TP R 3: Lois de probabilité, simulations, optimisation

Cours de Programmation

Vittorio Perduca, Master 1 Mathématiques et Applications

UFR Math-Info, Université Paris Descartes, septembre 2020

Table des matières

1. Lois de probabilités

Pour une variable aléatoire X de loi `law` :

- `dlaw(x, paramètres)` calcule $P(X = x)$ si X est discrète et $f(x)$ si X est continue de densité f.
- `plaw(q,paramètres)` calcule la fonction de répartition en q, c'est à dire $F(q) = P(X \leq q)$.
- `qlaw(p, paramètres)` donne le quantile q t.q. $p = P(X \leq q)$.
- `rlaw(m,parametres)` génère m nombres aléatoires selon la loi, i.e. un échantillon i.i.d. de taille m de X.

Consulter l'aide `?plaw`

La fonction `set.seed()` permet de spécifier l'amorce du générateur de nombre aléatoire, ce qui se révèle utile quand on veut répéter une simulation à l'identique :

```
runif(1) # tirage uniforme dans [0,1]
```

```
## [1] 0.5714351
```

```
set.seed(42); runif(1)
```

```
## [1] 0.914806
```

```
set.seed(42); runif(1)
```

```
## [1] 0.914806
```

1.1 Lois discrètes

- Loi binomiale $\mathcal{B}(n,p)$: `rbinom(m,size=n,prob=p)`. Cette loi a $n+1$ modalités distinctes : $0,1,\ldots,n$.
- Loi de Bernoulli $\mathcal{B}(p)$: `rbinom(m,size=1,prob=p)`
- Loi géométrique partant de zéro $\mathcal{G}(p)$ (nombre d'échecs avant succès) : `rgeom(n, prob=p)`
- Loi de Poisson $\mathcal{P}(\lambda)$: `rpois(m,lambda=lambda)`

1.2 Lois continues

- Loi uniforme $\mathcal{U}_{[a,b]}$: `runif(m,min=a,max=b)`
- Loi normale $\mathcal{N}(\mu,\sigma^2)$: `rnorm(m,mean=mu,sd=sigma)`
- Loi exponentielle $\mathcal{E}(\lambda)$: `rexp(m,rate=lambda)`
- Loi du Chi-deux à r degrés de liberté $\chi^2(r)$: `rchisq(m,df=r)`
- Loi de Student à r degrés de liberté : `rt(m,df=r)`

1.3 Echantilonnage d'une loi multinomiale

On utilise

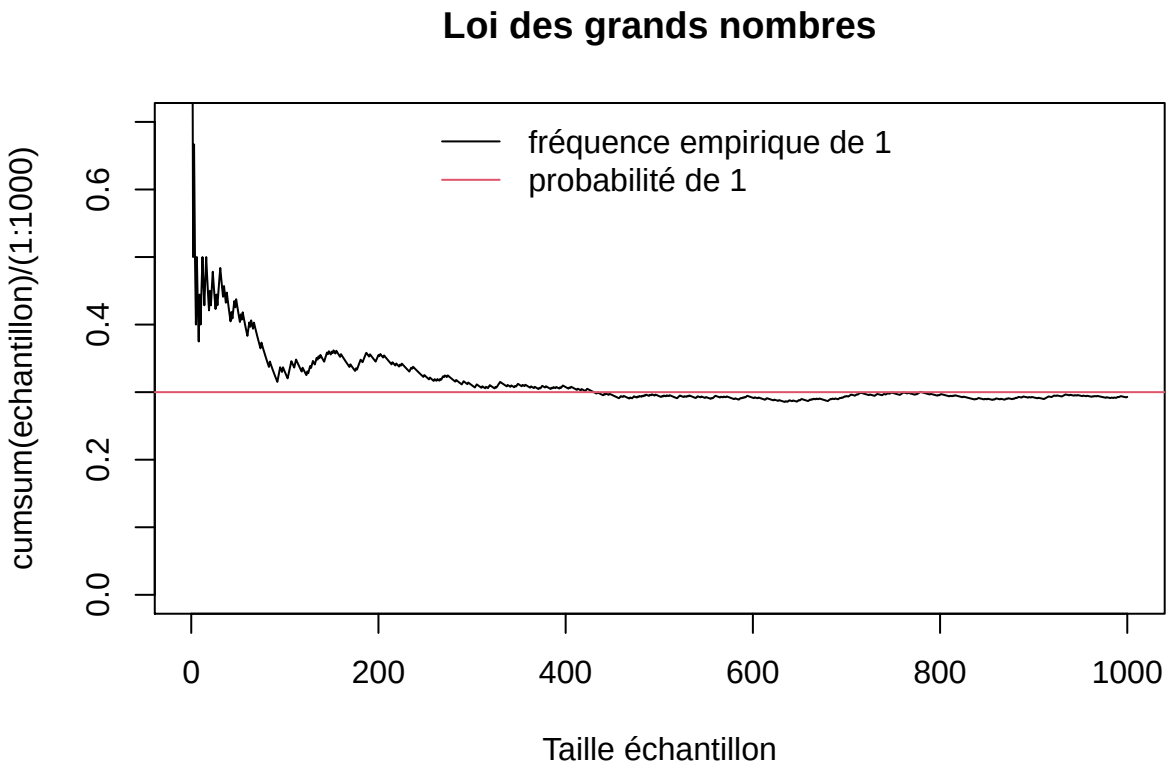
```
sample(x=urne,size=nombre boules à extraire,replace=TRUE/FALSE, prob = probabilité des
differents couleurs)
```

1.3.1 Illustration de la Loi des grands nombres

On tire n fois une pièce de monnaie biaisée t.q. la probabilité d'obtenir face est 0.3 et on calcule ensuite la proportion des faces obtenues.

```
echantillon=sample(x=c(0,1),size=1000,replace=T,prob=c(.7,.3))
plot(cumsum(echantillon)/(1:1000),
     type='l',
     main='Loi des grands nombres',
     ylim=c(0,0.7),
```

```
     xlab='Taille échantillon')
abline(h=.3,col=2)
legend(x='top',
       lty=1, col=1:2,
       legend=c('fréquence empirique de 1','probabilité de 1'),
       bty='n')
```



Remarques :

— Pour obtenir le nombre des faces en fonction de n on a utilisé la fonction `cumsum()` :

```
cumsum(c(0,0,1,0,1,1,0)) # nombre des 1s en fonction de n
```

```
## [1] 0 0 1 1 2 3 3
```

```
cumsum(c(0,0,1,0,1,1,0))/1:7 # fréquences de 1s en fonction de n
```

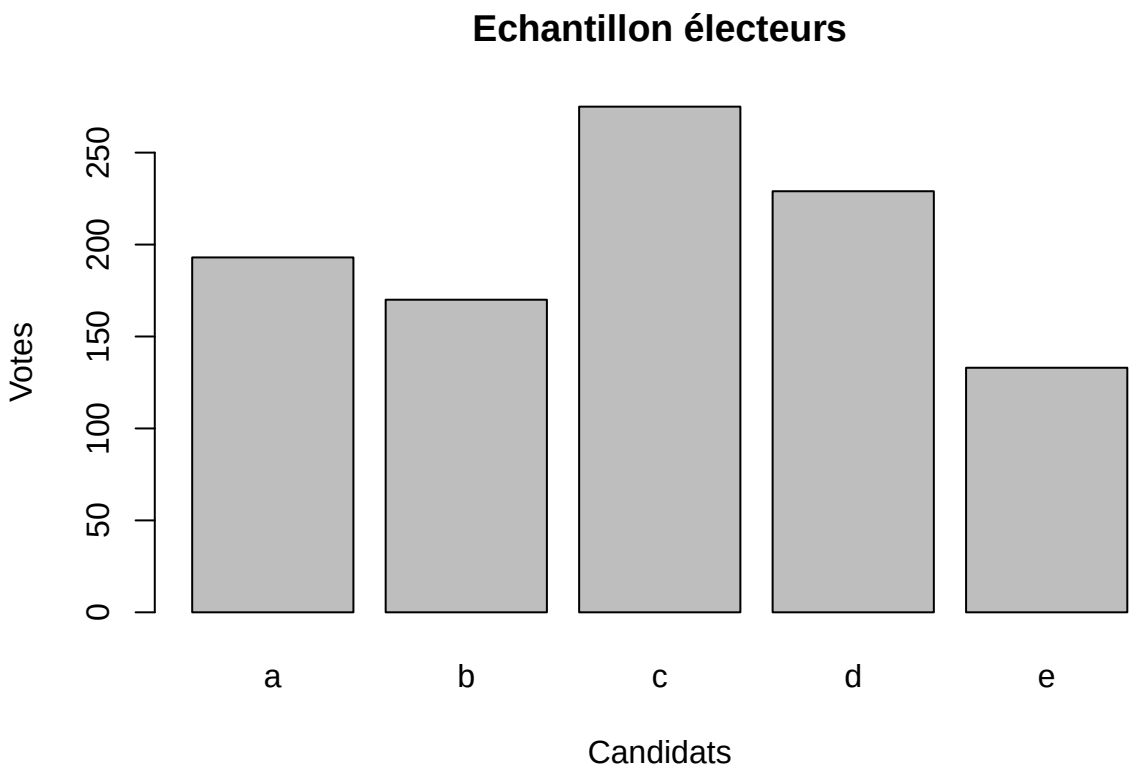
```
## [1] 0.0000000 0.0000000 0.3333333 0.2500000 0.4000000 0.5000000 0.4285714
```

— A la place de `sample()` on aurait pu utiliser `rbinom(n=1000,size=1,prob=.3)`

1.3.2 Echantillon loi multinomiale

```
candidats=letters[1:5]
echantillon=sample(x=candidats,
                   size=1000,replace=TRUE,prob=c(.2,.16,.26,.25,.13))
```

```r
barplot(summary(as.factor(echantillon)),
     ylab='Votes',
     xlab='Candidats',
     main='Echantillon électeurs')
```



2. La famille `_apply()`

Dans de nombreux cas il est possible de remplacer une boucle `for` par un appel beaucoup plus efficace (en terme de temps d'exécution et de mémoire utilisée) à une fonction de type `_apply()` ou `replicate()`.

2.1 La fonction `apply()`

On a déjà vu comment appliquer une fonction quelconque aux lignes ou aux colonnes d'une matrice :

`apply(X = matrice, MARGIN = 1 ou 2 pour lignes ou colonnes, FUN = fonction, ... = autres arguments de FUN)`

2.3 La fonction `tapply()`

Pour appliquer une fonction à tous les éléments d'un vecteur `X` selon les `levels` d'un facteur on utilise `tapply(X, INDEX = facteur, FUN, ... = autres arguments)`

2.4 Les fonctions `sapply()` et `lapply()`

Pour appliquer une fonction quelconque à tous les éléments d'une liste (ou d'un vecteur) `X` : `lapply(X, FUN = fonction, ... = autres arguments de FUN)` Le résultat est une **liste** avec le nombre d'éléments de `X` :

c'est comme si on exécutait `list(FUN(X[[1]], FUN(X[[2]],...)`.

```
(x=list(runif(7),runif(5)))
```

```
## [[1]]
## [1] 0.4384936 0.6999032 0.8890770 0.8341595 0.7344215 0.6469033 0.8428783
##
## [[2]]
## [1] 0.1596413 0.3637062 0.2750925 0.1247945 0.5656269
```

```
lapply(X=x,FUN=mean)
```

```
## [[1]]
## [1] 0.726548
##
## [[2]]
## [1] 0.2977723
```

Attention à bien spécifier tous les arguments nécessaires :

```
lapply(X=c(3,10), FUN=sample, x=1:10, replace=T)
```

```
## [[1]]
## [1] 10 10  1
##
## [[2]]
##  [1] 9 3 8 1 9 7 9 2 5 2
```

```
# Ici X = size (l'urne). C'est comme si on faisait
# sample(x=1:10,size=3,replace=T)
# et après sample(x=1:10,size=10,replace=T)
```

`sapply()` est comme `lapply()` mais rend en sortie des vecteurs ou des matrices (si cela est possible) :

```
sapply(x,mean)
```

```
## [1] 0.7265480 0.2977723
```

```
sapply(X=c(3,3),sample, x=1:10,replace=T)
```

```
##      [,1] [,2]
## [1,]    6    3
## [2,]    9    8
## [3,]    9   10
```

```
sapply(X=c(3,10),sample, x=1:10,replace=T) # impossible ici de rendre un vecteur ou une matrice
```

```
## [[1]]
## [1] 9 1 3
##
## [[2]]
##  [1] 8 6 4 7 6 4 5 4 4 1
```

`sapply()` permet donc de vectoriser une fonction qui n'est pas vectorielle.

2.5 La fonction `replicate()`

Pour exécuter n fois une expression `expr`, on utilise la fonction enveloppante de `sapply()` :

`replicate(n,expr)`

Cette fonction est très utile quand on fait des simulations.

3. Stratégies pour optimiser le temps d'exécution

3.1 Mesurer le temps d'exécution

La fonction `proc.time()` rend, entre autres choses, le temps écoulé à partir du début de la session `R`. On peut l'utiliser pour mesurer le temps d'exécution d'une expression :

```
(t0 <- proc.time()) # écoulé ou elapsed = temps écoulé en s à partir du début de la séssion
```

```
##    user  system elapsed
##   0.710   0.171   0.919
```

```
for (i in 1:50) median(runif(500))
t1 <- proc.time()
t1-t0
```

```
##    user  system elapsed
##   0.020   0.001   0.021
```

Pour mesurer le temps d'exécution, on préfère cependant utiliser la fonction `system.time({expr})` :

```
system.time({for(i in 1:1000000) mean(rnorm(1000))})
```

```
##    user  system elapsed
##  80.217   0.412  81.306
```

3.2 Comparaison de différentes méthodes dans un exemple

On reprend ici l'exemple présenté dans le site r-statistics.co.

On crée un data frame avec quatre colonnes numériques :

```
col1=runif(10^5,0,2)
col2=runif(10^5,0,2)
col3=rnorm(10^5,0,2)
col4=rnorm(10^5,0,2)
df=data.frame(col1,col2,col3,col4)
head(df)
```

```
##        col1      col2       col3        col4
## 1 0.3858930 0.3862958  0.3760222 -0.02721593
## 2 1.4860837 1.4954596  1.6697699  1.05442755
```

```
## 3 1.9682108 0.5232400 -2.7586654  0.34369920
## 4 1.3575402 1.1823967  1.7024358  0.68295732
## 5 1.4800629 0.1482839 -2.8510756  1.85935087
## 6 0.1948994 1.9132266  3.0710109  2.24713777
```

On veut ajouter une cinquième colonne avec les étiquettes `sup_a_trois` ou `inf_ a_trois` si la somme des quatre valeurs sur une même ligne est supérieure ou inférieure à trois.

3.2.1 Approche naïve par boucle :

```
s1 <- system.time({
  for(i in 1:10^5){
    if(df[i,1]+df[i,2]+df[i,3]+df[i,4]>3){
      df[i,5]='sup_a_trois'
    } else {
      df[i,5]='inf_a_trois'
    }
  }
})
s1
```

```
##    user  system elapsed
##  66.217   3.828  70.334
```

3.2.2 Pré-allocation :

Pour ne pas avoir à incrémenter la taille de `df` à chaque itération dans la boucle, ce qui est très peu efficace [1], il convient de initialiser les variables de sortie avant la boucle :

```
s2 <- system.time({
  output=character(length=10^5) # on crée un vecteur vide de caractères de longueur donnée
  for(i in 1:10^5){
    if(df[i,1]+df[i,2]+df[i,3]+df[i,4]>3){
      output[i]='sup_a_trois'
    } else {
      output[i]='inf_a_trois'
    }
  }
  df$output=output
})
s2
```

```
##    user  system elapsed
##   4.998   0.078   5.412
```

1. V. Goulet appelle cela le **Syndrome de la plaque à biscuit :** *pour ranger des biscuits on prend un premier biscuit et on le range dans un plat ne pouvant contenir qu'un seul biscuit. Arrivé au second biscuit, on constate que le contenant n'est pas assez grand, alors on sort un plat pouvant contenir deux biscuits, on change le premier biscuit de plat et on y range aussi le second biscuit. Arrivé au troisième biscuit, le petit manège recommence, et ainsi de suite jusqu'à ce que le plateau de biscuits soit épuisé.*

3.2.3 Pré-allocation et `ifelse()` :

En général, la fonction `ifelse()` permet de faire du traitement conditionnel de façon plus efficace que `if(){}else{}`. L'utilisation est très simple :

```
(x <- runif(1))
```

```
## [1] 0.7047977
```

```
ifelse(x<0.5, 'gagné', 'perdu')
```

```
## [1] "perdu"
```

Dans notre exemple :

```
s3 <- system.time({
  output=character(length=10^5)
  for(i in 1:10^5){
    output[i] = ifelse(df[i,1]+df[i,2]+df[i,3]+df[i,4]>3,'sup_a_trois','inf_a_trois')
  }
  df$output=output
})
s3
```

```
##    user  system elapsed
##   5.150   0.065   5.472
```

3.2.4 En utilisant les fonctions `_apply()`

```
ma_fonction=function(x){
  ifelse(x[1]+x[2]+x[3]+x[4]>3,'sup_a_trois','inf_a_trois')
}

s4 <- system.time({
  df$output=apply(df[,1:4],1,ma_fonction)
  # ma_fonction est appliquée à toutes les lignes de df
})
s4
```

```
##    user  system elapsed
##   1.116   0.022   1.144
```

3.2.5 En utilisant `which()`

En général, les boucles où à chaque itération il faut vérifier une condition logique sont très lentes. Il faut donc essayer de sortir le traitement conditionnel des boucles.

```
s5 <- system.time({
  lignes_sup=which(df[,1]+df[,2]+df[,3]+df[,4]>3)
  df$output=('inf_a_trois')
  df$output[lignes_sup]='sup_a_trois'
})
s5
```

```
##    user  system elapsed
##   0.001   0.000   0.002
```

En résumant voici le temps d'exécution de différentes approches :

Table 1: Temps d'exécution, s

	Boucle naive	Pré-alloc	Pré-alloc + ifelse()	apply()	which()
elapsed	70.334	5.412	5.472	1.144	0.002

4. Exercices

4.1 Introduction aux tests

Dans cet exercice on utilisera `R` pour calculer les valeurs de la fonction de répartition et les quantiles d'une loi normale.

On considère une maladie faisant chuter la numération des globules blancs. On cherche à établir un test sur lequel s'appuyer pour effectuer un diagnostic. On suppose que la numération chez les personnes saines suit une loi normale de moyenne 8000 et d'écart-type 1000. Le but du test est alors de choisir, au vu d'une valeur mesurée X_{obs}, entre les deux hypothèses suivantes :

— H_0 : la numération est normale
— H_1 : la numération est trop faible

H_0 est appelée l'*hypothèse nulle* et H_1 l'*hypothèse alternative*.

1. Supposons qu'on fixe à 6500 le seuil en-dessous duquel le nombre de globules blancs est jugé insuffisant. Quel est la probabilité de déclarer malade une personne saine ?
2. Quel est le seuil S qu'il faut fixer si on accepte de traiter inutilement $\alpha = 5\%$ des gens sains ? Cette probabilité est appelée *risque de première espèce* et $1 - \alpha$ est appelée la *confiance* du test.
3. Supposons que la numération chez les personnes atteintes de la maladie suit une loi normale de moyenne 5500 et d'écart-type 1000. Quelle est la probabilité β de déclarer saine une personne malade avec le seuil de 6500 ? Avec le seuil S ? Cette probabilité β est appelée *risque de deuxième espèce* et $1 - \beta$ est la *puissance* du test.
4. Tracer β en fonction de α pour des valeurs de α allant de 0.01 à .5 par pas de .01. Que constate-t-on ?
5. On décide de mettre en place le test avec un risque de première espèce de 5%, c'est-à-dire avec le seuil S. On suppose que la loi de la numération chez les personnes malades est toujours d'écart-type 1000 mais qu'on ne connaît pas son espérance, qu'on note μ. Tracer la puissance du test en fonction de μ. Cette courbe est appelée *courbe de puissance du test.*
6. La *probabilité critique* (ou *p-valeur*) d'une mesure est la probabilité d'observer une mesure au moins aussi basse sous l'hypothèse que la personne est saine :

$$p_c = \mathbb{P}(X \leq X_{obs} | X \sim \mathcal{N}(8000, 1000^2))$$

Déterminer les probabilités critiques des mesures suivantes allant de 5000 à 10000 par paliers de 500. Déterminer également celle correspondant à une mesure égale à S. En déduire une façon de conclure quand on fait un test à l'aide de la probabilité critique. Quel avantage y voyez-vous par rapport à la comparaison avec un seuil ?

4.2 Simulations selon la loi de Pareto et manipulations

On reprend l'exercice 6.3 du livre de V. Goulet (pages 127-128). La fonction de densité de probabilité et la fonction de répartition de la loi de Pareto de paramètres α et λ, sont respectivement

$$f(x) = \frac{\alpha \lambda^\alpha}{(x + \lambda)^{\alpha+1}}$$

et

$$F(x) = 1 - \left(\frac{\lambda}{x + \lambda}\right)^\alpha$$

Pour simuler une valeur selon une loi de fonction de repartion $F(x)$, on peut tirer une valeur u selon la loi uniforme $\mathcal{U}_{[0,1]}$ et calculer ensuite $F^{-1}(u)$, où $F^{-1}(u)$ est l'inverse (généralisée) de $F(x)$.

1. Trouver l'expression de $F^{-1}(u)$ et écrire une fonction `rpareto(n,shape,scale)` qui simule n valeurs i.i.d. selon une loi de Pareto de paramètres $\alpha =$`shape` et $\lambda =$`scale`.

2. Ecrire une expression `R` utilisant la fonction ci-dessus qui permet de simuler cinq échantillons aléatoires de tailles 100, 150, 200, 250, 300 d'une loi de Pareto avec $\alpha = 2$, $\lambda = 5000$. Les échantillons aléatoires doivent être stockés dans une liste.
3. Nommer les éléments de cette liste `sample1,...,sample5` à l'aide de la fonction `paste()`. Exemple d'utilisation de la fonction `paste` :

```
paste('a', 1:5, sep='')
```

```
## [1] "a1" "a2" "a3" "a4" "a5"
```

4. Calculer la moyenne de chacun des échantillons aléatoires obtenu en 2. Retourner le résultat dans un vecteur.
5. Evaluer la fonction de répartition de la loi de Pareto $\alpha = 2$, $\lambda = 5000$ en chacune des valeurs de chacun des échantillons aléatoires. Retourner les valeurs de la fonction de répartition en ordre croissant.
6. Faire l'histogramme des données du cinquième échantillon aléatoire. Comparer l'histogramme avec la densité f.
7. Ajouter 1000 à toutes les valeurs de tous les échantillons simulés en 1., ceci afin d'obtenir des observations d'une distribution de Pareto translatés.

4.3 Modèle linéaire, propriétés des estimateurs

Rappels sur le modèle linéaire. On considère des paires de points (x_i, y_i), $i = 1, \dots, n$ et on modélise leur relation par le modèle linéaire simple

$$Y_i = \alpha + \beta X_i + \epsilon_i$$

avec $(\epsilon_i)_{=1,\dots,n}$ erreurs i.i.d. avec $E[\epsilon_i] = 0$ et $V(\epsilon_i) = \sigma^2$ pour tout i. On a $E[Y_i|X_i = x] = \alpha + \beta x$ et $V(Y_i) = \sigma^2$.

On peut estimer α, β par la méthode des moindres carrés ordinaires. Si on fait l'hypothèse ultérieure que les ϵ_i suivent une loi normale $\mathcal{N}(0, \sigma^2)$, on peut montrer que les estimateurs des moindres carrés ordinaires $\hat{\alpha}, \hat{\beta}$ sont les estimateurs du maximum de vraisemblance. Les estimateurs du maximum de vraisemblance ont des propriétés très intéressantes, en particulier ils sont consistants et asymptotiquement normaux. Par exemple, pour β on a

— $\hat{\beta} \xrightarrow{P} \beta$ quand $n \to +\infty$ (consistance)
— $\frac{\hat{\beta}-\beta}{\hat{\text{se}}} \rightsquigarrow \mathcal{N}(0,1)$ quand $n \to +\infty$ (normalité asymptotique). Ici $\hat{\text{se}} = \sqrt{V(\hat{\beta})}$ est l'erreur standard de $\hat{\beta}$.

Dans cet exercice on veut vérifier empiriquement ces deux propriétés.

1. Consistance :
   a. Simuler $n = 1000$ observations i.i.d. x_i selon une loi $\mathcal{N}(165, 10^2)$.
   b. Simuler n observations y_i selon les lois $\mathcal{N}(\alpha + \beta x_i, \sigma^2)$, avec $\alpha = 165, \beta = 0.1$ et $\sigma = 0.5$. Cela revient à simuler $Y_i = \alpha + \beta x_i + \epsilon_i$ avec $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$. Visualiser le nuage des points (x_i, y_i).
   c. Pour $j = 1, \dots, n$, *fitter* le modèle linéaire expliquant y par x en utilisant les j premiers points $(x_1, y_1), \dots, (x_j, y_j)$ et récupérer $\hat{\beta}_j$. Indication : `lm()` permet de fitter un modèle linéaire et `model$coefficients` donne les estimations des coefficients dans le modèle `model <- lm(...)`.
   d. Montrer graphiquement la convergence de $\hat{\beta}_j$ vers β.
2. Normalité asymptotique :
   a. Simuler une population de $n = 10^6$ observations (x_i, y_i) comme ci-dessus.

b. Pour étudier la distribution de $\hat{\beta}$ on échantillonne aléatoirement les observations et pour chaque échantillon on estime le modèle (c'est l'idée à la base du *bootstrap*). Prendre 500 échantillons de taille 1000 : pour chaque échantillon fitter le modèle et récupérer $\hat{\beta}$.
c. Visualiser l'histogramme de $\frac{\hat{\beta}-\beta}{\hat{\text{se}}}$.
d. Superposer la densité de $\mathcal{N}(0,1)$ à l'histogramme.

Pour la question 1, on aurait aussi procéder de la façon suivante : pour tout $j = 1, \ldots, n$ on simule un échantillon de taille j, on fitte le modèle sur cet échantillon et on récupère $\hat{\beta}_j$. Quel différence observe-t-on avec l'approche précedante ?