *sbt* recompilera automatiquement les fichiers nécessaires avant de lancer le programme.

4 Validation des implémentations

Afin de valider l'implémentations des algorithme, les moyennes des valeurs générées par chaque générateur seront comparées à l'espérance calculée des fonctions de test. Si tout fonctionne correctement, les valeurs obtenues expérimentalement devraient être très proches de la valeur calculée.

Le tableau de la figure 3 présente les résultats obtenus dans cette première phase.

```
$ java -jar sbt.jar
[info] Loading project definition from C:/Users/.../project
[info] Set current project to Labo1 (in build file:/C:/Users/.../Labo1/)
> run 1000 100000 1 42
[info] Compiling 12 Scala sources to C:/Users/.../Labo1/target/classes...
[info] 'compiler-interface' not yet compiled for Scala 2.11.8. Compiling...
[info] Compilation completed in 10.482 s
[info] Running Labo1Tests 1000 100000 1 42
Warming up...
...
```

FIGURE 2 – Exemple de lancement avec `sbt.jar`

Fonction	**Algorithme**	**Estimation**	**Écart-type**	$\mathbf{IC}_{95\%}$	$\mathbb{E}[x] \in \mathbf{IC}$
	Acceptation-rejet	10.00004	0.002873	[9.999993 ; 10.00010]	✓
Uniforme	Géométrique	9.999937	0.002890	[9.999880 ; 9.999993]	✗
$\mathbb{E}[x] = 10.0$	Fonction inverse	10.00003	0.002902	[9.999981 ; 10.00009]	✓
	Acceptation-rejet	8.500078	0.004569	[8.499989 ; 8.500168]	✓
Triangulaire	Géométrique	8.499996	0.004619	[8.499906 ; 8.500087]	✓
$\mathbb{E}[x] = 8.5$	Fonction inverse	8.499984	0.004575	[8.499895 ; 8.500074]	✓
	Acceptation-rejet	10.75805	0.005199	[10.75795 ; 10.75815]	✓
Plat	Géométrique	10.75794	0.005171	[10.75784 ; 10.75804]	✓
$\mathbb{E}[x] = 10.757$	Fonction inverse	10.75802	0.005188	[10.75792 ; 10.75812]	✓
	Acceptation-rejet	10.26174	0.005763	[10.26163 ; 10.26185]	✓
Accidenté	Géométrique	10.26171	0.005678	[10.26159 ; 10.26182]	✓
$\mathbb{E}[x] = 10.261$	Fonction inverse	10.26174	0.005770	[10.26162 ; 10.26185]	✓

FIGURE 3 – Résultats obtenus pour l'estimation de l'espérance

4.1 Analyse

Les résultats sont dans l'ensemble conformes aux attentes. Les estimations sont systématiquement très proches de l'espérance calculée ce qui est une preuve du bon fonctionnement des générateurs. Un cas particulier se présente avec le générateur géométrique pour la fonction uniforme : l'intervalle de confiance calculé ne contient pas la valeur attendue. Rien d'inquiétant ici puisque le concept même d'intervalle de confiance implique un risque d'erreur (ici de 5%). Malgré tout, la valeur estimée reste extrêmement proche de la valeur attendue et l'intervalle manque de contenir cette dernière de très peu. De plus, l'utilisation d'une autre graine conduit à un intervalle de confiance pour la même fonction qui contient cette fois-ci la valeur calculée.

Nous pouvons donc en conclure que les générateurs fonctionnent de façon conforme aux attentes.

5 Temps de génération

Dans un second temps, nous nous intéresserons à mesurer le temps nécessaire à chaque générateur pour produire une série de réalisations (ici 1'000'000).

Le tableau de la figure 4 présente les résultats obtenus. Pour chaque fonction, le ratio r de l'aire sous la courbe de la fonction par rapport à l'aire du rectangle encadrant est indiqué.

La figure 5 propose une représentation graphique de ces temps de calculs. Les intervalles de confiances y sont indiqués mais leur étroitesse ne les rends pas très intéressants.

Fonction	**Algorithme**	**Moyenne**	**Écart-type**	**$IC_{95\%}$**	**Δ IC**
	Acceptation-rejet	53.552590	7.041156	[53.414584 ; 53.690597]	0.276013
Uniforme	Géométrique	92.319710	8.157461	[92.159824 ; 92.479596]	0.319772
$r = 1.0$	Fonction inverse	61.869008	7.200549	[61.727877 ; 62.010139]	0.282262
	Acceptation-rejet	137.81593	7.751517	[137.66405 ; 137.96864]	0.303859
Triangulaire	Géométrique	96.373801	7.497934	[96.226841 ; 96.520760]	0.293919
$r = 0.384$	Fonction inverse	73.564317	7.159326	[73.423994 ; 73.704640]	0.280646
	Acceptation-rejet	65.621054	7.580608	[65.472475 ; 65.769634]	0.297160
Plat	Géométrique	94.995949	7.540267	[94.848160 ; 95.143738]	0.295578
$r = 0.872$	Fonction inverse	71.114322	7.598602	[70.965389 ; 71.263254]	0.297865
	Acceptation-rejet	208.38447	8.445177	[208.21894 ; 208.54999]	0.331051
Accidenté	Géométrique	95.905446	6.879117	[95.770616 ; 96.040277]	0.269661
$r = 0.258$	Fonction inverse	74.447486	6.724518	[74.315685 ; 74.579286]	0.263601

FIGURE 4 – Résultats obtenus pour la mesure de temps

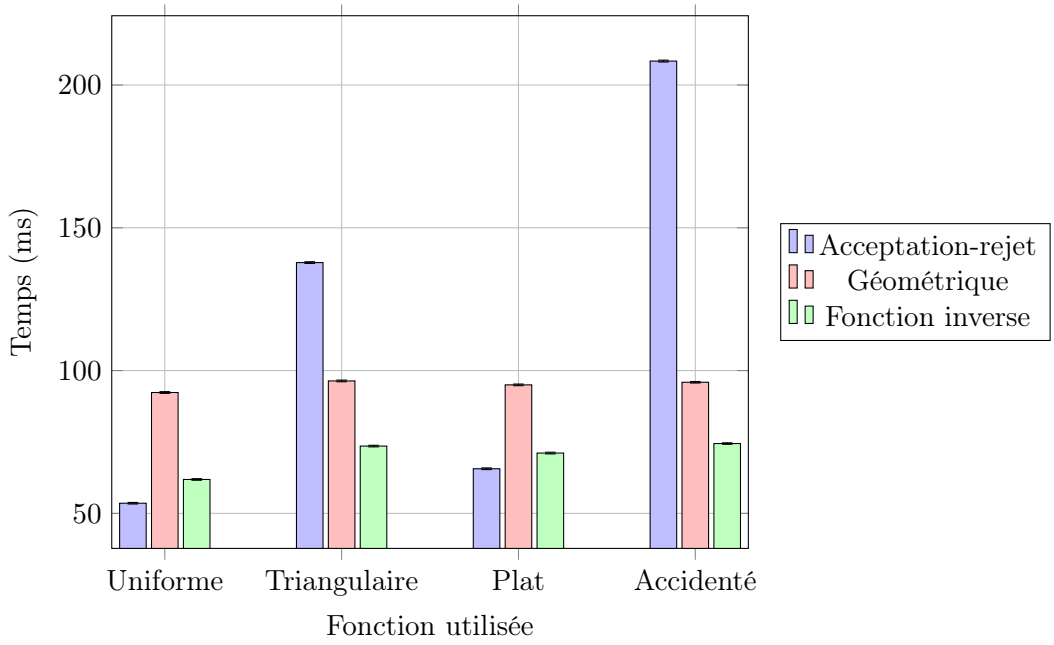

FIGURE 5 – Temps de génération pour 1'000'000 valeurs

5.1 Analyse

Dans un premier temps, il est important de mentionner que la JVM est un environement doté d'un compilateur JIT qui choisira arbitrairement, lors de l'exécution, quels morceaux de code semblent intéressant à compiler en langage machine pour améliorer les performances d'exécution. Les résultats obtenus au lancement de l'application risque ainsi d'être considérablement différent des résultats

obtenus après compilation JIT.

Dans le but de réduire ce phénomène, un *warmup* est effectué au lancement de l'application en générant 1'000'000 valeurs avec 100 fonctions générées aléatoirement sur les 3 générateurs. Le compilateur JIT étant non-déterministe, ceci ne garanti pas son exécution ni même la compilation du code, mais des tests empiriques montrent que cela est suffisant.

Un premier élément intéressant dans ces résultats est la corrélation entre le ratio d'aire et les performances du générateur par acceptation-rejet. Comme expliqué dans l'introduction des algorithmes, ce générateur nécessite potentiellement plusieurs itérations avant de fournir un résultat acceptable. La probabilité de réussir chaque essai est directement lié à la part d'aire du rectangle recouvert par la fonction. Les résultats confirment ce comportement en vérifiant plus ou moins la relation attendue :

$$t_f \approx \frac{t_{uniforme}}{r_f}$$

Les deux autres générateurs semblent quant à eux indépendants du ratio d'aire et relativement constants en terme de performances. En effet, ces algorithmes sont tous deux en complexité temporelle $O(1)$, indépendants donc à la fois de l'aire recouverte et du nombre de tranches.

Une particularité peut tout de même être observée avec la fonction uniforme pour laquelle le générateur par fonction inverse semble légèrement plus rapide que la normale. Tout autres choses étant égales, la différence provient probablement du fait que la fonction uniforme vérifie systématiquement la condition $y_0 = y_1$. La génération se fait alors selon la formule

$$f(y) = x_0 + y * (x_1 - x_0)$$

au lieu de la variante plus complexe lorsque $y_0 \neq y_1$

$$f(y) = x_0 + \frac{\sqrt{y * (y_1^2 - y_0^2) + y_0^2} - y_0}{m}$$

considérablement plus couteuse à cause de l'évaluation de la racine.

L'explication de la différence régulière entre le générateur géométrique et le générateur par fonctions inverses est, lui, plus délicat à expliquer sans analyse plus approfondie. Plusieurs facteurs peuvent y contribuer :

1. Le générateur géométrique nécessite la génération d'une uniforme supplémentaire par rapport au générateur par fonctions inverses.
2. Le générateur géométrique utilise la méthode `slice.evaluate` et effectue donc un appel de fonction alors que le calcul de l'inverse est réalisé directement dans le corps du générateur.

6 Conclusion

Dans l'ensemble les générateurs réalisés au cours de ce travail pratique fonctionnent conformément aux attentes. Les résultats sont cohérents avec la logique de leur implémentation et la régularité des deux générateurs $O(1)$ est une propriété intéressante pour faciliter la prévision de temps d'exécution d'une simulation dont le nombre de valeurs à générer est connu.

Avec un temps moyen d'environ 70 ns par réalisation, le générateur par fonctions inverses semble offrir des performances qui conviendront sans doute à la réalisation des prochains travaux pratiques.