Timothée
Muller

L'essentiel de Simulation

29 mars 2013

M. Dossantos-Uzarralde

Table des matières

1 Génération de variable aléatoire

Problématique :
- Comment générer une suite de nombres qui sont la réalisation d'une suite de variables aléatoires réelles indépendantes et de même loi ?
- Cet échantillon est-il une réalisation fidèle de cette variable aléatoire ?

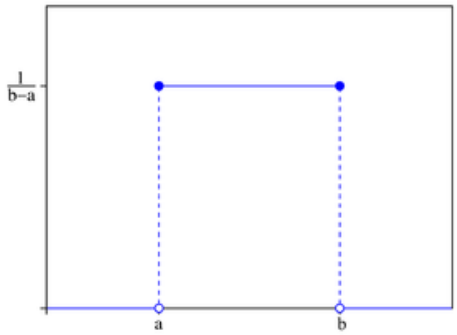
1.1 Rappels de probabilité

1.1.1 La loi uniforme

La loi uniforme est une loi continue a support fini.

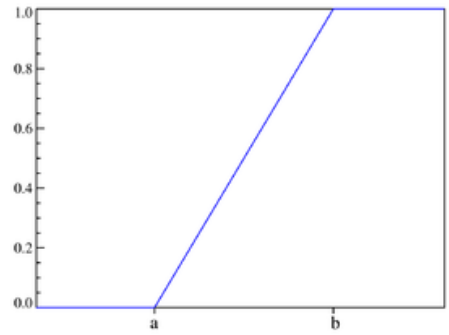
Elle est uniquement caractérisée par deux paramètres, souvent notés a et b. On dit que X suit une loi uniforme de paramètres a et b (se note, $X \sim> \mathcal{U}(a,b)$) si elle a pour densité

$$f_X(x) = \frac{1}{b-a}.\mathbb{1}_{[a,b]}(x)$$



et pour fonction de répartition

$$F_X(x) = \frac{x-a}{b-a}.\mathbb{1}_{[a,b]}(x) + \mathbb{1}_{]b,+\infty[}(x)$$



Sans perte de généralité, nous allons par la suite étudier cette loi uniquement pour les paramètres $a = 0$ et $b = 1$. En effet, il est assez simple de se ramener au cas $\mathcal{U}(a,b)$ en partant de $\mathcal{U}(0,1)$ par ce changement de variable :

$$\text{Si : } X \sim> \mathcal{U}(0,1) \text{ et } Y = (b-a).X + a$$
$$\text{alors : } Y \sim> \mathcal{U}(a,b)$$

inversement :

$$\begin{array}{c}\text{Si : } Y \sim> \mathcal{U}(a,b) \text{ et } X = \frac{Y-a}{b-a} \\ \text{alors : } X \sim> \mathcal{U}(0,1)\end{array}$$

1.1.2 Les moments

Après avoir créé un échantillon aléatoire suivant une loi uniforme, il faudra vérifier la validité de cet échantillon. Nous serons donc amenés à effectuer des tests statistiques qui portent essentiellement sur les paramètres que sont les moments centrés.
Voici un court rappel sur la notion de moments centrés.

1.1.2.1 Génératrice des moments

Avant de passer au calcul des moments à proprement parlé, il existe une fonction qui permet par simple dérivation de retrouver les moments, c'est la fonction génératrice des moments et elle se calcule simplement par :

$$M_X(t) = \int_{-\infty}^{\infty} e^{tx} f(x)\,\mathrm{d}x$$

1.1.2.2 La moyenne

La moyenne est une mesure de position d'une distribution. L'échantillon se trouvera donc autour de cette valeur. Elle est très souvent utilisée car très simple à calculer (une somme puis une division) et utilise donc un minimum de temps de calcul, ce qui pourra être intéressant sur de grandes simulations.

Cette moyenne est aussi appelée espérance (grâce à la fonction espérance $\mathbb{E}$) et se note communément μ

$$\mu = \mathbb{E}(X) = \int_{\mathbb{R}} x\, f_X(x)\,\mathrm{d}x$$

Dans le cas de la loi uniforme $\mathcal{U}(0,1)$ on a :

$$\begin{aligned}\mu &= \int_{\mathbb{R}} x\, \mathbb{1}_{[0,1]}(x)\,\mathrm{d}x = \int_0^1 x\mathrm{d}x \\ &= \left[\frac{x^2}{2}\right]_0^1 = \frac{1}{2}\end{aligned}$$

Elle peut aussi se calculer à partir de la fonction génératrice :

$$\mu = M'_X(0)$$

De plus, l'espérance a une propriété particulière, c'est-à-dire que par le Théorème du transfert on a :

$$\mathbb{E}\big(\varphi(X)\big) = \int_{\mathbb{R}} \varphi(x) f_X(x)\,\mathrm{d}x$$

Ceci va nous permettre de calculer les autres moments.

1.1.2.3 La variance

La variance est une mesure de dispersion d'une distribution. L'échantillon sera plus ou moins étalé autour de la moyenne. C'est le second paramètre qui définit la plupart des loi de probabilité.

Cette variance se note communément σ^2, avec σ étant défini comme l'écart-type.

Aparté :
l'écart-type est une mesure qui a même dimension que les valeurs de l'échantillon, il est donc souvent utilisé en pratique pour avoir un ordre d'idée de la dispersion de l'échantillon.

$$\begin{aligned}\sigma^2 &= \mathrm{Var}(X)\\ &= \mathbb{E}\left[(X-\mu)^2\right]\\ &= \mathbb{E}(X^2)-\mu^2\\ &= \int_{\mathbb{R}} x^2 f_X(x)\,\mathrm{d}x - \left(\int_{\mathbb{R}} x\, f_X(x)\,\mathrm{d}x\right)^2\end{aligned}$$

Dans le cas de la loi uniforme $\mathcal{U}(0,1)$ on a :

$$\begin{aligned}\sigma^2 &= \int_{\mathbb{R}} x^2 \mathbb{1}_{[0,1]}(x)\,\mathrm{d}x - \mu^2 = \int_0^1 x^2 \mathrm{d}x - \mu^2\\ &= \left[\frac{x^3}{3}\right]_0^1 - \frac{1}{2}^2 = \frac{1}{12}\end{aligned}$$

Elle peut aussi se calculer à partir de la fonction génératrice :

$$\sigma^2 = M''_X(0) - [M'_X(0)]^2$$

1.1.2.4 L'asymétrie

L'asymétrie est une mesure de la symétrie de la distribution. L'échantillon se trouvera plus à droite, à gauche ou centré sur la moyenne.

L'asymétrie se note communément γ_1 et se calcule à partir de la moyenne μ et de l'écart-type σ :

$$\gamma_1 = \mathbb{E}\left[\left(\frac{X-\mu}{\sigma}\right)^3\right]$$

Dans le cas de la loi uniforme $\mathcal{U}(0,1)$ on a :

$$\begin{aligned}\gamma_1 &= \int_{\mathbb{R}} \left[\frac{x-\mu}{\sigma}\right]^3 \mathbb{1}_{[0,1]}(x)\,\mathrm{d}x = \frac{1}{\frac{1}{\sqrt{12}^3}} \int_0^1 (x-\frac{1}{2})^3 \mathrm{d}x\\ &= \sqrt{12}^3 \left[\frac{\left(x-\frac{1}{2}\right)^4}{4}\right]_0^1 = 0\end{aligned}$$

Aparté :
Si le coefficient d'asymétrie est nul, on peut en déduire une propriété sur la médiane. En effet, si la distribution est centrée, on peut montrer que la moyenne est égale à la médiane. On pourra par exemple utiliser le test de Shapiro-Wilk qui porte sur la médiane et non sur la moyenne.

1.1.2.5 L'aplatissement

L'aplatissement est une mesure de l'aplatissement de la distribution. On peut en déduire si les masses de l'échantillon sont regroupées plus ou moins loin de la moyenne.

L'aplatissement se note communément β_2 et se calcule à partir de la moyenne μ et de l'écart-type σ :

$$\beta_2 = \mathbb{E}\left[\left(\frac{X-\mu}{\sigma}\right)^4\right]$$

Dans le cas de la loi uniforme $\mathcal{U}(0,1)$ on a :

$$\begin{aligned}\beta_2 &= \int_{\mathbb{R}} \left[\frac{x-\mu}{\sigma}\right]^4 \mathbb{1}_{[0,1]}(x)\,\mathrm{d}x = \frac{1}{\frac{1}{12^2}} \int_0^1 (x-\frac{1}{2})^4 \mathrm{d}x \\ &= 12^2 \left[\frac{\left(x-\frac{1}{2}\right)^5}{5}\right]_0^1 = \frac{9}{5} \\ \gamma_2 &= \frac{9}{5} - 3 = -\frac{6}{5}\end{aligned}$$

Souvent on utilise le Kurtosis normalisé γ_2 ou excès d'aplatissement.

1.2 Les générateurs de nombres aléatoires

1.2.1 Définition

Un générateur de variables aléatoires est une programme ou un algorithme qui permet de générer des nombres qui vont suivre une certaine loi, ces nombres pourront être dépendants ou indépendants.

1.2.2 Nos générateurs

Actuellement dans le monde de l'assurance deux logiciels de statistiques sont en concurrence directe à savoir R et SAS, le premier est libre et le second est payant. Il semble donc tout à fait intéressant de les confronter, mais aussi avec d'autres outils, comme le C qui sera meilleur en termes de performance dû au fait que c'est un langage compilé ou encore Excel, logiciel très populaire dans le milieu.

Liste des générateurs étudiés (dénomination dans le texte) :
- R (R)
- SAS (SAS)
- Excel (XLS)
- rand (C)
- standard minimal (STM)
- Mitchell et Moore (MEM)

1.2.2.1 Générateur de R

Le générateur d'échantillons de taille 100 de loi uniforme $\mathcal{U}(0,1)$ en R est déjà programmé en interne avec la fonction **RUNIF** :

```
> set.seed(20) # il est possible de fixer la graine
> runif(100)
```

1.2.2.2 Générateur de SAS

Le générateur d'échantillons de taille 100 de loi uniforme $\mathcal{U}(0,1)$ en SAS est déjà programmé en interne avec la fonction **RANUNI** :

```
DATA TableCree ;
   DO i=1 TO 100 ;
      y=ranuni(20) ; // il est possible de fixer la graine
      OUTPUT ;
   END ;
   KEEP y ;
RUN ;
PROC PRINT ; RUN ;
```

1.2.2.3 Générateur de XLS

Le générateur est directement implémenté dans excel par la fonction **RAND** ou **ALEA** , sans que l'on puisse gérer la graine, ce qui en fait l'un des moins bon générateur de variable aléatoire.

1.2.2.4 Générateur de C

Le générateur d'échantillons de taille 100 de loi uniforme $\mathcal{U}(0,1)$ en C est déjà programmé en interne avec la fonction **RAND** :

```
   srand(20) ; // il est possible de fixer la graine

   for (i = 0; i < 100; i++)
   {
      alea = (float) rand()/RAND_MAX ;
      printf("%F\N", alea) ;
   }
```

1.2.2.5 Générateur STM

Le générateur d'échantillons de taille 100 de loi uniforme $\mathcal{U}(0,1)$ à l'aide du générateur Standard Minimal doit être programmé en C :

```
   const long a = 16807, m = RAND_MAX, q = 127773, r = 2836 ;

   for (i = 0; i < 100; i++)
   {
      x = ldiv(alea, q) ;
      test = a*x.rem - r*x.quot ;
      if ( test >= 0 )
      {
         alea = test ;
```

```
    } else {
       alea = test + m ;
    }

    printf("%F\N", alea) ;
}
```

1.2.2.6 Générateur MEM

Le générateur d'échantillons de taille 100 de loi uniforme $\mathcal{U}(0,1)$ à l'aide du générateur de Mitchell et Moore doit être programmé en C :

```
srand(20) ; // il est possible de fixer la graine

for (i = 0; i < 55; i++) {
   alea[i] = (float) rand()/RAND_MAX ;
   printf("%F\N", alea[i]) ;
}

for (i = 55; i < 100; i++)
{
   alea[i] = modulo(alea[i-24] + alea[i-55],1) ;

   printf("%F\N", alea[i]) ;
}
```

1.3 Méthode de test

Après avoir programmé les différents générateurs, on va maintenant se demander quelles sont les différences entre ceux-ci ? Peut-on affirmer que l'un est significativement meilleur que les autres ?

Pour pouvoir juger de l'efficacité d'un générateur, on va donc lui faire passer plusieurs tests.

1. Conditions d'un bon générateur (en assurance)
2. Générations d'échantillons
3. Premières vérifications
4. Paramètres sur l'échantillon
5. Test du Chi^2
6. Gap test (randtoolbox)

1.3.1 Un bon générateur

D'après Cournot : "Le hasard, c'est la rencontre de deux séries causales indépendantes."

On va donc définir un bon générateur comme :

- prenant toutes les valeurs possibles, ici dans $[0,1] \in \mathbb{R}$
- sans liens entre un tirage et le suivant c'est-à-dire qu'ils sont indépendants
- ni de séries qui se répète de façon récurrente

1.3.2 Génération d'échantillon

Après avoir codé les différents générateurs de variables aléatoires, on automatise la génération des différents échantillons que l'on voudra tester à l'aide d'un code en R :

```
generation_echantillon <- function() {
   echant <- vector()

   nombres <- c("100", "1000", "10000", "100000", "1000000")
   programmes <- c("C", "MEM", "STM")

   for (i in nombres) {
      for(nom in programmes) {
         echant <- as.real(system(paste("../", nom, "/BASE␣", i, sep = ""),
                  intern = TRUE))
         write.table(echant, paste("RESULTATS/RESULTAT", nom, i, sep="_"),
                  quote = FALSE, row.names = FALSE, col.names = FALSE)
      }
      echant <- runif(i)
      write.table(echant, paste("RESULTATS/RESULTAT", "R", i, sep="_"),
               quote = FALSE, row.names = FALSE, col.names = FALSE)
   }
}
```

Les résultats sont donc envoyés dans un fichier dont le nom est :
resultat_nom-du-générateur_nombres-de-tirage

J'ai donc décidé de travailler sur des échantillons de taille 100, 1000, 10000, 100000 et 1000000, que je vais ainsi pouvoir étudier avec la même méthode de sorte qu'il soit possible de les comparer.

1.3.3 Premières vérifications

Dans un premier temps, on peut commencer par comparer les différents histogrammes pour avoir une première idée de l'efficacité de nos générateurs, ceci à l'aide de cette fonction :

```
print_hist <- function(v = dat$N10 ) {
   par(mfrow=c(2,3))

   for ( i in 1:length(v) ) {
      hist(v[[i]], main=names(v)[i], xlab="LES␣VALEURS␣PRISES",
                                       ylab="FRÉQUENCE")
   }
}

print_hist()
```

FIGURE 1.1 – N = 100

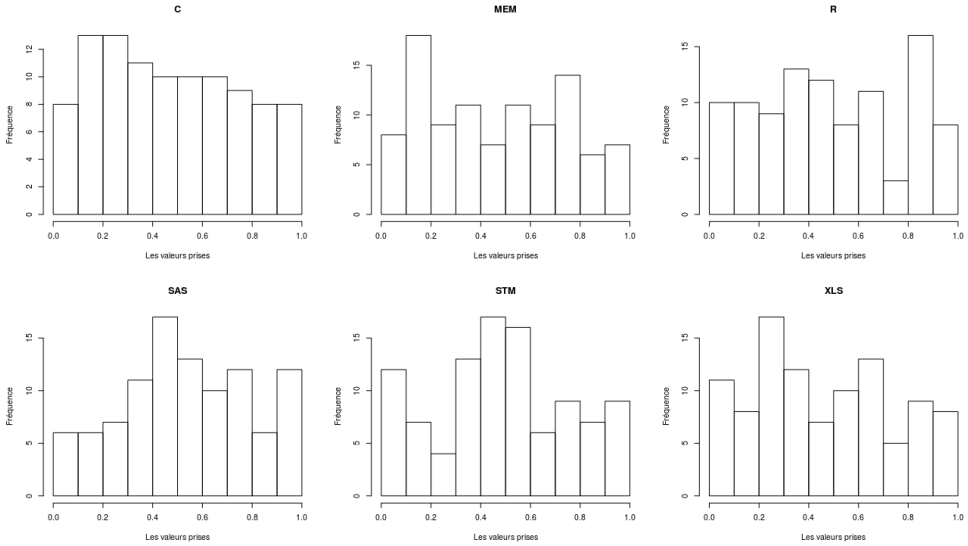


FIGURE 1.2 – N = 1000

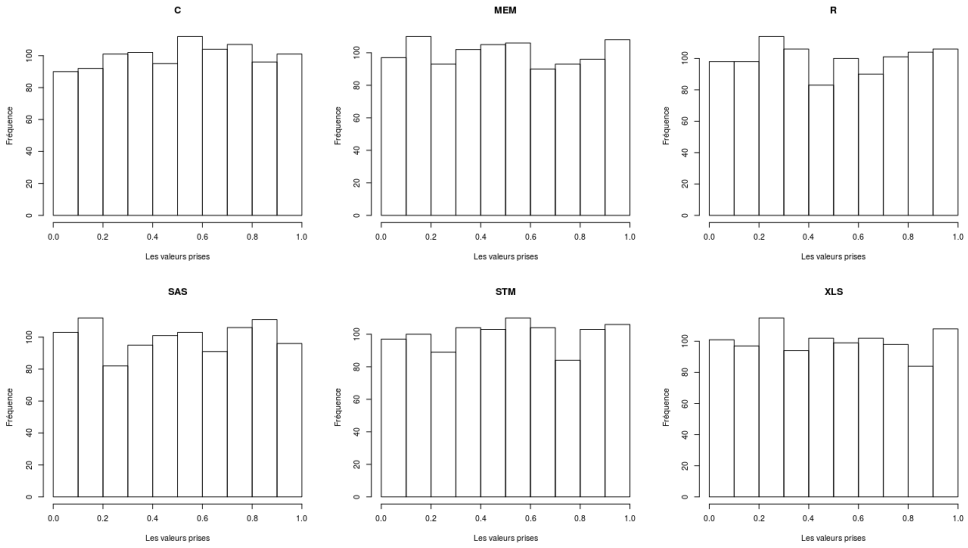
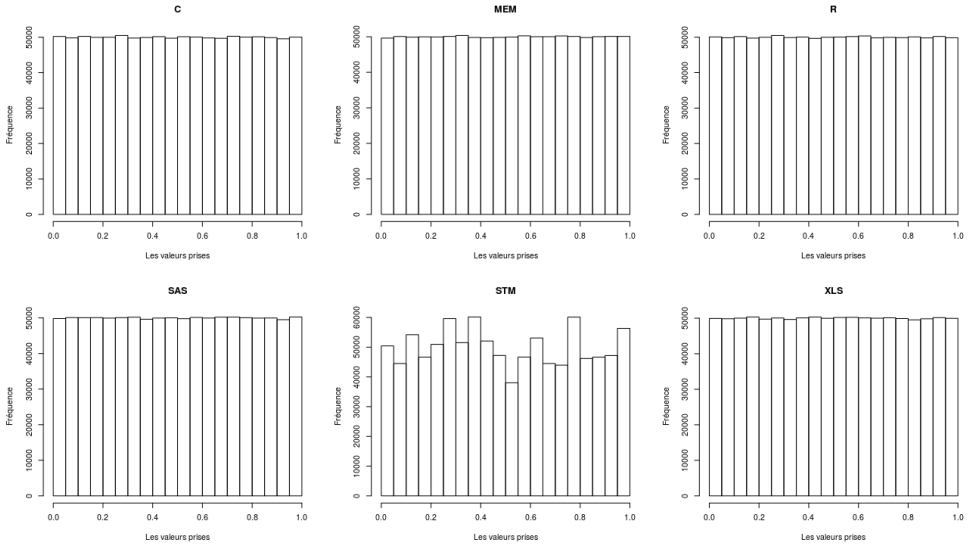


FIGURE 1.3 – N = 1000000



On peut donc en conclure que le générateur STM sera moins performant car sa distribution ne semble même pas converger vers la loi uniforme $\mathcal{U}(0,1)$ et à l'inverse les 5 autres ont l'air acceptables.

1.3.4 Paramètres sur l'échantillon

Avant tout test statistique, il peut être intéressant de se tourner vers les valeurs remarquables de la loi uniforme $\mathcal{U}(0,1)$ à savoir :

- la moyenne qui a pour valeur théorique 0.5
- l'écart-type qui a pour valeur théorique 0.28
- skewness ayant pour valeur théorique 0
- kurtosis ayant pour valeur théorique -1.2
- les quantiles (0, 25%, 50%, 75%, 100%) qui ont pour valeurs théoriques (0, 25%, 50%, 75%, 100%)

J'ai donc décidé de créer les tableaux de valeurs calculées sur les différents échantillons triés pour chaque tailles d'échantillon et ce grâce à la fonction suivante :

```
param_test <- function(donnees) {
   x <- numSummary( c(donnees), quantiles=c(0,.25,.5,.75,1),
               statistics=c("MEAN","SD","QUANTILES","SKEWNESS","KURTOSIS"))
   cbind("N"=x$n, x$table)
}
```

Dans un premier temps, on peut constater sur l'exemple du générateur de R l'évolution en fonction de la taille de l'échantillon.

n	mean	sd	skewness	kurtosis	0%	25%	50%	75%	100%
1e+02	0.491	0.286	7.76e-02	-1.20	4.36e-03	0.276	0.447	0.708	0.992
1e+03	0.501	0.291	1.77e-02	-1.24	1.90e-04	0.251	0.501	0.762	0.999
1e+04	0.498	0.289	9.17e-04	-1.20	8.60e-05	0.249	0.502	0.747	1.000
1e+05	0.502	0.288	-6.51e-03	-1.19	4.92e-06	0.252	0.503	0.751	1.000
1e+06	0.500	0.289	9.44e-05	-1.20	1.00e-06	0.250	0.500	0.750	1.000
théo	0.5	0.28	0	-1.2	0	0.25	0.50	0.75	1.00

On constate que la loi des grands nombres est bien vérifiée, car avec le même générateur de nombres aléatoires, on voit bien que les différents paramètres empiriques tendent bien vers leurs valeurs théoriques respectives.

On peut maintenant comparer les différents générateurs entre eux avec les résultats suivants pour N = 1000 :

géné	mean	sd	skewness	kurtosis	0%	25%	50%	75%	100%
C	0.5082	0.2862	-0.041080	-1.163	0.0004750	0.2632	0.5114	0.7535	0.9992
MEM	0.4981	0.2887	0.034067	-1.190	0.0006470	0.2400	0.4929	0.7463	0.9999
R	0.5015	0.2912	0.017747	-1.244	0.0001905	0.2512	0.5006	0.7624	0.9995
SAS	0.5021	0.2915	-0.021123	-1.247	0.0004493	0.2405	0.5043	0.7679	0.9991
STM	0.5043	0.2876	-0.002444	-1.182	0.0000080	0.2568	0.5063	0.7444	0.9998
XLS	0.4961	0.2876	0.034135	-1.199	0.0032338	0.2481	0.4897	0.7381	0.9999

Puis avec nos échantillons de taille N = 1000000 :

géné	mean	sd	skewness	kurtosis	0%	25%	50%	75%	100%
C	0.4997	0.2887	1.298e-03	-1.200	1.000e-06	0.2498	0.4996	0.7497	1
MEM	0.5003	0.2886	-9.300e-04	-1.200	2.000e-06	0.2505	0.5006	0.7502	1
R	0.4999	0.2886	9.441e-05	-1.200	1.004e-06	0.2502	0.5001	0.7498	1
SAS	0.5000	0.2886	-1.353e-04	-1.200	1.574e-06	0.2500	0.5001	0.7498	1
STM	0.4971	0.2896	4.058e-02	-1.207	8.000e-06	0.2542	0.4783	0.7560	1
XLS	0.4999	0.2885	3.742e-04	-1.198	1.034e-06	0.2501	0.5000	0.7494	1

D'après les valeurs obtenues, on peut en conclure que le générateur STM est une fois encore le plus loin du résultat attendu car ses valeurs empiriques sont les plus éloignées de leurs valeurs théoriques associées de toute la série de générateurs.

Il faut donc appliquer de nouveaux tests pour vérifier si d'autres différences entre les générateurs vont nous permettre de définir si l'un de ceux-ci est significativement meilleur que les autres.

1.3.5 Test du Chi^2

1.3.5.1 Mise en place

Étant donné que nous sommes entrain de tester des distributions suivant des lois uniformes et comme il existe relativement peu de test paramétrique qui conviennent à cette loi, j'ai choisi de me diriger vers le plus basique des tests à savoir le Test d'Adéquation du Chi^2 qui est définit de la façon suivante :

$$H_0 : \text{"l'échantillon suit une loi Uniforme de paramètres 0 et 1 ; } \mathcal{U}(0,1)\text{"}$$
$$H_1 : \text{"l'échantillon ne suit pas une loi Uniforme de paramètres 0 et 1 ; } \mathcal{U}(0,1)\text{"}$$

Sous R ce test est implémenté par la fonction **chisq.test** .

Ce test se basant sur le tableau de contingence d'une loi, il va donc falloir procéder à une découpe de notre distribution en différentes classes.
On va donc procéder de la façon suivante :

```
classes = table(trunc(10*distribution))
```

En multipliant par 10 la distribution, elle se trouve alors entre [0,10], puis la fonction **trunc** permet de séparer en 10 classes C_k allant de $[k-1, k]\ \forall k \in [[1, 10]]$ cette nouvelle distribution. Enfin, la fonction table permet de créer le tableau à tester.

```
chisq_test <- function(donnees) {
   classes = table(trunc(10*donnees))
   chisq.test(classes)$p.value
}

test_all <- function(fu) {
   do.call(rbind, lapply(dat, function(to_apply) { lapply(to_apply, fu) }))
}

chisq_test_all <- function() {
   test_all(chisq_test)
}
```

1.3.5.2 Résultats

Puis en affichant **mat** on obtient la matrice des p-value :

	C	MEM	R	SAS	STM	XLS
N100	0.955835	0.202268	0.289667	0.191687	0.048716	0.304126
N1000	0.897763	0.873987	0.655854	0.568739	0.767582	0.735908
N10000	0.0680178	0.93605	0.805938	0.929091	0.0954257	0.177434
N100000	0.188039	0.248141	0.543423	0.334491	7.01545e-107	0.28026
N1000000	0	0.776246	0.843519	0.986214	0	0.344678

Pour les p-value supérieurs à 5% le test est non significatif, l'on retient alors H_0 : "l'échantillon suit une loi Uniforme de paramètre 0 et 1 ; $\mathcal{U}(0,1)$" avec une puissance $power = 1 - \beta$ (β l'erreur de seconde espèce) qu'il faudrait encore calculer pour maitriser totalement l'erreur commise par ce test.

1.3.5.3 Conclusion

Au vu des résultats on peut donc en conclure que les générateurs de variables aléatoires les plus performants sont ceux de SAS, R ainsi que Mitchell et Moore. A l'inverse, les autres sont relativement peu efficaces avec une grande déception pour le générateur d'Excel 2013 (v. 15.0), logiciel payant pourtant tellement utilisé dans le domaine de l'assurance.

1.3.6 Gap test (randtoolbox)

Les tests précédents étant effectués sur un unique vecteur de nombre aléatoire et non directement sur le générateur lui-même, j'ai donc complété cette analyse avec un test qui permet de déterminer s'il y a un lien fort entre un tirage et son ou ses successeur(s).

Le "gap test" est issu du package "randtoolbox" disponible dans les dépôts officiels de R. Il est parfois connu sous le nom de "run test".

1.3.6.1 Mise en place

Ce test va dans un premier temps convertir en binaire les différents nombres aléatoires puis tester le nombre de 0 qui se suivent dans l'ensemble de la série.
Ce test part du principe que si un grand nombre de 0 se suivent, le générateur ne sera pas le meilleur car il y a normalement 1 chance sur 2 d'avoir un 0 ou un 1. On a donc le test suivant :

$$H_0 : \text{"les suites de 0 sont aléatoires"}$$
$$H_1 : \text{"il y a des suites de 0 trop longues"}$$

On applique donc le test à l'ensemble de nos données :

```
gap_test <- function(donnees) {
   gap.test(donnees, echo=FALSE)$p.value
}
gap_test_all <- function() {
   test_all(gap_test)
}
```

1.3.6.2 Résultats

Voici les résultats contenus dans **gap** :

	C	MEM	R	SAS	STM	XLS
N100	0.175574	0.0693842	0.590122	0.231964	0.0999984	0.761877
N1000	0.916837	0.344129	0.0709416	0.0795184	0.915902	0.114031
N10000	0.251546	0.724912	0.766005	0.644002	0.00424072	0.304353
N100000	0.89763	0.834583	0.0720055	0.0957013	1.76931e-288	0.686852
N1000000	0.0206244	9.58826e-08	0.685172	0.621592	0	0.0940449

1.3.6.3 Conclusion

Ici les résultats nous permettent d'affirmer que les générateurs de R et SAS ne rejettent pas l'hypothèse nulle et que le test est largement significatif pour les générateurs de MEM et STM, qui ne sont donc pas assez aléatoires.

1.4 Comparaison des générateurs

On peut donc en conclure plusieurs choses sur ces générateurs de variables aléatoire.
Le générateur le moins efficace de ceux testés est le générateur Standard Minimal car il ne passe réellement aucun des tests effectués, il semble donc aujourd'hui, avec la puissance de l'outil informatique, tout à fait désuet.
Ensuite, les autres générateurs à l'exception de celui de C n'ont pas de différence significative en termes de distribution. Ceci peut être dû à des contraintes de performance exigées en C notament la contrainte de temps puisque ce langage sert à créer bon nombres de programmes de jeux en temps réel par exemple.
Enfin, on peut aussi dire qu'il est préférable d'utiliser les générateurs de R et SAS plutôt que celui de Excel car leurs résultats sont meilleurs, mais aussi plus homogènes. Et ceci peut s'expliquer car leurs générateur sont en fait les mêmes générateurs de variables aléatoires basés sur le même algorithme, celui de Mersenne Twister qui est disponible en ligne par exemple à cette adresse : Wikipedia(en) Mersene Twister
Dans le cadre d'une étude plus poussée, il serait aussi intéressant de tester directement les générateurs de variables aléatoires et non certaines distributions que l'on peut en tirer. Ce qui nous permettrait alors d'avoir des résultats plus proches de la réalité, car certaines distributions peuvent être le résultat de la chance.

2 Simulation de variable aléatoire Normale

Dans le chapitre précédent nous avons comparé différents générateur de variable aléatoire uniforme $\mathcal{U}(0,1)$. Comme la loi Normale est l'une des principales lois utilisé dans la modélisation de phénomènes aléatoires (application du TCL par exemple) il semble être intéressant de pouvoir générer de bon échantillons qui suivent cette loi.

Sans perte de généralités, je vais uniquement étudier la Loi Normale Centrée Réduite, car si $Y \sim> \mathcal{N}(\mu, \sigma)$ alors en posant $X = \frac{Y-\mu}{\sigma}$, la loi de X est une Loi Normale Centrée Réduite $X \sim> \mathcal{N}(0,1)$.

On sait que pour une variable aléatoire X qui suit une loi de fonction de répartition F, on peut calculer une distribution $(x_1, ..., x_n)$ de cette loi en générant tout d'abord un échantillon $(u_1, ..., u_n)$ qui suit une loi U suivant la loi $\mathcal{U}(0,1)$ puis grâce à la relation $F(x) = u$, il suffit de poser $x_i = F^{-1}(u_i)$ pour obtenir un échantillon qui suit une loi de fonction de répartition F.

Étant donné les résultats du premier chapitre et comme le générateur de variable aléatoire de R ainsi que de SAS sont les mêmes, je trouve pertinent de l'étudier en particulier pour en prouver ses performances.

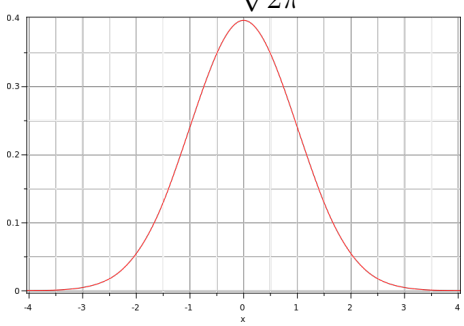
2.1 Rappels sur la Loi Normale

Si une variable aléatoire X suit la Loi Normale centrée réduite $\mathcal{N}(0,1)$ alors on a les différentes valeurs caractéristiques suivantes.

- $\mu = 0$
- $\sigma = 1$
- $skeness = 0$
- $kurtosis = 0$ (normalisé)

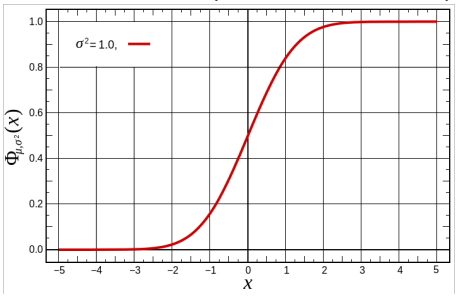
Elle a pour fonction de densité

$$f_X(x) = \frac{1}{\sqrt{2\pi}}.e^{-\frac{x^2}{2}}$$



et pour fonction de répartition

$$F_X(x) = \frac{1}{2}\left(1 + erf(\frac{x}{\sqrt{2}})\right)$$



2.2 Les méthodes de simulation

2.2.1 Utilisation de $erfinv$

Tout d'abord, la fonction appelée erf est la fonction d'erreur parfois appelée de Gauss qui sera utilisée par la suite, $erfinv$ n'est alors que erf^{-1}.

Comme cité dans les rappels, on a la fonction de répartition de la Loi Normale Centrée Réduite, à savoir $F_X(x) = \frac{1}{2}\left(1 + erf(\frac{x}{\sqrt{2}})\right)$.

Comme la méthode nous le demande, il faut alors trouver l'inverse de cette fonction, F_X^{-1} :

$$\begin{aligned} F_X(x) &= u \\ \frac{1}{2}\left(1 + erf(\frac{x}{\sqrt{2}})\right) &= u \\ erf(\frac{x}{\sqrt{2}}) &= 2.u - 1 \\ \frac{x}{\sqrt{2}} &= erfinv(2.u-1) \\ x &= \sqrt{2}\ erfinv(2.u-1) \\ F_X^{-1}(u) &= \sqrt{2}\ erfinv(2.u-1) \end{aligned}$$

Cette fonction n'étant pas directement implémenté sous R, elle est malgré tout rapidement implémentable :

```
erfinv <- function(x) {
   qnorm((1 + x)/2) / sqrt(2)
}
```

Pour avoir un échantillon de Loi Normale Centrée Réduite il suffit alors de faire comme suit :

```
unif_to_norm <- function(x) {
   sqrt(2)*erfinv(2*x-1)
}

echantillon_uniforme <- runif(100)

echantillon_normale <- unif_to_norm(echantillon_uniforme)
```

Cette méthode nous permet de comprendre la façon dont est créé l'échantillon de loi normale, dans la suite, j'utiliserai plutôt la fonction suivante, permettant de maitrisé la graine lors des boucles de test :

```
runif_seed <- function(nb, seed=0) {
   set.seed(seed)
   runif(nb)
}

rand_unif_to_norm <- function(nb, seed=20) {
   sqrt(2)*erfinv(2*runif_seed(nb, seed)-1)
}
```

2.2.2 Méthode de rejet

La méthode générale de rejet est la suivante :
- Génération de $U_1 \sim> \mathcal{U}(0,1)$
- Calcul de $X = G^{-1}(U_1)$
- Génération de $U_2 \sim> \mathcal{U}(0,1)$
- Si $c.g(X).U_2 \leq f(X)$ alors on garde X sinon on rejette
  (avec $c > 1$ et $f(x) \leq c.g(x) \forall x$)

L'idée globale de cette méthode est de garder un point choisit aléatoirement dans un premier temps seulement s'il ne s'éloigne "pas trop" de l'endroit ou il devrait être placer théoriquement.

2.2.2.1 Par Cauchy

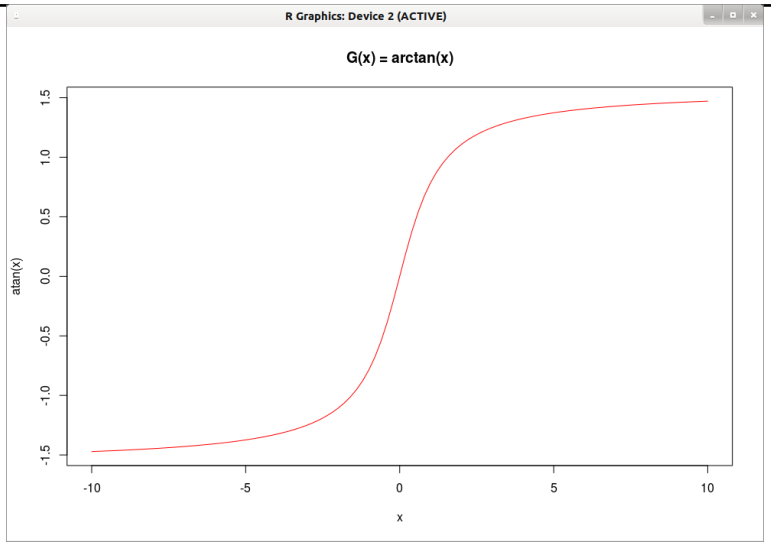
Dans le cas de la Méthode de rejet par Cauchy, on utilise :

$$\begin{aligned} f(x) &= f_X(x) = \frac{1}{\sqrt{2\pi}}.e^{-\frac{x^2}{2}} \\ c &= \frac{1}{\pi}\sqrt{\frac{2\pi}{e}} \\ g(x) &= \frac{1}{1+x^2} \end{aligned}$$

$$\begin{aligned} G(x) &= arctan(x) \text{ (calculée comme primitive de g)} \\ G^{-1}(x) &= tan(x) \text{ (calculée comme l'inverse de G)} \end{aligned}$$

Après quelques tests, appliquer directement cette méthode ne fonctionne pas, car on remarque un ensemble d'arrivé pour X qui n'est "que" $G^{-1}([0,1])$ et non $\mathbb{R}$, l'ensemble d'arrivé de la fonction G.

Comme la fonction tangente prend est une bijection de $]-\frac{\pi}{2};\frac{\pi}{2}[\to \mathbb{R}$, j'ai donc décidé d'appliquer une transformation à U_1 à savoir :
$U_1' = U_1 * \pi - \frac{\pi}{2}$
dans ce cas, on a bien : $G^{-1}(]-\frac{\pi}{2};\frac{\pi}{2}[) = \mathbb{R}$



En suivant la démarche à appliquer de la méthode de rejet, il en vient directement le code suivant :

```
rand_rejet_cauchy <- function(nb, seed=20) {
   # initialisation des paramètres
   f <- function(x) { exp((-x^2)/2)/sqrt(2*pi) }
   c <- sqrt(2*pi/exp(1))/pi
   g <- function(x) { 1/(1+x^2) }
   G_moins_un <- tan
   nb2 <- floor(nb/10)
   k = 1 # nombre de série aléatoire
   i = 0 # le nombre de valeur déjà collectées
   echantillon_normal <- 1:nb
              # on créé le vecteur de retour pour le remplir par la suite
   while ( i <= nb ) { # tant que l'on a pas toutes les valeurs
      # on applique la méthode
      li <- 1:nb2+(k-1)*nb2
      u_1 <- runif_seed(nb2*k, seed)[li] # on tire nb nouvelles valeurs
      x <- sapply(u_1, function(y) { G_moins_un(pi*y - pi/2)})
      u_2 <- runif_seed(nb2*k, floor(seed/2))[li]

      keeps <- c*g(x)*u_2 <= f(x) # on regarde lesquelles passent le test
      value_kept <- x[keeps] # les valeurs que l'on garde
      echantillon_normal[i+(1:length(value_kept))] <- value_kept
                                          # on les ajoute àla suite
      i <- i + length(value_kept) # nombre de valeurs ajoutées
      k <- k + 1 # tour de boucle suivante
   }
   echantillon_normal[1:nb]
}
```

2.2.2.2 Forme Exponentielle

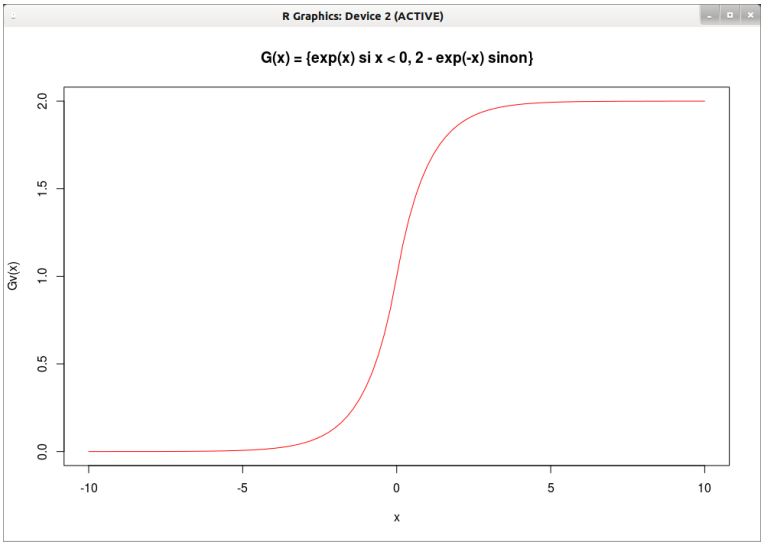
Dans le cas de la Méthode de rejet avec la forme Exponentielle, on utilise :

$$\begin{aligned} f(x) &= f_X(x) = \frac{1}{\sqrt{2\pi}}.e^{-\frac{x^2}{2}} \\ c &= \sqrt{\frac{e}{2\pi}} \\ g(x) &= e^{-|x|} \end{aligned}$$

$$G(x) = \begin{cases} e^x & \text{si } x < 0 \\ 2 - e^{-x} & \text{sinon} \end{cases} \quad \text{(calculée comme primitive de g)}$$

$$G^{-1}(x) = \begin{cases} log(x) & \text{si } x < 1 \\ -log(2-x) & \text{sinon} \end{cases} \quad \text{(calculée comme l'inverse de G)}$$

De même que précédemment, il faut corriger l'ensemble d'arriver de la variable aléatoire uniforme pour qu'elle colle avec l'ensemble de départ de la fonction G^{-1} ou encore celui d'arriver de G sa réciproque, ici il faut donc trouver avoir une loi uniforme sur $[0, 2]$ comme le montre ce graphique.



Comme la fonction ainsi définie est une bijection de $]0; 2[\to \mathbb{R}$, j'ai donc décidé d'appliquer une transformation à U_1 à savoir :
$U_1' = U_1 * 2$
dans ce cas, on a bien : $G^{-1}(]0; 2[) = \mathbb{R}$

En suivant la démarche à appliquer de la méthode de rejet, il en vient directement le code suivant :

```
rand_rejet_exp <- function(nb, seed=20) {
   f <- function(x) { exp(-x^2/2)/sqrt(2*pi) }
   c <- sqrt(2*pi/exp(1))/pi
   g <- function(x) { exp(-abs(x)) }
   G_moins_un <- function(y) {
               if (y < 1)
               {
                  log(y)
               } else {
                  -log(2 - y)
               }
            }

   nb2 <- floor(nb/10)
   k = 1
   i = 0
   echantillon_normal <- 1:nb
   while ( i <= nb ) {
      li <- 1:nb2+(k-1)*nb2
      u_1 <- runif_seed(nb2*k, seed)[li]
      x <- sapply(u_1, function(y) { G_moins_un(pi*y - pi/2)})
      u_2 <- runif_seed(nb2*k, floor(seed/2))[li]

      keeps <- c*g(x)*u_2 <= f(x)
      value_kept <- x[keeps]
      echantillon_normal[i+(1:length(value_kept))] <- value_kept
      i <- i + length(value_kept)
      k <- k + 1
   }
   echantillon_normal[1:nb]
}
```

2.2.3 Méthode de Box-Muller

La méthode de Box-Muller est un procédé très simple, qui permet très rapidement de générer 2 échantillons de variables aléatoires suivantes indépendement la Loi Normale Centrée Réduite, ceci tiens en une simple proposition :

$$X_1 \sim> \mathcal{U}(0,1) \text{ et } X_2 \sim> \mathcal{U}(0,1) \text{ alors}$$
$$(\sqrt{-2.ln(X_1)}.cos(2\pi.X_2), \sqrt{-2.ln(X_1)}.sin(2\pi.X_2))$$

Le code R est donc relativement simple :

```
rand_norm_box_muller <- function(nb, seed=20) {
   X1 = runif_seed(nb, seed)
   X2 = runif_seed(nb, floor(seed/2))
   sqrt(-2*log(X1))*cos(2*pi*X2)
   #sqrt(-2*log(X1))*sin(2*pi*X2)
}
```

2.2.4 Utilisation du Théorème Central-Limite

Supposons que les variables aléatoires $U_i \sim> \mathcal{U}(0,1)$, alors $X_n = \sum_{i=1}^{n} U_i/n$ est une variable aléatoire et par le Théorème Central-Limite, on sait que $X_n \xrightarrow{\mathcal{L}} \mathcal{N}(\mu, \frac{\sigma^2}{n})$ avec les paramètres de la loi uniforme : $\mu = 1/2$ et $\sigma^2 = 1/12$

Pour générer une variable aléatoire qui suit la Loi Normale Centrée Réduite, on peut alors utiliser :

$$Y_n = \frac{X_n - \mu}{\sigma}.\sqrt{n} = \frac{\sum_{i=1}^{n} U_i - n/2}{\sqrt{n/12}} \sim> \mathcal{N}(0,1)$$

Étant donné que la convergence en loi dans ce cas est relativement rapide (efficace pour des valeurs assez petite) et qu'au dénominateur se trouve $\sqrt{n/12}$, en pratique on prendra n = 12, ce qui nous permet de gagner en temps de calcul pour la machine, car il ne reste plus qu'une simple somme. On obtient alors Y_n :

$$Y_n = \frac{\sum_{i=1}^{n} U_i - 12/2}{\sqrt{12/12}} = \sum_{i=1}^{n} U_i - 6 \sim> \mathcal{N}(0,1)$$

Il nous reste alors à coder ce générateur :

```
rand_norm_tcl <- function(nb, seed=20) {
   rowSums(sapply(1:12 , function(col) {runif_seed(nb, seed + col)})) - 6
}
```

2.3 Test des méthodes

Après avoir créé ces différents générateurs de variable aléatoire suivant la Loi Normale Centrée Réduite, comme dans la section 1.3, il va donc maintenant falloir tester les diverses méthodes pour nous permettre d'obtenir rapidement et avec un maximum d'efficacité, un échantillon de variable aléatoire suivant la Loi Normale Centrée Réduite.

Le test d'hypothèse auquel nous allons répondre est le suivant :

H_0 : "le générateur, génère des échantillons suivant la loi $\mathcal{N}(0,1)$"
H_1 : "les échantillons du générateur ne suivent pas la loi $\mathcal{N}(0,1)$"

On va donc mettre en place la méthode de test suivante :
- Mise en place
- Premières vérifications (QQ plot, histogramme)
- Paramètres sur l'échantillon
- Les tests de normalité

2.3.1 Mise en place

Pour pouvoir tester le générateur de variable aléatoire et non 1 seule distribution, cette fois-ci, j'ai décidé de tester,les 10 distributions tirées des 10 graines d'un échantillon pris aléatoirement, mais qui seront les mêmes pour tous les tests.

```
> num_seeds <- floor(runif(10,0,length(.Random.seed)))
> num_seeds
 [1] 613 132 327 525 44 427 411 311 274 513
> seeds <- .Random.seed[num_seeds]
```

Ensuite, il faudra donc appliquer les différents tests aux générateurs (via les seeds) et pour ce faire, nous utiliserons la fonction suivante :

```
apply_test <- function(test, generator, nb, seed) {
   test(generator(nb, seed))$p.value
}
```

2.3.2 Premières vérifications

Dans un premier temps, il semble intéressant de regarder à quoi ressemble notre distribution de manière graphique, de façon à ce que l'on puisse juger à première vu si l'hypothèse de normalité peut être vérifiée ou non.

Pour ce faire, j'ai donc décidé d'utiliser 2 méthodes grahique. La première est l'histogramme auquel j'ai ajouté la fonction de densité empirique ainsi que la fonction de densité de la loi $\mathcal{N}(0,1)$. La seconde est le diagramme quantile-quantile qui va nous permettre d'évaluer la qualité de la normalité des données.

Pour ce faire, on utilise la fonction d'affichage suivante :

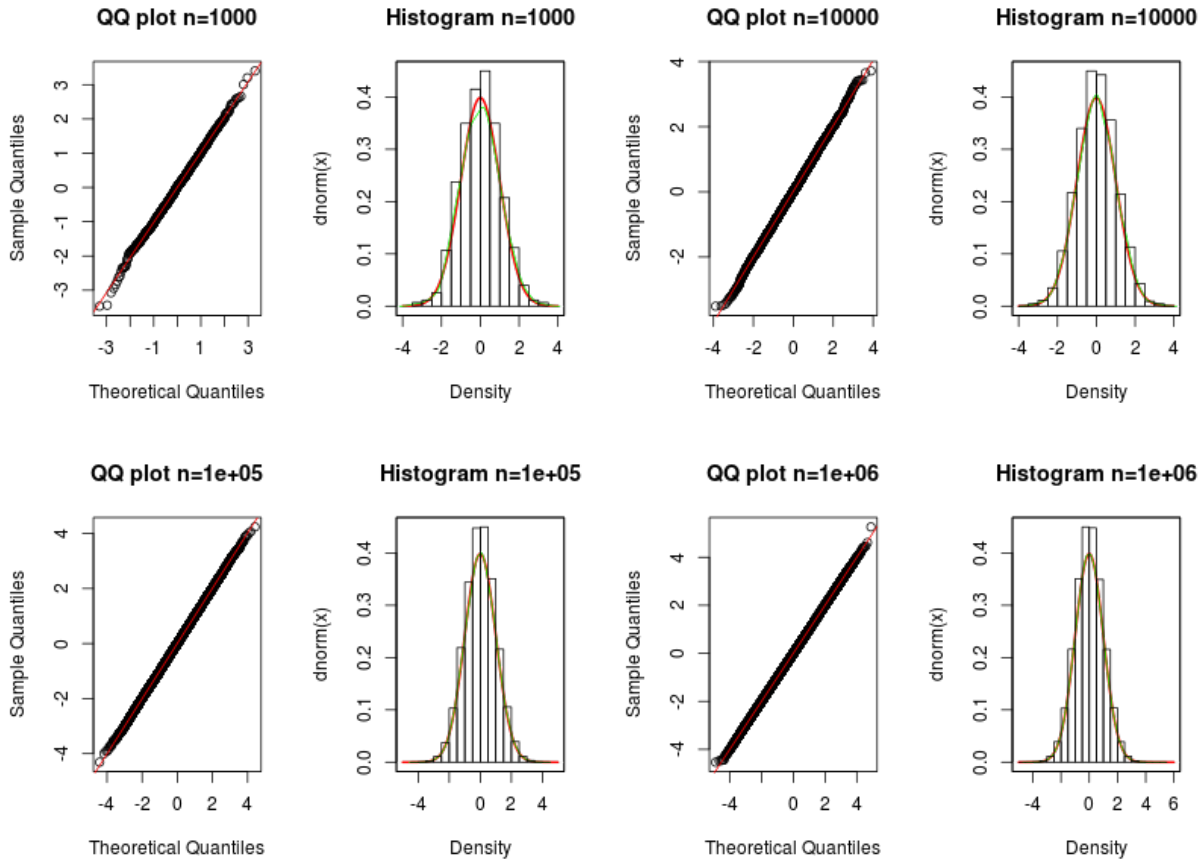
```
qq_plot_hist <- function(generator, nb, seed=20) {
   x <- generator(nb, seed)
   # QQ plot
   qqnorm(x)
   qqline(x, col="RED")
   # hist
   plot(dnorm, col="RED")
   par(new=T)
   plot(density(x), col="GREEN")
   par(new=T)
   hist(x)
}
```

Dans un premier temps, je me suis contenté d'un affichage pour 1 graine, de façon à ne pas surcharger inutilement la comparaison.

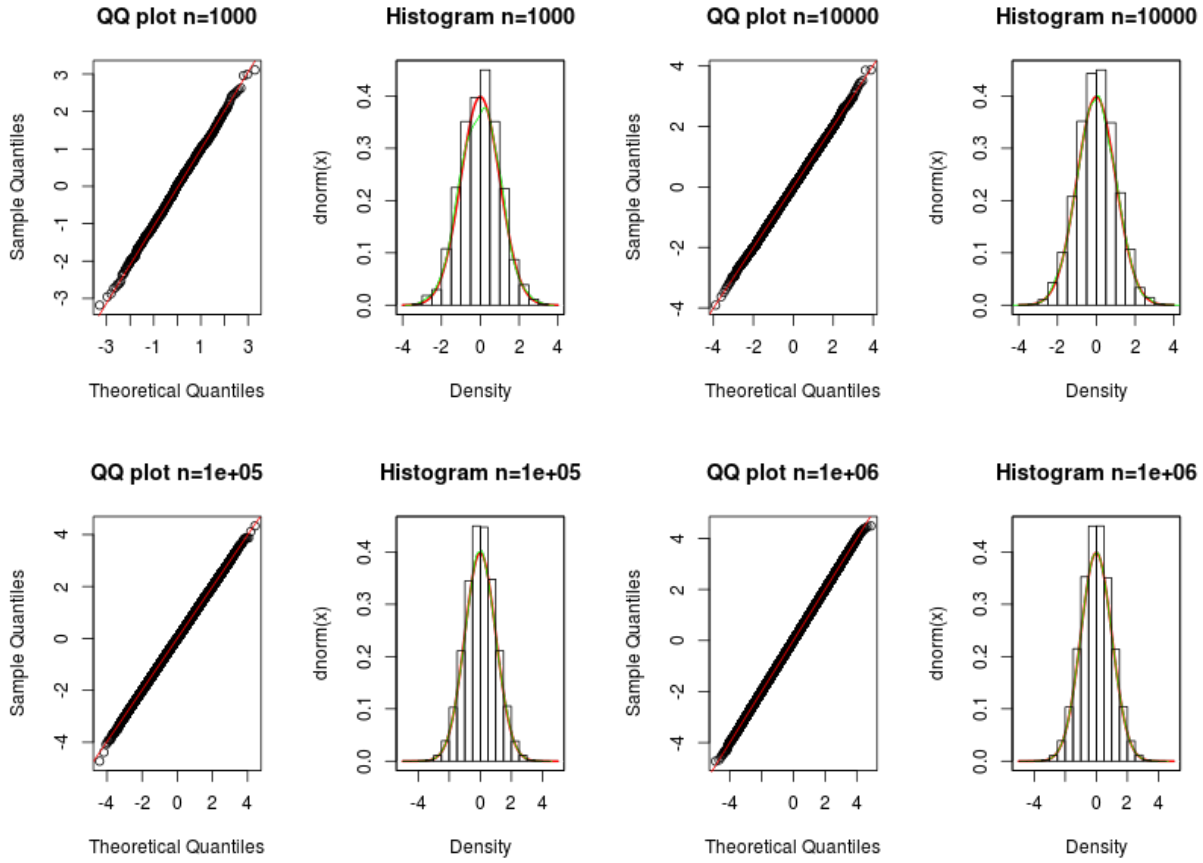
```
numbers <- c(1000,10000,100000,1000000)
generators <- c(rand_unif_to_norm, rand_rejet_cauchy,
                  rand_rejet_exp, rand_norm_box_muller, rand_norm_tcl)
sapply(generators, function(y) {sapply(numbers,
                           function(x) {qq_plot_hist(y, x, seeds[1])})})
```

On peut donc constater les résultats suivants :

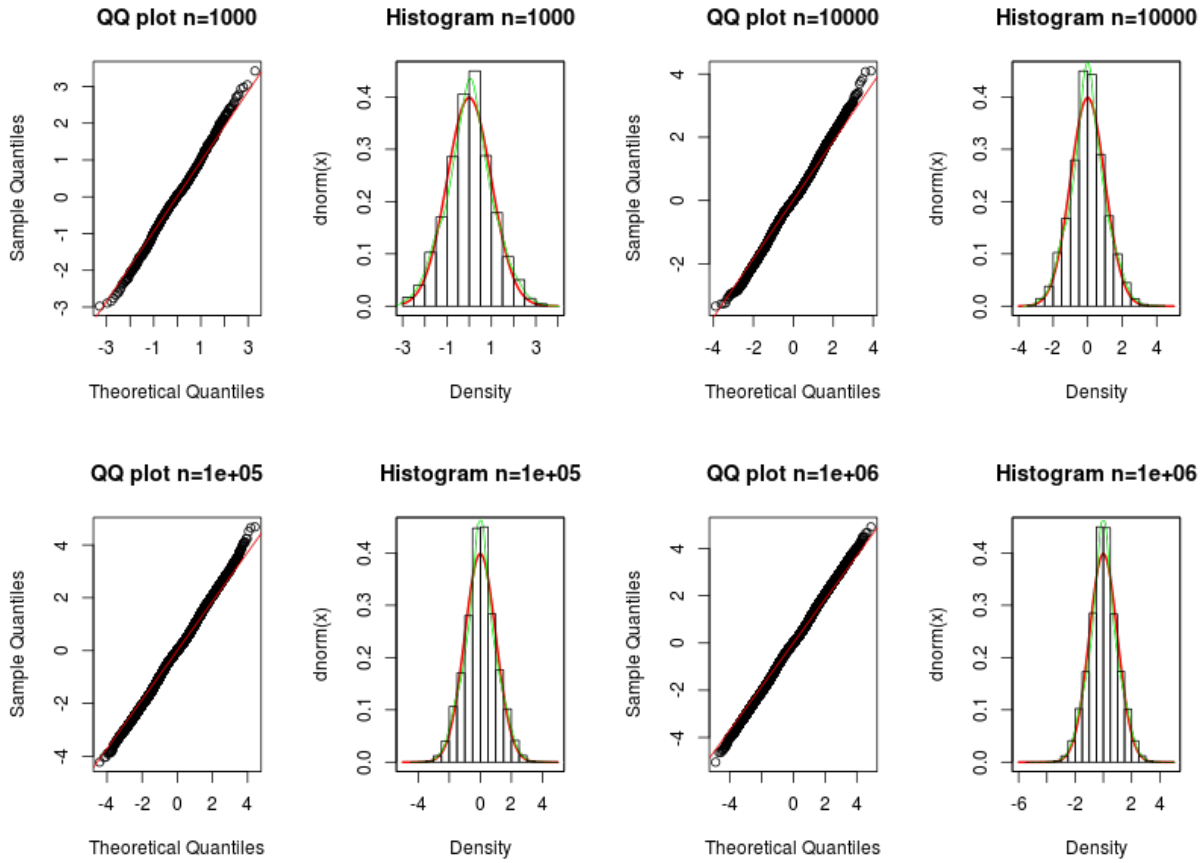
Pour le générateur **rand_unif_to_norm**



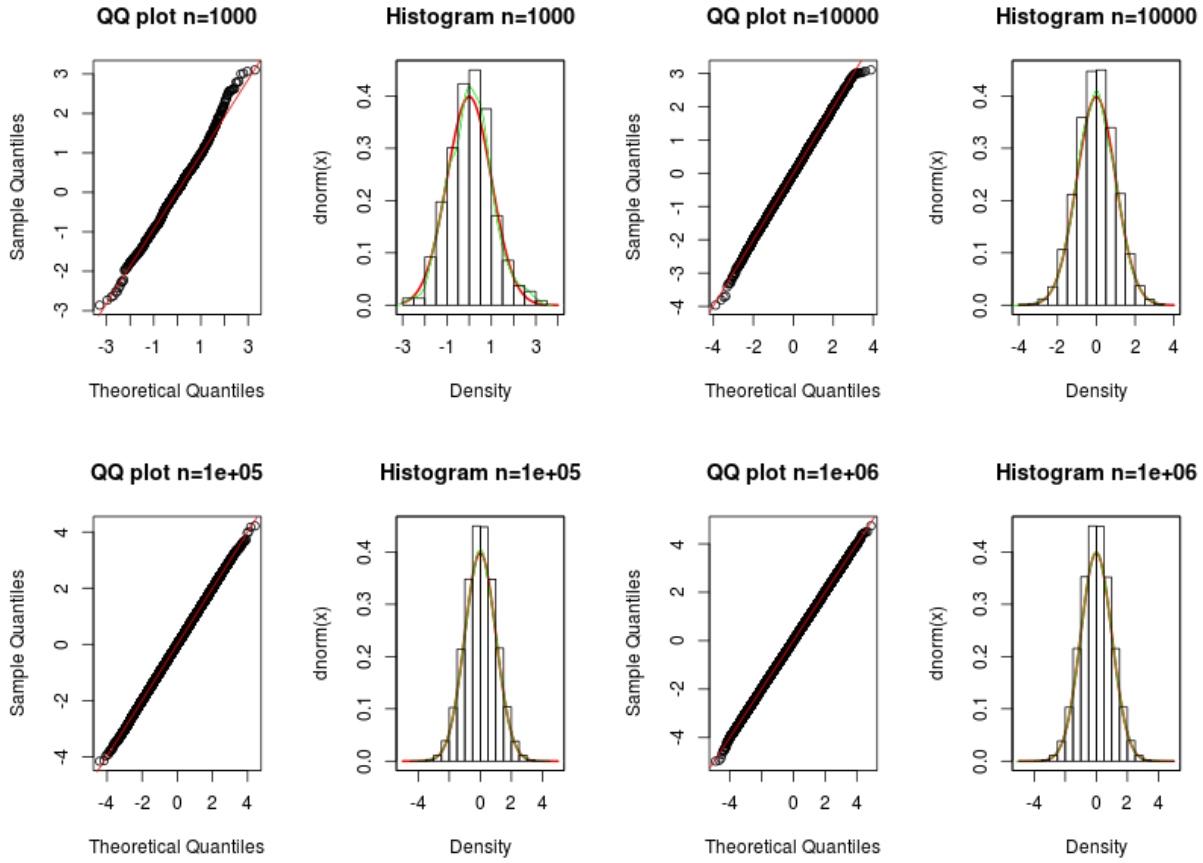
Pour le générateur **rand_rejet_cauchy**



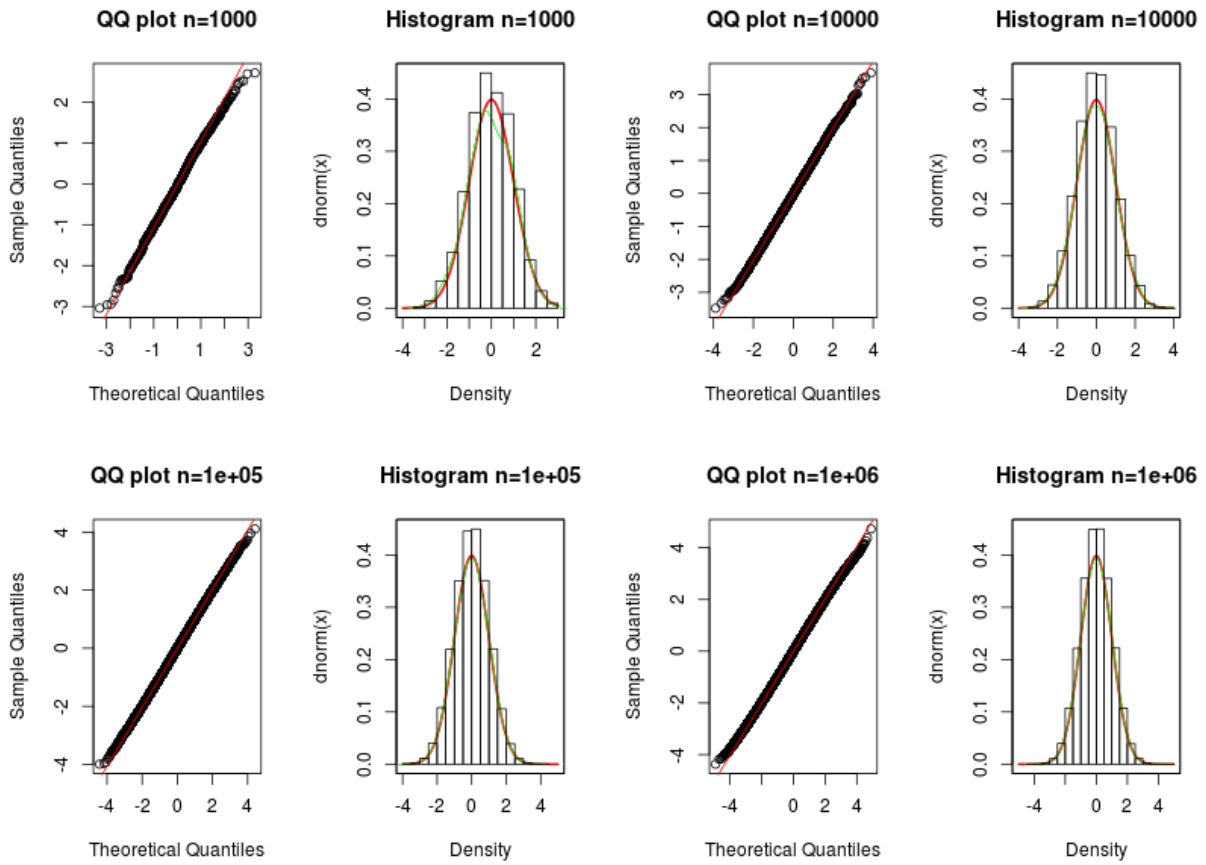
Pour le générateur **rand_rejet_exp**



Pour le générateur **rand_norm_box_muller**



Pour le générateur **rand_norm_tcl**



On ne manquera pas de mentionner l'extrême lenteur du générateur par la méthode de rejet.

2.3.3 Paramètres sur les échantillons

Comme dans le premier chapitre, on va s'intéresser aux valeurs remarquables de la loi $\mathcal{N}(0,1)$ à savoir :

- la moyenne qui a pour valeur théorique 0
- l'écart-type qui a pour valeur théorique 1
- skewness ayant pour valeur théorique 0
- kurtosis ayant pour valeur théorique 0
- les quantiles $(0, 25\%, 50\%, 75\%, 100\%)$ qui ont pour valeurs théoriques $(-Inf, -0.67449, 0, 0.67449, Inf)$

J'ai donc décidé de créer les tableaux de valeurs calculées sur les différents échantillons pour chaque tailles d'échantillon et ce grâce aux fonctions suivantes :

```
parametres_echantillon <- function(num_gen, nb, seed=20) {
   x <- generators[[num_gen]](nb, seed)
   param_test(x)
}

parametres_echantillon_all <- function(num_gen, seed=20) {
   options(digits=5)
   do.call(rbind, lapply(numbers, function(x) {parametres_echantillon(num_gen, x, seed)}
}
```

Ici aussi dans un soucis de lisibilité, on va se cantonner à une unique graine, les résultats sont donc les suivants :

rand_unif_to_norm :									
n	mean	sd	skewness	kurtosis	0%	25%	50%	75%	100%
1000	0.019623	1.04	0.05580	0.05807	-3.54	-0.686	0.033077	0.709	3.84
10000	-0.003066	1.00	0.01079	-0.08311	-3.54	-0.690	-0.006032	0.683	3.84
100000	0.003914	1.00	-0.00111	-0.01207	-4.20	-0.674	0.002757	0.685	4.57
1000000	-0.000307	1.00	-0.00356	-0.00726	-5.24	-0.675	-0.000343	0.675	4.58
rand_rejet_cauchy :									
théo	0	1	0	0	-Inf	-0.674	0	0.674	Inf
1000	-0.022189	1.05	0.01908	0.135100	-3.32	-0.724	0.000586	0.641	3.21
10000	0.003711	1.01	-0.02346	-0.040879	-4.13	-0.687	0.014420	0.693	3.91
100000	0.000168	1.00	0.00183	-0.019511	-4.13	-0.678	-0.001684	0.681	4.58
1000000	-0.001434	1.00	-0.00382	0.000763	-4.78	-0.676	-0.000387	0.673	4.79
rand_rejet_exp :									
n	mean	sd	skewness	kurtosis	0%	25%	50%	75%	100%
1000	-0.017860	1.04	0.011035	-0.033	-2.95	-0.682	0.010024	0.649	3.39
10000	-0.001332	1.02	-0.008461	0.159	-4.02	-0.638	0.000454	0.633	4.20
100000	0.003174	1.01	0.000765	0.167	-5.00	-0.624	0.001987	0.633	4.57
1000000	-0.000663	1.01	0.000556	0.171	-5.00	-0.624	-0.000538	0.622	4.66
rand_norm_box_muller :									
théo	0	1	0	0	-Inf	-0.674	0	0.674	Inf
1000	0.037611	1.000	0.07702	0.23530	-3.16	-0.625	0.016203	0.706	4.13
10000	0.013659	1.001	0.01529	-0.02863	-3.73	-0.662	0.012007	0.690	4.13
100000	-0.000331	0.999	0.00484	0.02879	-4.25	-0.671	0.001482	0.669	4.13
1000000	-0.000165	1.000	0.00239	0.00239	-4.65	-0.674	-0.000824	0.673	4.88
rand_norm_tcl :									
n	mean	sd	skewness	kurtosis	0%	25%	50%	75%	100%
1000	-0.026062	1.008	0.009585	-0.1091	-3.77	-0.720	-0.04655	0.676	2.94
10000	-0.009746	0.998	-0.016881	-0.1498	-3.77	-0.696	-0.00630	0.685	3.42
100000	-0.001396	0.998	-0.000912	-0.1053	-4.48	-0.684	-0.00153	0.681	4.09
1000000	-0.000607	1.000	-0.000281	-0.0996	-4.48	-0.682	-0.00140	0.682	4.38
théo	0	1	0	0	-Inf	-0.674	0	0.674	Inf

D'après ces résultats, on ne peut pas conclure à un réel avantage de l'une des méthodes par rapport aux autres puisque les valeurs sont toutes dans un interval relativement serré autour de leur valeur théorique. J'ai aussi essayé de mettre en évidence une différence en changeant de graine, mais cela n'a pas significativement abouti.

Il est donc tout à fait indispensable de poursuivre nos tests pour vérifier si les méthodes sont acceptables ou non.

2.3.4 Les tests de normalité

J'ai donc décidé d'utiliser les 5 tests suivants car tout en testant la normalité, ils testent des propriétés différentes, et sont ceux qui sont appliquables pour des échantillons assez grands. J'ai notamment du supprimer le test d'Agostino et de Shapiro pour cette raison.

- Test d'Anderson–Darling
- Test de Cramer-von Mises
- Test de Jarque Bera
- Test de Kolmogorov-Smirnov
- Test de Student

Dans un premier temps, je vais présenter brièvement sur quoi porte ces tests puis je vais les appliquer à mes 10 échantillons que j'ai généré pour chaque générateur.

Il est possible de retrouver des explications plus détaillées sur les différents tests aux liens suivants :
http://geai.univ-brest.fr/~carpenti/2007-2008/Documents-R/normalite.html
http://eric.univ-lyon2.fr/~ricco/cours/cours/Test_Normalite.pdf

Enfin, le test d'hypothèse étant le suivant :

$$H_0 : \text{"le générateur, génère des échantillons suivant la loi } \mathcal{N}(0,1)\text{"}$$
$$H_1 : \text{"les échantillons du générateur ne suivent pas la loi } \mathcal{N}(0,1)\text{"}$$

Comme j'ai décidé d'effectuer les tests sur 10 échantillons, pour pouvoir réellement conclure quant aux différentes p-values qui vont être tirées des tests, il va donc falloir les corriger. En effet, étant donné que dans le cadre de ce test, nous devons utiliser un multi test, j'ai donc utiliser la fonction **p.adjust** qui permet donc de corriger la p-value de chaque test.

En ajustant les p-values, il est donc maintenant possible de conclure au seuil de 5% sur le test posé ci-dessus sur les hypothèses.

2.3.4.1 Test d'Anderson–Darling

Ce test se base principalement sur la fonction de densité empirique [1] :

rand_unif_to_norm				
	1000	10000	100000	1000000
1	1.000	1.0000	1.000	0.701
2	1.000	1.0000	1.000	1.000
3	1.000	1.0000	1.000	1.000
4	1.000	0.0746	1.000	1.000
5	1.000	0.5172	1.000	1.000
6	1.000	1.0000	1.000	0.891
7	1.000	1.0000	1.000	1.000
8	1.000	0.8100	1.000	1.000
9	0.453	1.0000	0.452	1.000
10	1.000	1.0000	0.492	0.229

rand_rejet_cauchy				
	1000	10000	100000	1000000
1	1.000	1.000	0.449	1
2	0.393	0.150	1.000	1
3	1.000	1.000	1.000	1
4	1.000	1.000	1.000	1
5	0.921	1.000	1.000	1
6	0.997	0.425	1.000	1
7	1.000	1.000	0.915	1
8	1.000	1.000	1.000	1
9	0.162	1.000	1.000	1
10	1.000	1.000	1.000	1

1. http://en.wikipedia.org/wiki/Anderson-Darling_test

rand_rejet_exp				
	1000	10000	100000	1000000
1	1.12e-01	3.76e-24	1.07e-155	1
2	2.91e-03	2.32e-19	4.69e-159	1
3	2.34e-01	2.81e-18	1.28e-146	1
4	6.89e-02	2.81e-18	1.38e-149	1
5	5.00e-02	7.67e-19	5.68e-153	1
6	2.34e-01	1.71e-15	4.79e-159	1
7	9.42e-05	7.67e-19	1.17e-152	1
8	1.55e-02	2.77e-21	9.37e-152	1
9	1.12e-01	2.05e-17	1.97e-142	1
10	2.34e-01	2.26e-22	5.79e-164	1

rand_norm_box_muller				
	1000	10000	100000	1000000
1	1	1.000	1.000	1.000
2	1	1.000	1.000	0.582
3	1	1.000	1.000	0.428
4	1	1.000	1.000	1.000
5	1	1.000	1.000	1.000
6	1	1.000	1.000	1.000
7	1	1.000	1.000	1.000
8	1	1.000	1.000	0.135
9	1	0.786	1.000	1.000
10	1	1.000	0.919	0.428

rand_norm_tcl				
	1000	10000	100000	1000000
1	1.000	0.529	7.97e-04	3.31e-44
2	1.000	1.000	2.00e-05	6.58e-41
3	1.000	1.000	9.07e-05	6.13e-38
4	0.235	1.000	3.05e-05	4.21e-45
5	1.000	1.000	5.33e-03	1.59e-37
6	1.000	1.000	5.68e-05	5.30e-44
7	1.000	1.000	5.68e-05	9.05e-44
8	1.000	1.000	6.58e-05	8.60e-49
9	1.000	1.000	5.68e-05	2.11e-40
10	1.000	1.000	1.40e-05	4.01e-40

Au vu des différents résultats, on peut donc conclure que le test est pas significatif pour la méthode du théorème central limite ainsi que pour les "petits" jeux de données avec la méthode de rejet Exponentielle. Dans ces deux cas, il faut donc rejeter H_0, ces générateurs ne génèrent pas de distribution suivant une loi $\mathcal{N}(0,1)$.

2.3.4.2 Test de Cramer-von Mises

De même, ce test se base principalement sur la fonction de densité empirique.

rand_unif_to_norm				
	1000	10000	100000	1000000
1	1.000	1.000	1.000	0.488
2	1.000	1.000	1.000	1.000
3	1.000	1.000	1.000	1.000
4	1.000	0.112	1.000	1.000
5	1.000	0.837	1.000	1.000
6	1.000	0.971	1.000	0.644
7	1.000	1.000	1.000	1.000
8	1.000	0.837	1.000	1.000
9	0.227	1.000	0.386	1.000
10	1.000	1.000	0.829	0.255

rand_rejet_cauchy				
	1000	10000	100000	1000000
1	1.000	1.000	0.283	1
2	0.365	0.252	1.000	1
3	1.000	1.000	1.000	1
4	1.000	1.000	1.000	1
5	1.000	1.000	1.000	1
6	1.000	0.628	1.000	1
7	1.000	1.000	1.000	1
8	1.000	1.000	1.000	1
9	0.122	1.000	1.000	1
10	1.000	1.000	1.000	1

rand_rejet_exp				
	1000	10000	100000	1000000
1	0.067403	7.37e-09	7.37e-09	7.37e-09
2	0.002342	7.37e-09	7.37e-09	7.37e-09
3	0.233898	7.37e-09	7.37e-09	7.37e-09
4	0.050251	7.37e-09	7.37e-09	7.37e-09
5	0.034197	7.37e-09	7.37e-09	7.37e-09
6	0.233898	7.37e-09	7.37e-09	7.37e-09
7	0.000171	7.37e-09	7.37e-09	7.37e-09
8	0.008324	7.37e-09	7.37e-09	7.37e-09
9	0.067403	7.37e-09	7.37e-09	7.37e-09
10	0.225612	7.37e-09	7.37e-09	7.37e-09

rand_norm_box_muller				
	1000	10000	100000	1000000
1	1	1.000	0.773	1.000
2	1	1.000	1.000	0.472
3	1	1.000	1.000	0.660
4	1	1.000	1.000	1.000
5	1	1.000	1.000	1.000
6	1	1.000	1.000	1.000
7	1	1.000	1.000	1.000
8	1	1.000	1.000	0.114
9	1	0.534	1.000	1.000
10	1	1.000	0.759	0.306

rand_norm_tcl				
	1000	10000	100000	1000000
1	1.000	0.882	0.010741	7.37e-09
2	1.000	1.000	0.000201	7.37e-09
3	1.000	1.000	0.000739	7.37e-09
4	0.259	1.000	0.000113	7.37e-09
5	1.000	1.000	0.018839	7.37e-09
6	1.000	1.000	0.000299	7.37e-09
7	1.000	1.000	0.000739	7.37e-09
8	1.000	1.000	0.000739	7.37e-09
9	1.000	1.000	0.000739	7.37e-09
10	1.000	1.000	0.000113	7.37e-09

Avec le test de Cramer-von Mises on peut en conclure la même chose que le précédent.

2.3.4.3 Test de Jarque Bera

Ce test se base sur le test des moments d'ordre 3 et 4, en comparant les skewness et kurtosis avec leurs valeurs théoriques [2] :

rand_unif_to_norm				
	1000	10000	100000	1000000
1	1	1.000	1	1
2	1	1.000	1	1
3	1	1.000	1	1
4	1	0.328	1	1
5	1	0.561	1	1
6	1	1.000	1	1
7	1	1.000	1	1
8	1	1.000	1	1
9	1	1.000	1	1
10	1	1.000	1	1

rand_rejet_cauchy				
	1000	10000	100000	1000000
1	1.000	1.000	1.000	1.000
2	0.411	0.505	1.000	1.000
3	1.000	1.000	1.000	1.000
4	1.000	1.000	1.000	1.000
5	0.872	1.000	0.414	0.524
6	0.872	0.653	1.000	1.000
7	1.000	1.000	0.360	1.000
8	1.000	0.874	1.000	1.000
9	0.872	0.879	1.000	1.000
10	1.000	1.000	1.000	1.000

2. http://fr.wikipedia.org/wiki/Test_de_Jarque_Bera

rand_rejet_exp				
	1000	10000	100000	1000000
1	1.0000	0.000221	0	0
2	1.0000	0.011067	0	0
3	1.0000	0.143289	0	0
4	1.0000	0.143289	0	0
5	1.0000	0.004686	0	0
6	1.0000	0.053456	0	0
7	0.0572	0.049043	0	0
8	1.0000	0.000016	0	0
9	1.0000	0.053456	0	0
10	1.0000	0.004686	0	0

rand_norm_box_muller				
	1000	10000	100000	1000000
1	1	1	1.000	1.000
2	1	1	1.000	1.000
3	1	1	1.000	0.246
4	1	1	1.000	1.000
5	1	1	1.000	1.000
6	1	1	1.000	1.000
7	1	1	1.000	1.000
8	1	1	1.000	0.118
9	1	1	1.000	1.000
10	1	1	0.558	1.000

rand_norm_tcl				
	1000	10000	100000	1000000
1	1.000	0.391	5.12e-09	0
2	1.000	0.650	6.44e-09	0
3	1.000	1.000	1.22e-06	0
4	0.409	0.109	1.22e-06	0
5	1.000	1.000	5.25e-06	0
6	1.000	1.000	5.12e-09	0
7	1.000	0.725	6.85e-08	0
8	0.442	1.000	6.08e-12	0
9	1.000	1.000	5.53e-10	0
10	1.000	1.000	3.61e-07	0

Dans ce cas on remarque que même avec de grands échantillons, le test est significatif pour tous, c'est-à-dire que l'on rejette l'hypothèse de normalité.

2.3.4.4 Test de Kolmogorov-Smirnov

Ce test se base sur la fonction de répartition empirique, qu'il compare grâce à l'argument **"pnorm"** [3] :

rand_unif_to_norm				
	1000	10000	100000	1000000
1	1	1	1	1
2	1	1	1	1
3	1	1	1	1
4	1	1	1	1
5	1	1	1	1
6	1	1	1	1
7	1	1	1	1
8	1	1	1	1
9	1	1	1	1
10	1	1	1	1

rand_rejet_cauchy				
	1000	10000	100000	1000000
1	1	1	1	1
2	1	1	1	1
3	1	1	1	1
4	1	1	1	1
5	1	1	1	1
6	1	1	1	1
7	1	1	1	1
8	1	1	1	1
9	1	1	1	1
10	1	1	1	1

3. http://fr.wikipedia.org/wiki/Test_de_Kolmogorov-Smirnov

rand_rejet_exp				
	1000	10000	100000	1000000
1	1.000	3.80e-05	0	0
2	1.000	2.13e-04	0	0
3	1.000	1.56e-04	0	0
4	1.000	1.49e-05	0	0
5	1.000	1.46e-04	0	0
6	1.000	2.13e-04	0	0
7	0.161	1.56e-04	0	0
8	1.000	1.83e-04	0	0
9	1.000	1.56e-04	0	0
10	1.000	1.37e-05	0	0

rand_norm_box_muller				
	1000	10000	100000	1000000
1	1	1	1	1
2	1	1	1	1
3	1	1	1	1
4	1	1	1	1
5	1	1	1	1
6	1	1	1	1
7	1	1	1	1
8	1	1	1	1
9	1	1	1	1
10	1	1	1	1

rand_norm_tcl				
	1000	10000	100000	1000000
1	1	1	0.929	3.49e-06
2	1	1	0.929	5.29e-06
3	1	1	0.929	1.21e-05
4	1	1	0.464	1.66e-06
5	1	1	0.929	1.21e-05
6	1	1	0.929	5.29e-06
7	1	1	0.929	1.80e-06
8	1	1	0.929	2.24e-06
9	1	1	0.929	1.21e-05
10	1	1	0.464	4.45e-06

De même que ces précédesseurs, le test est significatif pour les méthodes de rejet sous Forme Exponentielle et par la méthode du théorème centrale limite.
On peut aussi noter que les autres p-values sont très très proches de 1.

2.3.4.5 Test de Student

Ce dernier test est basé directement sur les paramètres de la loi $\mathcal{N}(0,1)$ [4] :

rand_unif_to_norm				
	1000	10000	100000	1000000
1	1.000	1.000	1.0000	1
2	1.000	1.000	1.0000	1
3	1.000	1.000	1.0000	1
4	1.000	1.000	1.0000	1
5	1.000	1.000	0.0603	1
6	1.000	1.000	1.0000	1
7	1.000	0.671	1.0000	1
8	1.000	1.000	1.0000	1
9	1.000	1.000	1.0000	1
10	0.364	1.000	1.0000	1

rand_rejet_cauchy				
	1000	10000	100000	1000000
1	1.000	1	1.000	1.000
2	1.000	1	0.414	1.000
3	1.000	1	1.000	1.000
4	1.000	1	1.000	1.000
5	1.000	1	0.838	1.000
6	1.000	1	1.000	1.000
7	0.326	1	1.000	0.410
8	1.000	1	1.000	1.000
9	1.000	1	0.427	0.796
10	0.170	1	1.000	1.000

4. http://fr.wikipedia.org/wiki/Test_de_Student

rand_rejet_exp				
	1000	10000	100000	1000000
1	1.000	1.00	1.0000	1
2	1.000	1.00	1.0000	1
3	1.000	1.00	1.0000	1
4	1.000	1.00	1.0000	1
5	0.452	1.00	0.4167	1
6	1.000	1.00	1.0000	1
7	1.000	0.44	1.0000	1
8	1.000	1.00	1.0000	1
9	1.000	1.00	1.0000	1
10	0.767	1.00	0.0332	1

rand_norm_box_muller				
	1000	10000	100000	1000000
1	1.000	1.000	1.000	1.000
2	0.286	0.246	1.000	1.000
3	1.000	1.000	1.000	1.000
4	1.000	1.000	1.000	1.000
5	1.000	0.265	1.000	0.397
6	1.000	1.000	1.000	1.000
7	1.000	1.000	1.000	1.000
8	1.000	1.000	0.305	1.000
9	1.000	1.000	1.000	0.661
10	1.000	1.000	1.000	1.000

rand_norm_tcl				
	1000	10000	100000	1000000
1	1.0000	1	1	1
2	1.0000	1	1	1
3	1.0000	1	1	1
4	1.0000	1	1	1
5	1.0000	1	1	1
6	1.0000	1	1	1
7	1.0000	1	1	1
8	1.0000	1	1	1
9	1.0000	1	1	1
10	0.0234	1	1	1

Ici, le test est non significatif dans tous les cas, on peut donc se demander si la puissance de ce test est suffisante, car il a l'air de laisser passer "trop" facilement les tests.

2.4 Conclusion

Au vu des différents tests de normalité effectués, on peut donc conclure que la méthode la plus fiable et aussi l'une des plus rapide est la méthode qui utilise $erfinv$. A l'inverse, la méthode de rejet avec la Forme Exponentielle est très coûteuse et peu efficace, il semble donc qu'il vaille mieux ne pas l'utiliser, de même pour la méthode du théorème central limite avec un échantillon de 12.

3 Méthode de Monte-Carlo

3.1 Description de la méthode de Monte-Carlo

Pour calculer des intégrales de la forme :

$$I = \int_a^b f(x).dx$$

la méthode de Monte-Carlo consiste à utiliser la forme :

$$I_1 = \mathbb{E}(f(X))$$

avec $X \sim> \mathcal{U}(a,b)$. De plus, on a :

$$I_2 = \frac{1}{n}\sum_{i=1}^{n} f(x_i)$$

par la Loi Forte des Grands Nombres $I_2 \longrightarrow I_1 = I$.
Or $\mathbb{E}(f(X)) = \int_a^b f(x).f_X(x).dx$ avec f_X la densité de la loi suivie par X.
Comme $X \sim> \mathcal{U}(a,b)$ on a donc $f_X(x) = \frac{1}{b-a}.\mathbb{1}_{[a,b]}(x)$ d'ou :
$I_2 = \frac{1}{b-a}\int_a^b f(x).dx$
On peut donc en déduire que :

$$\frac{(b-a)}{n}\sum_{i=1}^{n} f(x_i) \longrightarrow \int_a^b f(x).dx$$

3.2 Calcul d'intégrale simple par la méthode MC

3.2.1 Introduction

On va donc traiter un exemple, à savoir le calcul de l'intégrale suivante :

$$I = \int_{0.1}^{0.9} \frac{x}{2\sqrt{1-x^2}} e^{arcsin(x)} dx$$

On va alors effectuer M expériences de Monte-Carlo de N tirages et estimer la valeur de l'intégrale pour chacune des expériences. On admettra que l'intégrale vaut la moyenne des M expériences.

Comme expliqué dans le cours, l'intervalle de confiance de la moyenne $\overline{x}_M$ permettant d'estimer l'intégrale I de seuil α est définie par :

$$IC_\alpha(\overline{x}_M) =]\overline{x}_M - u_\alpha.\frac{\sigma}{\sqrt{M}}; \overline{x}_M + u_\alpha.\frac{\sigma}{\sqrt{M}}[$$

σ^2 étant inconnu, on va donc l'estimé par la variance empirique S_M^2, on a donc :

$$IC_\alpha(\overline{x}_M) =]\overline{x}_M - u_\alpha.\frac{S_M}{\sqrt{M}}; \overline{x}_M + u_\alpha.\frac{S_M}{\sqrt{M}}[$$

3.2.2 Mise en place sur R

Pour estimer la valeur de l'intégrale, j'ai donc créé la fonction qui va appliquer la méthode de Monte-Carlo :

```
mmc <- function(f, a, b, M=1000, N=1000, conf.level=0.975) {
   echant_unif <- sapply(rep(M,N), runif) # génération des échantillons
                                          # uniformes [0,1]
   # echant_unif[i,] : 1 échantillon aléatoire

   echant_ab <- echant_unif*(b-a) + a # conversion en [a,b]

   exp_mc <- sapply(1:M, function(x) { (b-a)*mean(f(echant_ab[x,])) })
                                       # calcul de l'intégrale

   err <- 2*qnorm(conf.level)*sd(exp_mc)/sqrt(M) # estimation de l'erreur

   RVAL <- list(estim = mean(exp_mc), error = err,
                IC = c(mean(exp_mc) - err/2, mean(exp_mc) + err/2))
    return(RVAL) # retour de données
}
```

Étant donné que je souhaite étudier l'influence de N et M sur l'erreur, j'ai donc modifié cette fonction dans le but de générer plus rapidement nos échantillons. Dans la fonction précédente, tous les échantillons étaient stockés en mémoires avant d'être étudié, en utilisant une boucle plutôt qu'une fonction apply (l'inverse logique du logiciel R) j'ai donc gagné en performance grâce à cette fonction :

```
mmc_modif <- function(f, a, b, M=1000, N=1000, conf.level=0.975){
   exp_mc <- rep(0,M)
   for ( m in 1:M)
   {
      echant_ab <- runif(N)*(b-a) + a # tirage d'un échantillon
      exp_mc[m] <- (b-a)*mean(f(echant_ab)) # calcul direct dessus
   }
   err <- 2*qnorm(conf.level)*sd(exp_mc)/sqrt(M) # estimation de l'erreur

   RVAL <- list(estim = mean(exp_mc), error = err,
                IC = c(mean(exp_mc) - err/2, mean(exp_mc) + err/2))
    return(RVAL) # retour de données
}
```

Il est maintenant possible de calculer la valeur de cette intégrale grâce à cette fonction, pour ce faire, on va directement effectuer les applications avec plusieurs valeurs de N et de M dont voici le résultat :

		M			
		10	100	1000	10000
N	10	0.6011852	0.6314547	0.6022699	0.6013445
	100	0.5975692	0.6145209	0.6016366	0.6022152
	1000	0.6063407	0.6029724	0.6015135	0.6027958
	10000	0.6028674	0.6034042	0.6031141	0.6028653
	100000	0.6027166	0.6030402	0.6028465	0.6028349

On remarque alors une convergence pour les grandes valeurs de M et N, on peut alors en conclure que :

$$I \approx 0.6028$$

3.2.3 Estimation de l'erreur commise

A l'aide de la fonction précédente, on peut directement récupérer l'erreur effectuée par rapport à notre estimation. Voici alors les résultats :

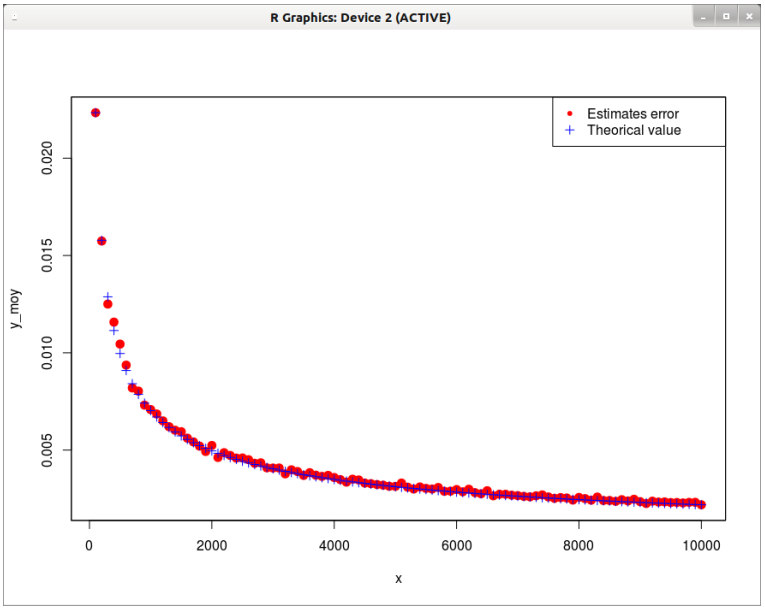
		M			
		10	100	1000	10000
N	10	0.148605986	0.0795900551	0.0226339832	7.105627e-03
	100	0.052509333	0.0243307459	0.0070234275	2.244421e-03
	1000	0.016821414	0.0075485917	0.0022115192	7.094758e-04
	10000	0.008893501	0.0021692154	0.0007243925	2.236932e-04
	100000	0.002280973	0.0006754426	0.0002194146	7.151626e-05

Ces estimations d'erreurs commises, peuvent nous permettre de connaitre si besoin les différents intervalles de confiances autour des valeurs estimées.

Il peut maintenant être intéressant d'estimer l'erreur en fonction du nombre de tirages. Pour ce faire, j'ai créé la fonction suivante qui utilise notamment la fonction précédente nous permettant d'appliquer la Méthode de Monte-Carlo.

```
estim_error <- function(n) {
   mmc_modif(f, a=0.1, b=0.9, M=100, N=n, conf.level=0.975)$error
}
```

Pour accélérer l'étude, j'ai décidé d'effectuer 10 fois cette procédure et d'en faire la moyenne de façon à ne pas dépendre totalement des échantillons aléatoire. Ceci avec une valeur de $M = 100$ car au delà, les temps d'executions sont relativement long lors de boucles. On obtient alors le graphique suivant :



On remarque très bien que la courbe décroît en $\frac{1}{\sqrt{N}}$, qui est affichée en bleu.

3.2.4 Comparaison avec une autre méthode

La valeur obtenue par la méthode de Monte-Carlo ci-dessus est 0.6028349, on peut maintenant la comparer avec la fonction **integrate** de R :

```
> integrate(f, 0.1, 0.9)
0.6028454 with absolute error = 8.354803e-07
```

On a donc l'intervalle de confiance suivant :

$$IC_{95\%,1}(I) =]0.602845; 0.6028458[$$

De plus, l'intervalle de confiance ayant l'erreur la plus faible, centré en 0.6028349 est le suivant :

$$IC_{95\%,2}(I) =]0.6027991; 0.6028707[$$

On remarque donc que le second est plus large, car l'erreur est plus grande, malgré tout $IC_{95\%,2} \subset IC_{95\%,1}$. De plus, la méthode utilisant la fonction **integrate** est beaucoup plus rapide que celle de Monte-Carlo, ce qui peut être dû à la génération des distributions aléatoires dans la méthode MC, car la fonction **integrate** utilise des subdivisions de l'intervalle ainsi que des fonctions externes codées en C, nettement plus rapide que celle programmées en R.

3.3 Nombre d'itérations en fonction de l'erreur

Cette section va traiter le cas ou l'on souhaite maitriser l'erreur. Elle peut avoir diverses applications, notament en sondage, pour connaître le bon nombre de personnes que l'on va devoir interroger pour avoir une erreur de 0.1 % sur l'ensemble de la population.

3.3.1 Point de vu théorique

Dans ce paragraphe, on cherche à calculer l'intégrale suivante :

$$\theta = \int_0^1 cos(\frac{\pi x}{2})dx$$

Étant donné que l'on a des fonctions usuelles, on peut dans ce cas calculer la valeur réelle :

$$\begin{aligned} \theta &= \left[\frac{2}{\pi} sin(\frac{\pi x}{2})\right]_0^1 \\ &= \frac{2}{\pi}\left(sin(\frac{\pi}{2}) - sin(0)\right) \\ &= \frac{2}{\pi} \\ &\approx 0.6366198 \end{aligned}$$

3.3.2 Mise en place sur R

Pour effectuer cette estimation, il suffit de réutiliser la fonction **mmc** et de la modifier légèrement, c'est-à-dire que l'on fixe un M maximal, ici 10000 puis on fait varier M à partir de 1 jusqu'à la valeur telle que l'erreur soit assez petite. On utiliser alors la fonction suivante :

```
mmc_iterate <- function(f, a, b, error=10^-2, N=1000, conf.level=0.975){
   M <- 2 # sd a besoin d'au moins 2 valeurs
   exp_mc <- runif(N)*(b-a) + a
   while ( M < 10000 ) {
      echant_ab <- runif(N)*(b-a) + a # tirage d'un nouvel échantillon
      exp_mc <- c(exp_mc, (b-a)*mean(f(echant_ab)))

      err <- 2*qnorm(conf.level)*sd(exp_mc)/sqrt(M)

      if ( err < error ) # si l'erreur est assez faible
      {
         break; # on s'arrète
      }

      M <- M + 1
   }

   RVAL <- list(estim = mean(exp_mc), error = err, M = M,
                IC = c(mean(exp_mc) - err/2, mean(exp_mc) + err/2))
    return(RVAL) # retour de données
}

f <- function(x) {
   y <- cos(pi*x/2)
   y
}

mmc_iterate(f, 0, 1)
```

3.3.3 Résultats

On obtient le résultat suivant :

```
> mmc_iterate(f, 0, 1)
$estim
[1] 0.6041835

$error
[1] 0.009998092

$IC
[1] 0.5991844 0.6091825

$M
[1] 3399
```

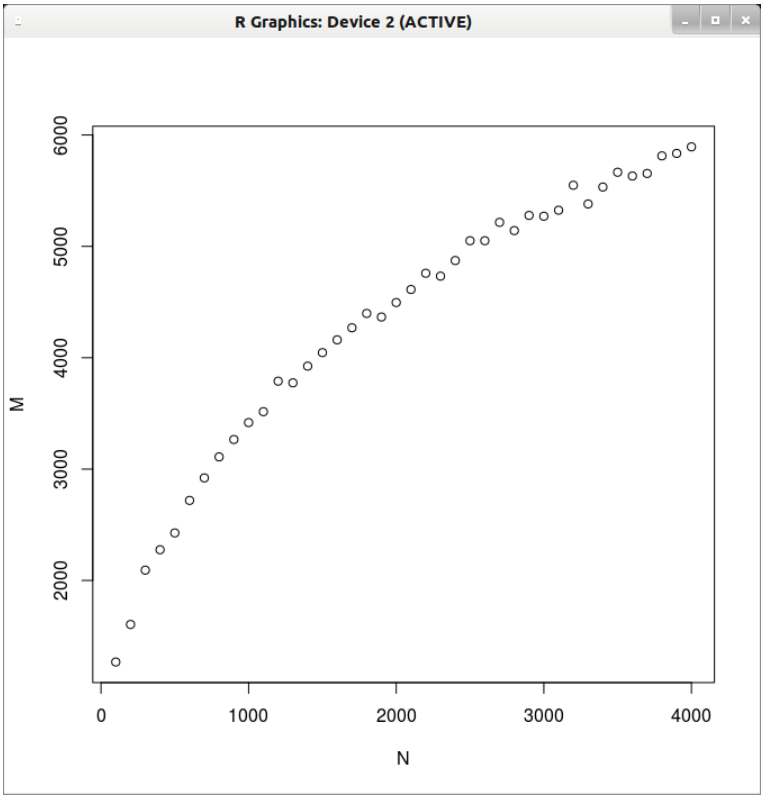
De plus, on remarque qu'avec une valeur plus faible de N, la méthode a besoin de moins d'itérations.

```
> mmc_iterate(f, 0, 1, N=100)
$estim
[1] 0.6271044

$error
[1] 0.009998661

$IC
[1] 0.6221051 0.6321038

$M
[1] 1258
```

Avec $N = 1000$, il faut donc environ $M = 3500$ alors que pour $N = 100$, seul $M = 1200$ suffit. On peut alors tracer le de M en fonction de N :

```
M_x <- function(x) { mmc_iterate(f, 0, 1, N=x)$M }
N <- 1:40 * 100
M <- sapply(N, M_x)
plot(N, M)
```

On obtient alors le résulat suivant :



Pour une erreur constante, on remarque que le nombre de tirage de la méthode de Monte-Carlo augmente avec la taille de l'échantillon aléatoire, avec l'allure d'une racine carré. On peut alors se demander si en augmentant encore la taille de l'échantillon, on va tomber sur une valeur maximale, ce qui nous permettrait d'optimiser notre algorithme. Il sera donc possible de fixer une valeur maximale de M "intelligente".

3.3.4 Conclusion

On remarque sur les exemples donnés ci-dessus que la méthode de Monte-Carlo est plus précise avec une taille N d'échantillonage plus faible. Il semble aussi que cette méthode ne concerge pas vers la vraie valeur de l'intégrale, en effet, $0.6366198 \notin]0.5991844; 0.6091825[$ et $0.6366198 \notin]0.6221051; 0.6321038[$. On peut donc penser que certains paramètres peuvent être ajustés pour obtenir une meilleure performance. On pourrait notament modifier la méthode de génération de variables aléatoires ici **runif** utilisée sans fixation de graine. De plus, dans ce cas, cette méthode peut éventuellement mettre en défaut l'hypothèse de convergence du Théorème Central Limite, bien qu'avec un échantillon de taille $M > 1000$ ce théorème est largement vérifié dans bon nombre de domaines.