Rapport du projet de Simulation :

Schérer Robin Tondeur Pierre

Année académique : 2017-2018
Simulation
Université de Mons

28 mai 2018

Table des matières

1 Introduction

Ce projet à été réalisé dans le cadre du cours dispensé par Alain Buys, "Simulation sur ordinateur".

Le but de ce projet est de premièrement étudier le caractère pseudo-aléatoire des décimales de π (1.000.000 de décimales ont été étudiées) via des tests vue au cours. Nous devions ensuite utilisé ces décimales pour implémenter un générateur de loi uniforme $[0, 1[$ et ensuite le comparer avec le générateur par défaut de Python via les tests implémentés

2 Analyse des décimales de π

Tentons de déterminer si les décimales de π suivent bien une loi uniforme via les tests suivant.

2.1 Test du χ^2

Ce test statistique permet de tester l'adéquation d'une série de données à une famille de lois de probabilités. Pour ce test nous avons besoin d'un certain nombre d'intervalle, et nous comptons le nombre de valeurs qu'on a générées dans chaque intervalles, ceci est notre hypothèse. Nous avons aussi besoin une hypothèse nulle notée H_0 qui considèrent que les données suivent une lois de probabilité donnée et nous avons notre hypothèse qui sont nos données obtenu expérimentalement. Nous comparons ensuite ces 2 échantillons de données via la formule suivante :

$$K_n = \sum_{i=1}^{r} (\frac{n_i - Np_i}{\sqrt{Np_i}})^2$$

on a que r est le nombre d'intervalle, Np_i est l'effectif théorique pour la classe i, et n_i est l'effectif qu'on à obtenu expérimentalement dans la classe i. Et nous avons un degrés de libertés égale à r - 1. Puis on se donne une probabilité α qui est le risque d'erreur, avec α et le degré de liberté, on obtient une valeur critique, C_i. Et si $K_n < C_i$, alors on rejette notre hypothèse.

Pour les décimales de π on choisit 10 intervalles chaque intervalle contient 1 chiffre. On a que pour H_0, les décimales de π suivent une loi uniforme, donc la probabilité que chacun des chiffres apparaissent le même nombre de fois.

Voici nos résultats pour notre hypothèse :

	Résulats		
Chiffre	Théorique	Expérimentaux	Erreur relative(%)
0	100 000	99 959	-0.041
1	100 000	99 758	-0.242
2	100 000	100 026	0.026
3	100 000	100 229	0.229
4	100 000	100 230	0.23
5	100 000	100 359	0.359
6	100 000	99 548	-0.452
7	100 000	99 800	-0.2
8	100 000	99 985	-0.015
9	100 000	100 106	0.106

Maintenant effectuons le test du χ^2 sur nos 2 hypothèse :

α	K_n	C_i	On garde notre hypothèse
0,01	5.51	27.88	Oui
0.025	5.51	19.02	Oui
0.05	5.51	16.92	Oui

On a donc que les décimales de π passe bien le test du χ^2 pour nos alpha et que donc elles suivent bien une lois uniformes.

2.2 Test du Gap

Pour ce test on a une suite de nombre $u_1, u_2, ..., u_n$. On choisit $1 \leq a \leq b \leq 9 et on marque ceux qui tombent dans [a, b], qui pour chaq$ - a$\frac{}{10}$ de se produire. Et on s'intéresse ensuite à la distance entre 2 nombres d'affiler marqués. On se donne n nombre de classe, dans la classe i, on a le nombre de fois qu'on a un trou de distance i entre 2 nombre marqués après analyse de notre séquence de nombre. Ceci est notre hypothèse.

L'hypothèse nulle est que pour chaque classe i on a : $p(1-p)^i$

On compare ensuite ces 2 hypothèses avec le test du χ^2 pour savoir si le test est réussi ou non.

Dans notre cas, on choisit de faire 10 fois ce test pour chacun des 10 chiffre, et que a chaque fois a = b. On a donc à chaque fois que p = 0.1.

Voici ce qu'on obtient pour le nombre d'occurrence de certaines longueurs de trou pour le chiffre 5.

Longueurs du trou	Nombre d'occurrences théorique	Nombre d'occurrences obtenu	Erreur relative (%)
1	10 000	10 232	2.32
2	9000	8957	-0.47
3	8100	8091	-0.11
4	7290	7300	0.14
5	6561	6586	0.38
6	5904	5943	0.66
...	...	...	...
21	1216	1215	-0.08
22	1094	1071	-2.1
23	985	965	-2
24	886	880	-0.68
25	798	769	-3.5
...	...	...	...
53	42	43	2.4
54	37.6	38	1.3
55	33.8	33	2.4

Pour le test de χ^2 on a les résultats suivant :

α	K_n	C_i	On garde notre hypothèse
0,001	42	91.87	Oui
0.025	42	76.2	Oui
0.05	42	72.15	Oui

Pour α = 0.05, on a pour les 10 chiffres :

Chiffre	K	C_i	On garde notre hypothèse
0	49.85	91.87	Oui
1	55.58	76.2	Oui
2	44.34	72.15	Oui
3	51.53	72.15	Oui
4	64.17	72.15	Oui
5	42	72.15	Oui
6	51.46	72.15	Oui
7	56.23	72.15	Oui
8	45.84	72.15	Oui
9	69.64	72.15	Oui

On a que pour les 10 chiffres tout les tests passe bien, comme α = 0.05 est notre α le plus restrictif parmi nos α testé. On a donc que les décimales de π réussissent bien le test du Gap.

2.3 Test du Poker

Ce test statistique permet de tester l'adéquation d'un échantillon de données à une famille de lois de probabilités. Pour ce test, les données sont organisées sous forme de tuples, chaque valeur de ces tuples appartiennent à une intervalle et chaque tuple sera classé en fonction de la répartition de ses valeurs dans les différent intervalles, on a choisit d'avoir 10 intervalles dont chacun est un des 10 chiffre. Ainsi nous comptons le nombre de tuples qu'on a généré dans chaque classe et ceci est notre hypothèse.

On a 5 classes :

— 0 : chaque valeurs du tuple sont différents donc 5 chiffres différents ;
— 1 : on a 1 paire, donc 2 valeurs sont identiques, donc 4 chiffres différents ;
— 2 : on a 1 brelan, 3 valeurs identiques, ou 2 paires, donc 3 chiffres différents ;
— 3 : on a 1 full, 1 brelan et 1 paire, ou 1 carré, 4 valeurs identiques, donc 2 chiffres différents ;
— 4 : on a 1 poker, on a 5 valeurs identiques.

Nous avons aussi besoin d'une hypothèse nulle notée H_0 qui considèrent que les données suivent une lois de probabilité donnée et nous avons notre hypothèse qui sont nos données obtenu expérimentalement. Sur base de ces hypothèse, le reste du Test se déroule avec le test du χ^2.

On a que les probabilités pour chaque classes sont :

$$P_r = \frac{\left\{ {k \atop r} \right\} \cdot d(d-1)...(d-r+1)}{d^k}$$

Tel que r = le nombres de chiffres différents, k = le nombre de chiffre analyser dans la séquence, d = le nombre d'intervalles et

$$\left\{ {k \atop r} \right\}$$

est le Nombre de Sterling pour k = k, et n = r.

Pour les décimales de π on choisit 10 intervalle et chaque intervalle contient 1 chiffres. Le test du poker se basant sur des donnée organisées en tuples de 5, les donnée seront extraites par groupe de 5 des décimales de π. On a donc que :

	Résulats		
Classe	Théorique	Expérimentaux	Erreur relative(%)
Poker	20	13	-35
Carré ou full	2700	2644	-2.07
Brelan ou double paire	36 000	36172	0.47
Paire	100 800	100 670	0.29
Rien	60 480	60 501	0.03

Maintenant effectuons le test du χ^2 sur nos 2 hypothèse :

α	K_n	C_i	On garde notre hypothèse
0.001	4.61	18.47	Oui
0.025	4.61	11.14	Oui
0.05	4.61	9.49	Oui

On a donc que les décimales de π passe bien le test du Poker pour nos alpha.

2.4 Conclusion

On a donc que les 1.000.000 de premières décimales de π suivent bien une loi uniforme, tout nos tests le confirme bien.

3 Générateur de nombre pseudo-aléatoire avec les décimales de π

3.1 Principe de notre générateur

Pour notre générateur de nombre pseudo-aléatoire nous avons choisit d'utiliser la congruence linéaire, et le caractère pseudo-aléatoire sera le temps actuelle sur la machine. Nous avons donc une valeur pour a, c, et m utilisé pour la congruence linéaire, un index compris entre 0 et 1.000.000, et bien sûr les 1.000.000 de première décimales de π dans un string.

Les valeurs choisies pour la congruence linéaire sont a = 41, c = 11, et m = 1.000.000. Ces valeurs respectent le Théorème de Hull et Dobel pour maximiser la période pour revenir à notre index de départ. On a que m = 1.000.000 car les index pour π vont de 0 à 999.999, on doit aussi avoir que c est premier avec m, donc 11 est premier avec m. On doit aussi avoir que b est un multiple de 4 si 4 est un diviseur de m, ce qui est le cas. On a que a = b + 1, et que b est un multiple de p $\forall$ p diviseur premier de m. On a donc que 2 et 5 sont les diviseurs premier de m, et donc b est un multiple de 2, 4 et 5. On a choisit b = 40, et donc a = 41.

Quand on initialise le générateur, il initialise l'index de base avec le temps actuelle avec la méthode time.time(). À chaque fois qu'on demandera un nombre à notre générateur, il assignera $(((a*index)+c)\%m)$ à l'index et retournera "0." suivit par les n (16 par défaut) chiffres à partir de cet index dans les décimales de π, sauf si index vaut plus de 999984, il retournera alors (999.999 - index) chiffres.

3.2 Comparaison de notre générateur avec celui de Python

Pour comparé notre générateur avec celui de Python on va voir si les 2 générateur suivent bien une loi uniforme, pour ce faire nous allons lancer nos tests avec comme valeur pour les tests un certains nombres de nombre généré par les 2 générateurs. Et nous compareront les valeurs expérimentales avec les valeurs théorique qu'un générateur aléatoire devrait idéalement retourner.

Le "meilleur" générateur indépendamment du temps pour générer un nombre sera celui qui à la valeur K la plus petite lorsque l'on appellera le test du χ^2, donc celui qui est le plus proche d'une loi uniforme.

3.3 Test du χ^2

Pour faire le test du χ^2 sur les générateurs nous allons faire 10 intervalles, chaque intervalle sera un espace de 0.1, on aura donc l'intervalle 1 qui comprend [0,0.1[, l'intervalle 2 [0.1,0.2[, etc jusqu'à l'intervalle 10 [0.9,1[. On génère 1 000 000 de nombres et on place chacun dans le bon intervalle pour compter le nombre de nombre généré dans chaque intervalle.

Notre probabilité théorique dans chacun des intervalles est de 100 000 comme dans un générateur idéal chaque nombre à la même probabilité d'apparaître.

Voici nos résultats en moyenne après avoir lancer 20 fois le test :

α	K_n du gén de Python	K_n de notre gén	C_i	Le gén de Python passe le test	Notre gén passe le test
0,001	7.68	5.50	27.88	Oui	Oui
0.025	7.68	5.50	19.02	Oui	Oui
0.05	7.68	5.50	16.91	Oui	Oui

On peut donc remarqué que les 2 générateurs passent bien le test du χ^2. Et que notre générateur à une valeur pour K_n qui en moyenne est plus proche de 0 qui pour le générateur de Python, on a donc que notre générateur respecte mieux une loi uniforme.

3.4 Test du Gap

Pour ce test on va généré 100 000 nombres et changé chaque nombre par sa première décimale, donc c'est comme pour le test du χ^2, on a 10 intervalles égaux de 0.1, et on ajoute ce chiffre dans un string, c'est la séquence de nombre à tester. Donc on effectue le test du gap sur cette séquence, et pour les 10 chiffres.

On va faire 20 fois nos tests et voir en moyenne combien de fois le test du Gap réussi pour nos 3 α, et la valeur de K_n moyenne pour les 2 générateur.

On a donc pour 600 tests du Gap (On effectue 10 test pour chacun des chiffres, et ce pour les 3 α, et ce 20 fois) :

α	K_n du gén de Python	K_n de notre gén	C_i	Réussite gén de Python (%)	Réussite de notre gén (%)	Le générateur de Python passe le test	Notre générateur passe le test
0,001	56.74	54.62	92.87	99.5	99.5	Oui	Oui
0.025	56.74	54.62	76.19	97	97	Oui	Oui
0.05	56.74	54.62	72.15	94.5	95	Oui	Oui

On a donc que les 2 générateurs passe bien le test du gap et que le pourcentage de réussite des tests pour les 2 générateurs sont assez égaux pour tout les α. On a encore que K_n de notre générateur est légèrement inférieur à celui du générateur de Python.

3.5 Test du Poker

Pour faire le test du Poker sur les générateurs nous allons toujours effectuer avec nos 10 intervalles. Tout d'abord On génère 100 000 de tuples composés du premièr chiffre après la virgule d'un nombre généré aléatoirement (ce qui correspond à un intervalle d'une distance de 0.1). On va placer chacun des tuples dans la bonne classe pour compter le nombre de tuple appartenant a chaque classes. Et on les comparera à la probabilité théorique.

Pour notre probabilité théorique on utilise toujours la formule du point 2.3.

Le but va donc d'être de comparer la probabilité théorique et la probabilité expérimentale par un test du χ^2

Voici nos resultats en moyenne après avoir lancer 20 fois le test :

α	K_n du gén de Python	K_n de notre gén	C_i	Le gén de Python passe le test	Notre gén passe le test
0,001	3.95	3.5	18.47	Oui	Oui
0.025	3.95	3.5	11.14	Oui	Oui
0.05	3.95	3.5	9.49	Oui	Oui

On peut donc remarqué que les 2 générateurs passent bien le test du Poker. Et encore une fois, la valeur de K_n pour notre générateur est inférieur à celui du générateur de Python.

3.6 Conclusion

On a donc que les 2 générateur passent bien tout les tests, donc on a bien qu'ils suivent tout 2 une loi uniforme. Notre générateur a pour chaque test une valeur moyenne pour K_n inférieure à celui de Python, on a donc que le notre est plus proche d'une loi uniforme. Mais notre générateur est aussi plus lent que celui de Pyhton.

Cette conclusion est pour une génération de au maximum 1 000 000 de nombre aléatoire par test, la conclusion pourrait éventuellement changer pour un très grand nombre de génération.