Langage R en Actuariat

Nicolas Baradel

28 avril 2021

Table des matières

Introduction

Le cours requiert une première initiation à R. Pour une introduction au langage, on pourra se référer à [2], ou à [5] pour aller plus en profondeur. Les structures conditionnelles, de boucle ou de fonction sont considérées comme connues. Nous revoyons les structures de conteneur de R, comme les vecteurs, tableaux de données, qui sont approfondis afin d'apprendre à écrire un code R aussi concis qu'efficace.

1 Structure de données et programmation efficace

1.1 Les objets à structure vectorielle

1.1.1 Le vecteur

Le **vecteur** est l'objet fondamental de R. Il s'agit d'un regroupement de valeurs de même type. Pour créer un vecteur de type `double` et de taille 3, on écrit :

```
(x ← numeric(3))
```

```
[1] 0 0 0
```

On peut accéder à un élément en utilisant `[]` :

```
x[1]
```

```
[1] 0
```

L'indexation démarre à **1** et se fait de 1 à n. Un vecteur fondamental est une suite d'entierS de a à b qui s'obtient avec `a:b`.

```
1:5
```

```
[1] 1 2 3 4 5
```

La fonction `length` permet de renvoyer la longueur d'un vecteur.

```
length(x)
```

```
[1] 3
```

Il est possible de **concaténer** des éléments pour former un vecteur.

```
(x ← c(1, 3, 7))
```

```
[1] 1 3 7
```

Et même de concaténer des vecteurs.

```
c(x, x)
```

```
[1] 1 3 7 1 3 7
```

En fait, une variable de taille 1 est représentée comme un vecteur dans R.

```
x ← pi
x[1]
```

```
[1] 3.141593
```

```
length(x)
```

```
[1] 1
```

La fonction **rep** permet de créer un vecteur de taille n dont chaque composante est identique.

```
rep(5, 3)
```

```
[1] 5 5 5
```

La fonction **seq** (pour sequence) permet de créer une suite de nombre régulière.

```
seq(1, 2, 0.2)
```

```
[1] 1.0 1.2 1.4 1.6 1.8 2.0
```

```
seq(0, 1, length=6)
```

```
[1] 0.0 0.2 0.4 0.6 0.8 1.0
```

Il est possible de donner un vecteur de booléens qui donne les indices à garder.

```
x ← c(1, 3, 5)
x[c(FALSE, TRUE, TRUE)]
```

```
[1] 3 5
```

Il est possible de donner directement la valeur des indices choisis.

```
x[c(1, 3, 3, 2)]
```

```
[1] 1 5 5 3
```

Une autre possibilité est d'appeler tous les éléments sauf un. La syntaxe est `x[-a]` où `a` est un indice entier.

```
x[-2]
```

```
[1] 1 5
```

La règle suivante est **fondamentale** sur R.

Tous les opérateurs (arithmétiques, logiques, etc.) appliqués à deux vecteurs de même taille renvoient un vecteur de même taille où l'opérateur a été appliqué **élément par élément**.

```
y ← 0:2
x+y
```

```
[1] 1 4 7
```

```
x*y
```

```
[1] 0 3 10
```

```
x^y
```

```
[1] 1 3 25
```

```
x == y
```

```
[1] FALSE FALSE FALSE
```

Autre règle **fondamentale** : le **recyclage**.

Tous les opérateurs définis précédemment (arithmétiques, logiques, etc.) appliqués à un vecteur et une variable de taille un renvoient un vecteur où l'opérateur a été appliqué à la variable et à tous les éléments du vecteur initial **un à un**.

```
2*x+1
```

```
[1] 3 7 11
```

```
x^2
```

```
[1] 1 9 25
```

```
factorial(x) %% 2
```

```
[1] 1 0 0
```

```
x <= 2
```

```
[1] TRUE FALSE FALSE
```

Exercices

Les exercices sont à faire **sans boucle**.

- Ecrire une fonction `f` qui prend n en argument et qui renvoie un vecteur composé des $n+1$ premiers carrés de $\mathbb{N}$.

$$f : \mathbb{N} \to \bigcup_{n\in\mathbb{N}} \mathbb{N}^n$$
$$n \mapsto (0^2, 1^2, \dots, n^2)$$

- Ecrire une fonction `deriv1` qui prend un vecteur x, un pas h, et qui renvoie un vecteur de taille `length(x)` − `1` l'approximation du nombre dérivé :

$$\partial_{[h]} x_i := \frac{x_{i+1} - x_i}{h}.$$

Correction

-
```
f ← function(n)
    return((0:n)^2)
```

-
```
deriv1 ← function(x, h)
    return((x[−1] − x[−length(x)])/h)
```

1.1.2 Fonctions vectorielles et programmation efficace

On dira qu'une fonction `f` est **vectorielle** si `f` prend un ou des vecteurs en argument (et éventuellement d'autres arguments de taille 1) et renvoie un vecteur où une fonction a été appliquée élément par élément. C'est le cas par exemple de la fonction `sqrt` ou `exp`.

Soit la boucle de la forme :

```
for(i in I)
    x[i] ← f(i,x[i],z[i])
```

Si `f` est **vectorielle** alors cette boucle est **toujours évitable**, la solution est :

```
x ← f(1:length(x),x,z)
```

Par exemple, si nous souhaitons affecter à x le carré de i, la solution est

```
x ← (1:length(x))^2
```

Ou alors, associer à x_i l'exponentielle d'un élément z_i d'un vecteur z auquel on ajoute la constante 2, la solution est

```
x ← exp(z) + 2
```

Il se peut que nous souhaitions modifier x uniquement sur une partie de ses indices. Par exemple, quelque chose de la forme

```
for(i in I)
    if(h(i,x[i],z[i]))
        x[i] ← f(i,x[i],z[i])
```

où `h` est une fonction **vectorielle** qui renvoie `TRUE` ou `FALSE`, `f` n'est appliquée que sur un **sous-ensemble** de I où `h` est vérifiée.

Si `f` et `h` sont **vectorielles** alors cette boucle est **toujours évitable**, la solution est :

```
ind ← h(1:length(x),x,z)
    x[ind] ← f((1:length(x))[ind], x[ind], z[ind])
```

Il ne faut pas être effrayé par le fait d'écrire x dans x. Ce qui est à l'intérieur n'est que le calcul d'un vecteur de `logical` en fonction de x (s'il vaut `NA` ou non). Ensuite, nous effectuons une opération dans les indices de x vérifiant cette condition. Par exemple

```
(x ← c(7, 2, NA, 3, −1, NA))
```

```
[1] 7 2 NA 3 -1 NA
```

```
x[is.na(x)] ← 0
x
```

```
[1] 7 2 0 3 -1 0
```

Un autre exemple : là où la somme de x et y (deux vecteurs de même taille) est supérieure à z, affecter à x le modulo de z par 2, sinon celui de y par 2

```
ind ← x + y > z
x[ind] ← z[ind]%%2
x[!ind] ← y[!ind]%%2
```

Rappel : le point d'exclamation est le NON logique, il inverse les `TRUE` et `FALSE`. Nous affectons, en fonction de $x + y > z$, à chaque élément, soit `z%%2`, `y%%2`.

Un dernier exemple intervient dans la classification d'une variable. Par exemple, si x est inférieur à un seuil a fixé, nous le mettons dans la classe 0, s'il est supérieur à b, nous le mettons dans la classe 2, et enfin s'il est entre les deux, dans la classe 1.

```
y ← rep(1, length(x))
y[x < a] ← 0
y[x > b] ← 2
```

Pour les conditions, les opérateurs `&&` et `||` ne sont pas vectoriels. Les versions vectorielles correspondantes sont `&` et `|`.

Pour tester si une condition est vérifiée sur **tous** les éléments d'un vecteur, on utilisera la fonction `all`.

Pour tester si une condition est vérifiée sur **au moins** un élément d'un vecteur, on utilisera la fonction `any`.

Exercices

Les exercices sont à faire **sans boucle**.

- Écrire une fonction `gammaEuler` qui approxime à l'ordre $n \in \mathbb{N}^*$ la constante γ d'Euler définie par la limite :
$$\gamma = \lim_{n\to+\infty} \left(\sum_{k=1}^{n} \frac{1}{k} - \log(n) \right).$$
On pourra vérifier avec `-digamma(1)` qui vaut γ.
- Soit X une variable aléatoire dont la densité est définie par la fonction f :
$$\forall x \in \mathbb{R},\ f(x) = \frac{1}{\sqrt{2\pi}(1+x^2)} \exp\left(-\frac{\tan^2(x)}{2}\right) \mathbf{1}_{]-\frac{\pi}{2},\frac{\pi}{2}[}(x)$$
où $\mathbf{1}$ est la fonction indicatrice (elle vaut 1 si x est dans l'intervalle, 0 sinon). Écrire cette fois-ci la fonction f de manière vectorielle sous R afin qu'elle puisse prendre un vecteur x et renvoie un vecteur de même taille $f(x)$ où f est appliquée élément par élément.
- Estimons π par méthode de Monte-Carlo. Pour ce faire, prenons le carré unité $[-1;1]^2$ et le disque de rayon 1 et de centre 0 de ce carré. L'aire du carré est 4, l'aire du disque est π. Si on tire uniformément dans le carré (ce qui revient à tirer deux lois uniformes dans [-1,1], une étant l'axe des abscisses, l'autre l'axe des ordonnées), la probabilité d'être dans le disque est $\frac{\pi}{4}$. Écrire une fonction qui prend en argument le nombre de simulations n et qui renvoie une estimation π.

Correction

- ```
gammaEuler ← function(n)
    return(sum(1/(1:n)) − log(n))
```

- ```
f ← function(x)
{
    y ← numeric(length(x))
    ind ← x > −pi/2 & x < pi/2
    z ← x[ind]
    y[ind] ← exp(−0.5∗tan(z)^2)/(sqrt(2∗pi)∗(1+z^2))
    return(y)
}
```

- ```
MCpi ← function(n)
    return(4∗mean(runif(n,−1,1)^2 + runif(n,−1,1)^2 <= 1))
```

### 1.1.3 Les matrices : une extension du vecteur

Une **matrice** est un vecteur avec une représentation à **deux indices**. Une matrice se crée avec la fonction **matrix** dans laquelle on donne un vecteur, le nombre de lignes, le nombre de colonnes.

```
(X ← matrix(1:9, 3, 3))
```

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

On peut accéder à un élément (i, j) en utilisant `[, ]` :

```
X[2, 3]
```

```
[1] 6
```

On peut aussi l'appeler en utilisant un indice de type *vecteur*, i.e. on utilisant :

$$i' := j + (i - 1) \times \texttt{nrow}$$

```
X[6]
```

```
[1] 6
```

Il est possible de remplir une matrice par *ligne* avec l'argument `byrow = TRUE`.

La fonction **as.matrix** permet de convertir un vecteur de taille n en une matrice à n lignes et une colonne.

La fonction **t** permet d'obtenir la matrice **tranposée**.

Le **produit matriciel** se fait avec **%∗%** et non pas avec $*$, ce dernier est le produit des deux matrices **élément par élément**.

Il est possible de construire une matrice **diagonale** avec **diag** :
```

```r
diag(1:3)
```

```
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3
```

La fonction **cbind** permet de combiner des vecteurs en une matrice en les plaçant par colonne :

```r
cbind(rep(1,3), rep(2,3), rep(3,3))
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    1    2    3
[3,]    1    2    3
```

On peut extraire une ligne ou une colonne sous la forme d'un vecteur en ne spécifiant que l'indice la ligne ou de la colonne

```r
X[1, ]
```

```
C1 C2 C3
 1  4  7
```

```r
X[, 1]
```

```
R1 R2 R3
 1  2  3
```

On peut aussi utiliser le nom de la ligne ou de la colonne

```r
X["R1", ]
```

```
C1 C2 C3
 1  4  7
```

Il est possible de forcer la conservation du type **matrix** avec l'argument **drop = FALSE**

```r
c(is.matrix(X[1, ]), is.matrix(X[1, , drop=FALSE]))
```

```
[1] FALSE TRUE
```

```r
dim(X[1, , drop=FALSE] #dim renvoie le nombre de lignes et de
   colonnes
```

```
[1] 1 3
```

La fonction **length** renverra le nombre total d'éléments de la matrice

```r
length(X)
```

```
[1] 9
```

On peut obtenir le nombre de lignes grâce à la fonction **nrow** et le nombre de colonnes grâce à la fonction **ncol**

```
c(nrow(X), ncol(X))
```

```
[1] 3 3
```

Le langage vectoriel fonctionne également avec les matrices. On peut appeler `X[a, b]` où `a` est un vecteur de **TRUE** et **FALSE** qui sélectionne les lignes, et `b` de même qui sélectionne les colonnes. Enfin, il est possible d'appeler `X[A]` où `A` est une matrice de booléen qui sélectionne les indices à conserver de la matrice.

Exercices

- On suppose avoir la matrice suivante :

```
X ← cbind(c(1,2,1,3,2), c(121, 256, 842, 510, 82), c(1, 2, 3,
   4, 5), c(5, 11, 2, 7, 3))
```

  Trier la matrice par ordre croissant selon la première colonne et, en cas d'égalité, selon la deuxième colonne (toujours par ordre croissant). On utilisera la fonction **order**, celle-ci renvoie les indices d'un vecteur de telle sorte que `x[order(x)]` renvoie `sort(x)`.

- Écrire une fonction qui prend une matrice $X = (x_{i,j})$ de taille $n \times m$ en argument et qui renvoie un vecteur de taille m où l'élément j est la moyenne sur i de $(cos(x_{i,j})^i)_{1 \leq i \leq n}$ (attention, bien voir que le cosinus est élevé à la puissance i). On pourra utiliser la fonction **row** qui renvoie une matrice de même dimension que celle donnée en argument, où chaque indice est le numéro de ligne : $(\texttt{row}(x))_{i,j} = i$, et la fonction **col** qui fait de même mais avec les colonnes : $(\texttt{col}(x))_{i,j} = j$.

- Écrire une fonction qui prend une matrice carrée $X = (x_{i,j})$ de taille $n \times n$ en argument et renvoie la trace de la matrice X, sans utiliser la fonction **diag**. La trace de la matrice X est la somme des éléments diagonaux définie par

$$\texttt{Trace}(X) = \sum_{k=1}^{n} x_{k,k}.$$

Correction

-
```
X ← cbind(c(1, 2, 1, 3, 1), c(121, 256, 842, 510, 82), 1:5,
    c(5, 11, 2, 7, 3))
X ← X[order(X[, 1], X[, 2]), ]
```

-
```
f ← function(X)
    return( colMeans(cos(X)^row(X)) )
```

-
```
Trace ← function(x)
    return(sum(X[row(X) == col(X)]))
```

1.2 Les listes

La liste est un regroupement d'**objets arbitraires**. L'exemple suivant illustre la création d'une liste

```
li ← list(a = TRUE, b = 1:3)
```

```
$a
[1] TRUE
$b
[1] 1 2 3
```

L'appel peut se faire par le nom avec `$` ou `[[]]`

```
li$a
```

```
[1] TRUE
```

```
li[["a"]]
```

```
[1] TRUE
```

L'ajout de nouveaux éléments à la liste se fait facilement :

```
li$c ← "s"
li$c
```

```
[1] "s"
```

Il est possible de récupérer une sous liste en plaçant le vecteur des noms de la sous liste entre `[]`.

```
li[c("b", "c")]
```

```
$b
[1] 1 2 3
$c
[1] "s"
```

1.3 Supplément : Associativité et dépassement de capacité

Les variables numériques (non entières) dans **R** sont les **double**. Une telle variable est stockée sur 8 octets (64 bits) et son nom provient du fait que la capacité est doublée par rapport au `float` qui lui pèse 4 octets. Cela implique naturellement qu'il y a un nombre fini de nombres représentables : 2^{64} nombres au maximum, ainsi par construction,

- Il y a un nombre minimum et un nombre maximum ;
- Il y a un plus petit nombre strictement positif ;
- Il y a une précision maximale.

Pour bien comprendre, il faut savoir comment est représenté le **double** dans **R** (ou tout autre langage).

Tout nombre x a la représentation scientifique unique (sauf pour 0) suivante en base 10

$$x = s \times m \times 10^e, \quad s = \pm 1,\ 1 \leq m < 10,\ e \in \mathbb{Z}.$$

En informatique, les calculs (et le stockage) se font en base 2, ce qui se traduit par la représentation :

$$x = s \times m \times 2^e, \quad s = \pm 1,\ 1 \leq m < 2,\ e \in \mathbb{Z}.$$

Les 64 bits de stockages se répartissent en :

- 1 bit représente le signe (+ ou -) ;
- 11 bits pour l'exposant ;
- 53 bits pour la mantisse.

On peut en déduire que

- La partie en exposant donnera le nombre le plus grand qui sera tout au plus $2^{2^{11-1}} = 2^{1024} \approx 10^{308}$, (ce nombre représente l'infini),
- Tandis que, sans entrer dans les détails, le nombre strictement positif minimum est $2^{-1074} \approx 10^{-324}$.
- La précision donnée par la mantisse sera tout au plus de $1/2^{53} \approx 1.110223 \times 10^{-16}$ pour la mantisse (à multiplier par la partie puissance).

En conséquence nous pouvons observer que le calcul n'est pas toujours associatif, et que l'addition de deux nombres, dont l'un est plus petit que l'autre d'un facteur d'environ 10^{16} peut en supprimer l'information, et rendre le calcul non associatif.

```
-1 + (1 + 1e-16)
```

```
[1] 0
```

```
(-1 + 1) + 1e-16
```

```
[1] 1e-16
```

Pour les dépassements de capacité (nombre trop petit ramené à 0, ou trop grand et ramené à l'infini) en utilisant les bornes données, on observe :

```
c(2^(-1074), 2^(-1075))
```

```
[1] 4.940656e-324  0.000000e+00
```

```
c(2^1023, 2^1024)
```

```
[1] 8.988466e+307  Inf
```

En conséquence, des choses simples comme la composition de fonctions inverses peuvent ne plus fonctionner, tant bien même que le résultat final est d'un ordre de grandeur habituel.

```
log(exp(1000))
```

```
[1] Inf
```

Ici, il est facile de simplifier à l'avance le calcul, mais dans certaines situations où le résultat est d'un ordre de grandeur habituel, mais s'exprime comme $\log(A)$ où A est une valeur de calcul intermédiaire très grande, dépassant la capacité, résoudre ce problème paraît moins évident. Comme par exemple pour le calcul de $\log(n!)$ dès $n = 200$.

```
log(factorial(200))
```

```
[1] Inf
```

Pour ces situations, R a prévu des fonctions qui utilisent un aglorithme spécifique et qui permettent d'éviter un dépassement de capacité lors des calculs intermédiaires. Il s'agit ici de la fonction `lfactorial`.

```
lfactorial(200)
```

```
[1] 863.232
```

R a introduit d'autres fonctions spécifiques comme `expm1(x)` $= e^x - 1$ pour x au voisinage de 0, `log1p(x)` $= \log(1+x)$ pour x au voisinage de 0, ou encore `lgamma(x)` $= \log(\Gamma(x))$.

Exercices

- Construire une fonction qui puisse évaluer

$$\frac{\Gamma(\alpha)}{\Gamma(\beta)},$$

en particulier, au point $(\alpha = 200 + \sqrt{\pi}, \beta = 200)$.

- Construire une fonction qui évalue le logarithme de la densité de la loi de student, (sans utiliser la fonction `dt`). La densité de la loi de student de paramètre n est définie par :

$$\begin{aligned} f : \mathbb{R} \times \mathbb{R}_+ &\to \mathbb{R}_+^* \\ (x, n) &\mapsto \frac{1}{\sqrt{n\pi}} \frac{\Gamma(\frac{n+1}{2})}{\Gamma(\frac{n}{2})} \left(1 + \frac{x^2}{n}\right)^{-\frac{n+1}{2}}. \end{aligned}$$

La fonction doit être évaluable pour $x \in [-1000, 1000]$ et $n \in [-1000, 1000]$.

Correction

-
```
f ← function(alpha, beta)
    return(exp(lgamma(alpha) - lgamma(beta)))

f(200 + sqrt(pi), 200)
```
```
[1] 12021.34
```

-
```
f ← function(x, n)
    return( -0.5*log(n*pi) + lgamma((n+1)/2) - lgamma(n/2)
       -0.5*(n+1)*log(1+x^2/n) )

f(1000, 1000)
```
```
[1] -3458.751
```

2 Manipulation de données

L'objet R pour la manipulation de données est la `data.frame`.

Le paquet **data.table** permet de manipuler des **data.frame** avec des outils supplémentaires. Il offre également de meilleures performances lorsque les données sont volumineuses. Il est compatible avec les **data.frame**.

Le paquet **dplyr** (de la famille **tidyverse**) offre une alternative complètement différente en terme de manipulation de données. Toutefois, **dplyr** n'est pas compatible avec les **data.frame** , est moins performant que **data.table** (et parfois que **data.frame**), et sa syntaxe d'usage diffère complètement de celle de **R** : ils ne seront pas abordés.

2.1 La data.frame

La **data.frame** est une liste qui regroupe des vecteurs de même taille. Cette liste se présente comme un tableau à deux dimensions, où chaque colonne est un élément de la liste : c'est un vecteur. En conséquence, chaque colonne a un type, mais ce type peut varier d'une colonne à une autre. En résumé, une data.frame est un regroupement de vecteurs de même taille sous forme de liste, où chaque colonne est un vecteur qui a son propre type.

Cette particularité fait que la data.frame partage la syntaxe des matrices et des listes, mais est bien stockée comme une liste.

```
is.list(data.frame())
```

```
[1] TRUE
```

L'objet peut également être vu comme un tableau à deux dimensions pour utiliser la syntaxe matricielle. On préférera l'appel sous forme de liste, quand c'est possible.

Prenons la data.frame **iris** présente dans **R**. Pour appeler une colonne, par exemple **Sepal.Length** , on utilise la syntaxe des listes :

```
iris$Sepal.Length
```

```
[1] 5.1 4.9 4.7 4.6 5.0 ...
```

Si le nom de la colonne est une chaîne de caractère stockée dans une variable, nous passerons par :

```
nomcol ← "Sepal.Length"
iris[[nomcol]]
```

```
[1] 5.1 4.9 4.7 4.6 5.0 ...
```

Pour sélectionner plusieurs colonnes, on utilisera :

```
iris[c("Sepal.Length", "Sepal.Width")]
```

```
Sepal.Length Sepal.Width
...
```

À noter que **iris["Sepal.Length"]** renvoie une **data.frame** d'une colonne tandis que **iris[["Sepal.Length"]]** et **iris$Sepal.Length** renvoient un vecteur.

La notation *matricielle* équivalente est **iris[, "Sepal.Length"]** (renvoie un vecteur, sauf si on ajoute **drop = FALSE**) et **iris[, c("Sepal.Length", "Sepal.Width")]**.

Pour sélectionner (filtrer) les lignes, on utilisera la notation matricielle : on placera en premier indice le vecteur des lignes à sélectionner (un vecteur d'entier, ou un vecteur de booléens du

nême nombre de lignes que la data.frame où les TRUE sont positionnés dans les lignes à conserver).

```
iris[iris$Species == "versicolor", ]
```

Ceci est équivalent à `subset(iris, iris$Species == "versicolor")`, mais on préférera la version matricielle.

Pour sélectionner une colonne, on pourra utiliser `iris[iris$Species == "versicolor", "Sepal.Length"]` ou, légèrement plus efficacement : on etrait dans un premier temps le vecteur qui est la colonne sélectionnée, puis on y applique le filtre.

```
iris$Sepal.Length[iris$Species == "versicolor"]
```

On peut sélectionner plusieurs colonnes.

```
iris[iris$Species == "versicolor", c("Petal.Length","Sepal.Width")]
```

Comme pour les vecteurs, il est possible de combiner avec `&` (et logique vectoriel) et `|` (ou logique vectoriel) les conditions.

```
iris[iris$Species == "versicolor" | iris$Species == "virginica", c(
   "Petal.Length", "Sepal.Width")]
```

Pour tester plus simplement l'appartenance à un ensemble (ici si `iris$Species` $\in$ {`"versicolor"`, `"virginica"`}, il y a l'opérateur `%in%`.

```
iris[iris$Species %in% c("versicolor", "virginica"), c("
   Petal.Length", "Sepal.Width")]
```

Quelques fonctions utiles pour les `data.frame`

- `str` affiche les différentes colonnes de la data.frame ainsi que leur type et les premières valeurs. Exemple : `str(iris)`.
- `summary` : pour chaque colonne numérique : affiche le minimum, le premier quartile, la médiane, la moyene, le troisième quartile et le maximum. Pour chaque colonne de chaînes de caractères ou de facteurs affiche les modalités et le nombre d'occurences. Exemple : `summary(iris)`.
- `tapply` regroupe les éléments d'un vecteur (premier argument), selon les modalité d'un second vecteur de même taille (deuxième argument) avec une fonction d'agrégation (troisième argument). Exemple : `tapply(iris$Sepal.Length, iris$Species, mean)`.

La fonction `tapply` s'assimile à un *group by* en requête SQL et est très utile, car on peut y insérer ses propres fonctions. Par exemple, pour connaître le taux de données qui ne sont pas des `NA` par Species, on peut introduire une fonction puis l'utiliser derrière avec `tapply`.

```
f ← function(x) return(sum(!is.na(x)))
tapply(iris$Sepal.Length, iris$Species, f)
```

2.2 Le paquet `data.table`

Le paquet utilise son programe objet : le `data.table`. Il est possible de convertir une `data.frame` en `data.table` avec la fonction `as.data.table` (et de revenir à une `data.frame` avec `as.data.frame`).

```
tiris ← as.data.table(iris)
```

Un **data.table** garde la structure d'une **data.frame**.

```
c(is.data.frame(tiris), is.data.table(tiris))
```

```
[1] TRUE TRUE
```

Cependant, la sélection de colonnes avec `tiris["Sepal.Length"]` ne fonctionne plus avec les data.frame, il utiliser la version matricielle en rajoutant une vigule : `tiris[, "Sepal.Length"]`.

Lors de la sélection de colonnes avec la notation matricielle, **data.table** n'évalue pas les variables, mais prend la chaîne de caractères.

```
tiris[, Sepal.Length]
```

```
[1] 5.1 4.9 4.7 4.6 5.0
```

Il faut voir cela comme utiliser `tiris$Sepal.Length`. Cela est compatible avec plusieurs colonnes

```
tiris[, c(Sepal.Length, Sepal.Width)]
```

A noter que lors d'un appel de la forme `x[, a]`, `a` n'est jamais évalué : on recherche la colonne a. Pour évaluer `a`, on passe par la syntaxe de liste `tiris[[a]]`.

Cependant cela pourrait être problématique : comment sélectionner plusieurs colonnes avec une variable ? Il est possible de forcer l'évaluation en variable avec l'argument `with = FALSE`, si `a` est une variable représentant une ou plusieurs colonnes :

```
tiris[, a, with=FALSE]
```

Pour sélectionner des lignes, il est possible d'écrire

```
tiris[tiris$Species == "versicolor"]
```

L'écriture matricielle `iris[iris$Species == "versicolor", ]` fonctionne également. Si on met le nom d'une colonne directement dans la formule de sélection de lignes, cela fonctionne : en premier **data.table** associe les nom de variables aux colonnes, puis dans un second temps les évalue.

```
tiris[Species == "versicolor"]
```

Trier une **data.frame** se fait avec **order**. Pour trier par `Sepal.Length` puis par `Species`,

```
iris[order(iris$Species, iris$Sepal.Length), ]
```

Cela fonctionne avec **data.table** car **order** renvoie les numéros de lignes. Avec **data.table** il est possible d'écrire

```
tiris[order(Species, Sepal.Length)]
```

Il est parfois utile de passer d'un format à l'autre avec **as.data.frame** et **as.data.table**. Ces fonctions passent par une copie. Pour modifier directement l'objet de manière très efficace nous avons les fonctions **setDF** (set data.frame) et **setDT** (set data.table).

```
setDF(tiris)
setDT(tiris)
```

Les `data.frame` ont la fonction `read.csv` pour importer des données. Le paquet `data.table` a la fonction `fread`. Cette fonction a les mêmes arguments que `read.csv`, mais pas les même valeurs par défaut pour tous. Elle est plus rapide.

2.3 Les recherches REGEX

Regex est l'abréviation de *regular expression* (expression régulière ou expression rationnelle).

Une recherche *basique* en chaine de caractères est de vérifier si une chaîne est contenue dans une autre Une recherche Regex permet de faire une recherche par motif, et de capturer des textes qui ont des formats particulier et qui ont un environnement textuel particulier.

2.3.1 Recherche par motif

La fonction dans R qui permet de trouver des expressions régulières est `gregexpr(motif, texte)`. L'argument `motif` est le motif recherché dans `texte`. La fonction renvoie les positions trouvées, pour obtenir les chaînes de caractère, il faut utiliser `regmatches(texte, matches)`.

L'appel `regmatches(texte, gregexpr(motif, texte))` permet de récupérer le résultat d'une recherche Regex.

Nous définissons la fonction `regex` qui permet de faire une recherche Regex directement et nous l'utiliserons par la suite.

```
regex ← function(motif, texte)
    return( regmatches(texte, gregexpr(motif, texte))[[1]] )
```

Une expression régulière est une chaîne de caractères qui décrit le motif recherché. Elle est composée de caractères simples et de *metacharactères* qui ont une fonction spéciale.

Par exemple le caractère **|** signigie *ou*.

```
texte ← "Un oiseau, deux oies, une autre oie et une pintade."
regex("oiseau|oie", texte)
```

```
[1] "oiseau" "oie" "oie"
```

Un caractère suivi d'un signe **?** est optionnel. Le **?** ne s'applique qu'au caractère précédent.

```
regex("oiseaux?|oies?", texte)
```

```
[1] "oiseau" "oies" "oie"
```

Dans l'exemple suivant :

```
texte ← "Un chat blanc, un chat  brun, un chat tacheté et un chat\
   nnoir."
regex("chat blanc|chat brun|chat noir", texte)
```

```
[1] "chat blanc"
```

On échoue à capturer le chat brun et noir. Dans le cas du noir, le caractère **\n** représente le retour à la ligne. On souhaite récupérer le motif, peu importe qu'il s'agisse d'un retour à la ligne ou d'un espace. Pour cela, on utilise le caractère spécial **\s** qui symbolise *un caractère d'espacement*, les tabulations sont aussi incluses. Rappelons que dans R , pour obtenir le caractère ****, il faut écrire ****.

```
regex("chat\\sblanc|chat\\sbrun|chat\\snoir", texte)
```

```
[1] "chat blanc" "chat\nnoir"
```

Le chat brun n'est pas capturé car il y deux espaces. Un caractère suivi d'un signe **+** doit être présent une ou plusieurs fois. Le **+** ne s'applique qu'au caractère précédent.

```
regex("chat\\s+blanc|chat\\s+brun|chat\\s+noir", texte)
```

```
[1] "chat blanc" "chat  brun" "chat\nnoir"
```

Il est possible de factoriser la partie commune en utilisant des parenthèses.

```
regex("chat\\s+(blanc|brun|noir)", texte)
```

```
[1] "chat blanc" "chat  brun" "chat\nnoir"
```

Un caractère suivi d'un signe ***** peut être présent zéro, une ou plusieurs fois.

```
texte ← "Un chatblanc, un chat  brun et un chat\nnoir."
regex("chat\\s*(blanc|brun|noir)", texte)
```

```
[1] "chatblanc" "chat  brun" "chat\nnoir"
```

Les symboles **?**, **+** et ***** sont des quantificateurs, ils s'appliquent directement au caractère (ou groupe de caractère en cas de parenthèse) qui le précède.

Un caractère devant **?** doit être présent zéro ou une fois. Un caractère devant **+** doit être présent une ou plusieurs fois. Un caractère devant ***** doit être présent zéro, une ou plusieurs fois.

Il est possible de spécifier le nombre de fois exact avec **{n}** avec $n \in \mathbb{N}$. Par exemple **a{3}** demande à ce que **a** soit présent exactement 3 fois, c'est équivalent à **aaa**.

De manière plus générale, **{n, m}** précise que le caractère doit être présent entre n et m fois. **{n, }** précise que le caractère doit être présent au moins n fois et **{, m}** doit être présent au plus m fois (et au moins une fois, le minimum par défaut est fixé à 1).

Les symboles **?**, **+** et ***** sont respectivement équivalents à **{0, 1}**, **{1, }** et **{0, }**.

Le symbole **.** représente tous les caractères. Par exemple, si on veut récupérer les mots en gras dans une chaîne de caractères provenant d'une page internet :

```
texte ← "Un <strong>cheval blanc</strong> et un <strong>cheval noir
   </strong>."
regex("<strong>.*</strong>", texte)
```

```
[1] "<strong>cheval blanc</strong> et un <strong>cheval noir</strong>"
```

Cela fonctionne, mais on pourrait s'attendre à récupérer deux chaînes. Par défaut, les quantificateurs sont *gourmands* : ils essaient de prendre un maximum de caractères. Pour rendre le quantificateur ***** non gourmand, on lui fait suivre **?**.

```
regex("<strong>.*?</strong>", texte)
```

```
[1] "<strong>cheval blanc</strong>" "<strong>cheval noir</strong>"
```

Avec ***** il est possible de rendre non groumand **+** et **{n, }**, avec **+?** et **{n, }?**
Pour choisir un caractère qui appartient à une sélection, on utilise **[]**. Pour demander à ce que le caractère soit `a`, `b` ou `c`, on écrit simplement `[abc]`.

Pour sélectionner une plage de caractère, par exemple entre `a` et `g` on écrit `[a-g]`. Les cas les plus utilisés sont **[a-z]** (lettre minuscule), **[A-Z]** (lettre majuscule). La plage correspond à celle du code ASCII, les caractères accentués ne sont pas pris en compte dans ces plages.

Il est possible de combiner avec d'autres choix ou plages. `[a-z_]` représentera une lettre minuscule ou le caractère souligné. Tandis que `[A-Za-z]` représente les lettres minuscules et majuscules.

Il existe des groupes de caractères particuliers.

`[[:lower:]]` et `[[:upper:]]` considèrent tous les caractères minuscules ou majuscules, en incluant les caractères accentués.

`[[:digit:]]` est équivalent à `[0-9]` lui même équivalent **\\d**.

`[[:alpha:]]` est équivalent à `[[:lower:][:upper:]]`.

`[[:alnum:]]` est équivalent à `[[:alpha:][:digit:]]`.

Il est possible d'exiger que la recherche commence sur le début de la chaîne et non pas n'importe où. En faisant commencer la commande par **^**, la recherche doit correspondre au début de la chaîne, tandis qu'avec **$** en fin de chaîne, celle-ci doit se terminer avec le motif recherché. En combinant les deux, la chaîne passée en argument doit être le motif exact recherché.

```
texte ← "email@domaine.com"
regex("^[[:alnum:]]+@[[:alnum:]]+\\.[a-z]{2,5}$", texte)
```

```
[1] "email@domaine.com"
```

Ci-dessus, comme nous cherchons dans la recherche un `.` et que ce caractère a une signification spéciale, on doit l'échapper, cela se fait en rajouter un `\` qui doit être doublé en **R** pour être interprété comme tel.

L'email est renvoyé car cela correspond à la recherche. L'adresse passée a donc le format d'une adresse email. Pour tester un motif et non le capturer, on utilise la fonction **grepl**.

```
texte ← "email@domaine.com"
grepl("^[[:alnum:]]+@[[:alnum:]]+\\.[a-z]{2,5}$", texte)
```

```
[1] TRUE
```

Remarque : une adresse email peut contenir certains caractères spéciaux (comme un `.`), mais ne peut pas commencer avec. Une bonne vérification d'adresse email peut se faire avec le motif `^[[:alnum:]]([-_.]?[[:alnum:]])*@[[:alnum:]]([-.]?[[:alnum:]])*\\.([a-z]{2,5})$`.

Exercices

- Tester si une chaîne de caractères a le format d'une adresse web. Celle-ci peut commencer (mais c'est facultatif) par **http://** ou **https://**, doit être suivie d'une chaîne alphanumérique non accentuée d'au moins 2 caractères et peut également contenir – ou `.` sans

qu'ils puissent se suivre, et doit terminer par {.extension} où l'extension est une chaîne de caractère de taille 2 à 5. On testera avec les adresses candidates :

```
texte ← c("http://domaine.com", "https://domaine-123.info", "
   www.sous-domaine.domaine.org", "1and1.net", "100%.info", "
   http://sousdomaine..domaine.info", "ftp://domaine.com", "
   .domaine.com")
```

Les 4 premières sont des adresses valides tandis que les 4 dernières sont invalides.

Correction

- ```
  grepl("^(https?://)?([a-z0-9][-.]?)+[a-z0-9]\\.[a-z]{2,5}$"
     , texte)
  ```
  ```
  [1] TRUE TRUE TRUE TRUE FALSE FALSE FALSE FALSE
  ```

2.3.2 Capture d'un sous-motif

La recherche via un motif renvoie l'intégralité du motif. Parfois, on cherche une chaîne de caractères incluse dans un motif, et seule la chaîne incluse nous intéresse. Si on recherche le texte en gras dans

```
texte ← "Voici un cheval blanc."
```

Le texte est encadré par et , mais on souhaite ne récupérer que le texte intérieur. Pour cela, nous faisons une **capture** dans le motif. La fonction **regexec** permet de capturer des sous motifs et remplace **gregexpr** . Nous définissons la fonction **regexc** qui permet de faire une recherche Regex avec capture directement et nous l'utiliserons par la suite.

```
regexc ← function(motif, texte)
    return( regmatches(texte, regexec(motif, texte)) )
```

Toute zone entre parenthèses dans le motif sera capturée. Il est possible de capturer plusieurs sous motifs.

```
regexc('(.*?)', texte)[[1]]
```

```
[1] "un cheval blanc" "un cheval blanc"
```

En premier elle renvoie le motif complet, puis chacun de sous motifs capturés. Parfois, les parenthèses sont utilisées pour appliquer un quantificateur à un groupe, et non pour capturer. Pour refuser la capture, on fait suivre la parenthèse ouvrante par **? :**.

```
regexc('(?:.*?)', texte)[[1]]
```

```
[1] "un cheval blanc"
```

Le défaut de **regexec**, utilisée dans **regexc**, c'est qu'elle ne permet qu'une seule correspondance de motif, mais permet de capturer dans le motif. Le défaut de **gregexpr**, utilisée dans **regex** , c'est qu'elle ne permet pas de capturer dans le motif. En revanche, elle capture toutes les correspondances du motif.

Pour capturer dans un motif plusieurs fois, il faut combiner les deux.

Soit le texte suivant :
```

```
texte ← 'Du contenu préalable...
<ul id="autre">
    <li>autre</li>
</ul>
<ul id="planetes">
    <li>Vénus</li>
    <li>Mars</li>
    <li>Terre</li>
</ul>
Du contenu postérieur...'
```

Objectif : capturer la liste des planètes en un vecteur. On ne peut pas simplement rechercher le motif `"<li>(.*?)</li>"` car dans ce cas, on capturerait d'autres éléments d'autres listes. Dans un premier temps, on extrait la liste complète avec les planètes.

```
(global ← regex('<ul id="planetes">.*?</ul>', texte))
```

```
[1] "<ul id=\"planetes\">\n    <li>Vénus</li>\n    <li>Mars</li>\n
<li>Terre</li>\n</ul>"
```

Ensuite, on capture les motifs.

```
(sous_motifs_complets ← regex("<li>(.*?)</li>", global))
```

```
[1] "<li>Vénus</li>" "<li>Mars</li>" "<li>Terre</li>"
```

La fonction capture bien l'ensemble des motifs mais ne capture pas de sous motif. Pour capturer chacun des sous-motifs, on utilise la fonction **regexc**.

```
(sous_motifs ← regexc("<li>(.*?)</li>", sous_motifs_complets))
```

```
[[1]]
[1] "<li>Vénus</li>" "Vénus"
[[2]]
[1] "<li>Mars</li>" "Mars"
[[3]]
[1] "<li>Terre</li>" "Terre"
```

Exercices

- Capturez la liste des paquets R actuellement disponibles sur le CRAN qui se trouvent dans la page : `https://cran.r-project.org/web/packages/available_packages_by_name.html`. On pourra importer la page dans R avec :

  ```
  html ← paste(readLines("https://cran.r-project.org/web/packages
     /available_packages_by_name.html", encoding="UTF-8"),
     collapse = "\n")
  ```

  Le résultat final sera une **data.frame** dont la première colonne est les noms des paquets et la seconde colonne est les descriptions des paquets.
- Capturez le dernier mois de cours du CAC 40 : `https://www.boursorama.com/bourse/indices/cours/historique/1rPCAC`. Le résultat final sera une **data.frame** qui correspond au tableau de données (24 lignes, 6 colonnes).

Correction

- On importe dans un premier temps la page web

```
html ← paste(readLines("https://cran.r-project.org/web/packages
   /available_packages_by_name.html", encoding="UTF-8"),
   collapse = "\n")
```

On extrait le tableau qui contient les données (cela évite d'entrer en conflit avec un autre éventuel tableau, par exemple).

```
html_table ← regex('<table summary="Available CRAN packages by
   name.">.*?</table>', html)
```

On extrait le motif de chaque ligne de paquet (sans capture).

```
html_packages ← regex('<tr>.*?</tr>', html_table)
```

Pour chaque motif, on capture le nom du paquet et sa description.

```
packages ← regexc('<tr>\\s*<td>\\s*<a.*?>(.*?)</a>\\s*</td>\\s
   *<td>(.*?)</td>\\s*</tr>', html_packages)
```

Puis on transforme en **data.frame**.

```
nom ← resume ← numeric(length(packages))
for(i in 1:length(packages))
{
        nom[i] ← packages[[i]][2]
        resume[i] ← packages[[i]][3]
}
P ← data.frame(nom, resume)
```

- On importe dans un premier temps la page web.

```
html ← paste(readLines("https://www.boursorama.com/bourse/
   indices/cours/historique/1rPCAC", encoding="UTF-8"),
   collapse="\n")
```

On extrait le tableau qui contient les données (cela évite d'entrer en conflit avec un autre éventuel tableau, par exemple) en repérant comment extraire le bon.

```
html_table ← regex('<table class="c-table" data-table-sorter>\\
   s*<thead>.*?</thead>.*?</table>', html)
```

On extrait le motif de chaque ligne de cotation (sans capture).

```
html_days ← regex('<tr\\s*class="c-table__row">.*?</tr>',
   html_table)
```

Pour chaque motif, on capture les 6 éléments des colonnes en nettoyant des espaces.

```
motif ← paste0('<tr.*?>', strrep('\\s*<td.*?>\\s*(.*?)\\s*</td>
   ', 6), '\\s*</tr>')
days ← regexc(motif, html_days)
```

On met en forme dans le format souhaité.

```
date ← dernier ← varp ← haut ← bas ← ouverture ← numeric(length
   (days))
for(i in 1:length(days))
{
        date[i] ← days[[i]][2]
        dernier[i] ← days[[i]][3]
        varp[i] ← days[[i]][4]
        haut[i] ← days[[i]][5]
        bas[i] ← days[[i]][6]
        ouverture[i] ← days[[i]][7]
}
```

3 R en finance

En modélisation financière, il est fréquent de rencontrer des processus X définis par une équation différentielle stochastique (une dynamique) de la forme :

$$X_t = x + \int_0^t \mu(s, X_s)ds + \int_0^t \sigma(s, X_s)dW_s, \quad t \in [0, T], \tag{1}$$

avec

- $x \in \mathbb{R}^q$ où $q \geq 1$ est la dimension du processus $(X_t)_{0 \leq t \leq T}$,
- $(W_t)_{t \geq 0}$ est un mouvement brownien en dimension d,
- $\mu : [0, T] \times \mathbb{R}^q \to \mathbb{R}^q$ est une fonction lipschtiz à croissance sous linéaire qui représente l'évolution moyenne locale,
- $\sigma : [0, T] \times \mathbb{R}^q \to \mathcal{M}_{q,d}(\mathbb{R})$ est une fonction lipschtiz à croissance sous linéaire qui représente le bruit local.

La loi d'un portefeuille peut dépendre d'un tel processus et, afin d'évaluer des statistiques sur ce portfeuille, comme des quantiles, ou évaluer des options, il est possible d'utiliser des méthodes de type Monte Carlo, c'est-à-dire simuler massivement le processus X.

On pourra se référer à [3] pour un approfondissement de ces méthodes.

3.1 Simulation de processus stochastique

Pour simuler une trajectoire du processus X caractérisé par l'équation différentielle stochastique (1), nous commençons par simuler un mouvement brownien W en dimension 1. Afin d'appliquer une méthode de Monte Carlo, il faudra simuler plusieurs copies de W. Nous allons en simuler $n \geq 1$. De plus, il nous faut discrétiser l'intervalle $[0, T]$. Nous le discrétisons en $\mathbf{T}_m^{\Delta t} := \{j\Delta t, 0 \leq j \leq m\}$ avec $\Delta t = T/m$ ce qui fait une discrétisation en $m + 1$ points. Nous allons simuler

$$W^i_{j\Delta t}, \quad 1 \leq i \leq n, 0 \leq j \leq m.$$

que nous allons représenter, dans **R** , par `W[i,j]` avec `W` une matrice de dimension $n \times m + 1$. L'une des propriétés du mouvement brownien est :

$$W_{(j+1)\Delta t} = W_{j\Delta t} + (W_{(j+1)\Delta t} - W_{j\Delta t}),$$

où $(W_{(j+1)\Delta t} - W_{j\Delta t})$ est une variable aléatoire indépendante de $W_{j\Delta t}$ de loi $\mathcal{N}(0, \Delta t)$. Cette propriété incrémentale permet de simuler, pour toutes les trajectoires en simultané, $\{W^i_{(j+1)\Delta t},\ 1 \leq i \leq n\}$ partant de $\{W_{j\Delta t},\ 1 \leq i \leq n\}$ en simulant n variables aléatoires indépendantes de loi $\mathcal{N}(0, \Delta t)$ qui sont les accroissements.

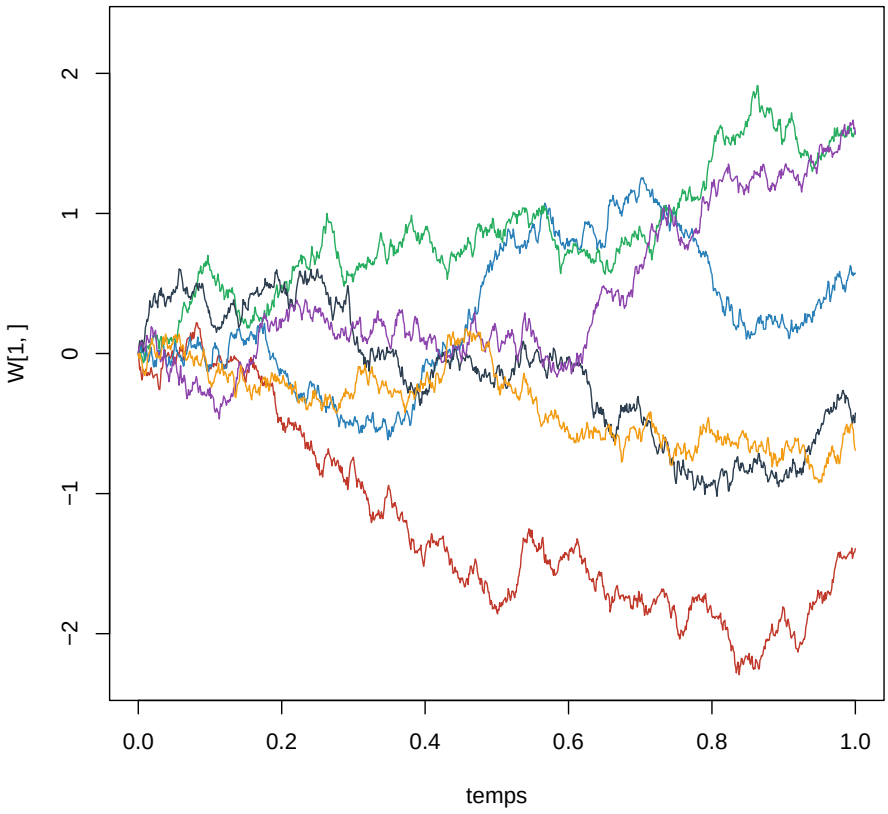
```
n ← 10^4
m ← 10^3
T ← 1

dt ← T/m; sdt ← sqrt(dt)

W ← matrix(0, n, m+1)
for(j in 1:m)
    W[, j+1] = W[, j] + rnorm(n, 0, sdt)
```

Puis nous pouvons afficher sommairement quelques trajectoires du mouvement brownien simulé.

```
couleurs = c("#c0392b", "#2980b9", "#27ae60", "#2c3e50", "#8e44ad",
    "#f39c12")
temps ← (0:m)*dt
nW_plot ← 6
plot(temps, W[1, ], ylim = c(-1,1)*max(abs(W[1:nW_plot, ])), type =
    "l", col = couleurs[1])
for(i in 2:nW_plot)
    lines(temps, W[i, ], col = couleurs[i])
```



Le processus $(X_t)_{0\leq t\leq T}$ caractérisé par l'équation (1) est markovien : si nous connaissons $\mathcal{L}(X_{t+\Delta t} \mid X_t)$ alors il est possible de simuler X sur $\mathbf{T}_m^{\Delta t}$ et, comme pour le mouvement brownien, pour toutes les trajectoires en simultané.

Prenons l'exemple du processus de Vašíček. Il est caractérisé par l'équation différentielle stochastique :

$$X_t = x + \int_0^t -a(X_s - \mu)ds + \int_0^t \sigma dW_s,$$

avec $a, \sigma > 0$ et $\mu \in \mathbb{R}$. Le processus se réécrit :

$$X_t = \mu + (x - \mu)e^{-at} + \sigma e^{-at} \int_0^t e^{as} dW_s.$$

En particulier, on a, en loi, pour tout $0 \leq j \leq m$,

$$X_{(j+1)\Delta t} = \mu + (X_{j\Delta t} - \mu)e^{-a\Delta t} + \sigma e^{-a\Delta t} \int_0^{\Delta t} e^{as} dW_s.$$

Ceci implique

$$X_{(j+1)\Delta t} \mid X_{j\Delta t} \sim \mathcal{N}\left(\mu + (X_{j\Delta t} - \mu)e^{-a\Delta t}, \frac{\sigma^2}{2a}(1 - e^{-2a\Delta t})\right).$$

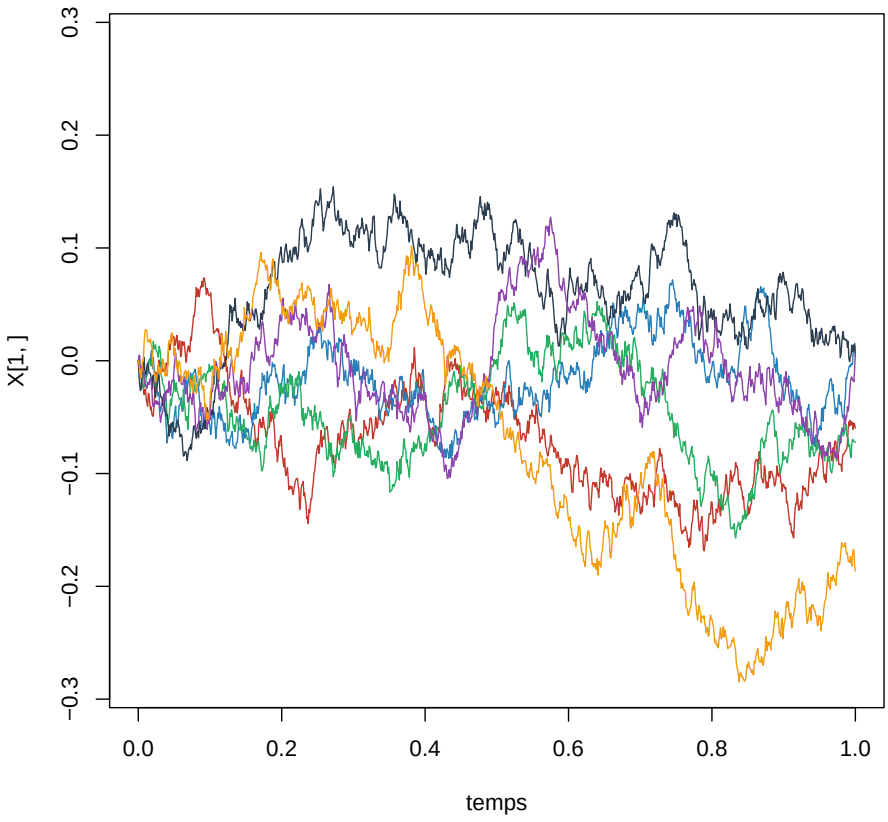
```
n ← 10^4
m ← 10^3
T ← 1
dt ← T/m

x ← 0; mu = 0
a ← 1; sigma ← 0.2

expmaDelta ← exp(−a∗dt)
s ← sqrt( (sigma^2/(2∗a))∗(1−expmaDelta^2) )

X ← matrix(x, n, m+1)
for(j in 1:m)
    X[, j+1] = rnorm(n, mu + (X[, j] − mu)∗expmaDelta, s)
```

En adaptant légèrement le code pour afficher quelques trajectoires du mouvement brownien, nous pouvons faire de même pour le processus de Vašíček.



Lorsqu'on ne connait pas $\mathcal{L}(X_{t+\Delta t} \mid X_t)$, on utilise un schéma numérique d'approximation de X. Le schéma d'Euler est le schéma le plus simple : il considère $\mu(s, X_s)$ et $\sigma(s, X_s)$ constants sur $[j\Delta t, (j+1)\Delta t[$ aux valeurs respectives $\mu(j\Delta t, X_{j\Delta t})$ et $\sigma(j\Delta t, X_{j\Delta t})$.

$$X^m_{(j+1)\Delta t} = X^m_{j\Delta t} + \mu(j\Delta t, X^m_{j\Delta t})\Delta t + \sigma(j\Delta t, X^m_{j\Delta t})(W_{(j+1)\Delta t} - W_{j\Delta t}).$$

On a donc en loi :

$$X^m_{(j+1)\Delta t} \mid X^m_{j\Delta t} \sim \mathcal{N}\left(X^m_{j\Delta t} + \mu(j\Delta t, X^m_{j\Delta t})\Delta t, \sigma(j\Delta t, X^m_{j\Delta t})^2 \Delta t\right).$$

Nous illustrons en appliquant le schéma d'Euler avec le processus

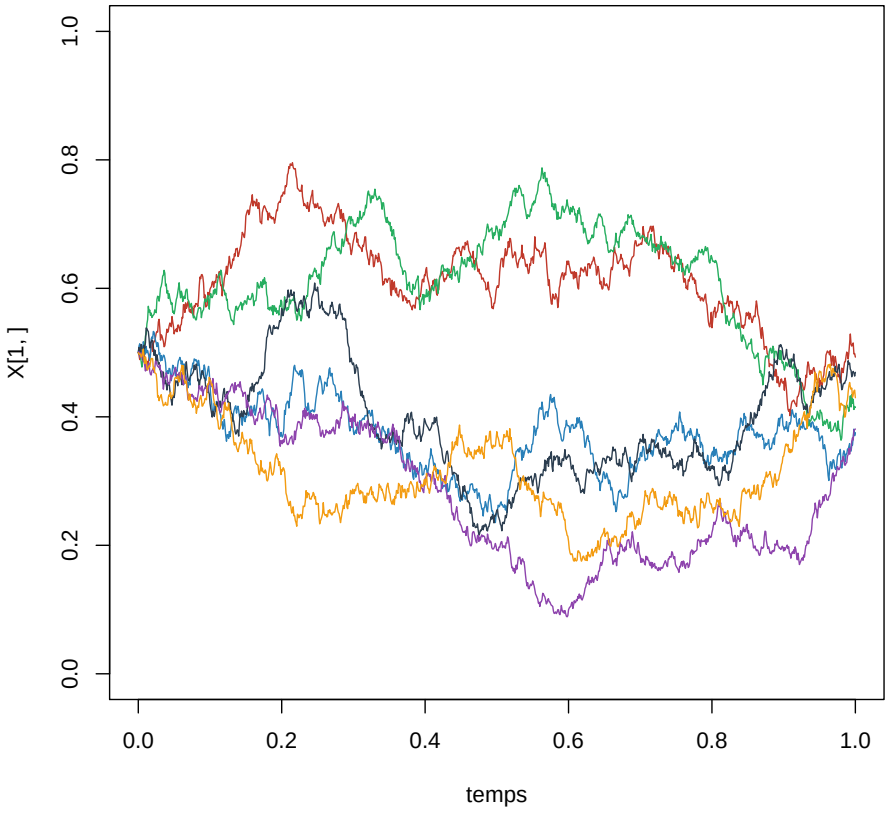
$$dX_t = -a(X_t - \mu)dt + \sigma\sqrt{X_t(1-X_t)}dW_t. \tag{2}$$

```
n ← 10^4
m ← 10^3
T ← 1
dt ← T/m

x ← 0.5; mu = 0.5
a ← 0.5; sigma ← 0.5

X ← matrix(x, n, m+1)
for(j in 1:m)
    X[, j+1] = rnorm(n, X[, j] - a*(X[, j] - mu)*dt, sigma*sqrt(dt*
       X[, j]*(1-X[, j])))
```

Illustrons quelques trajectoires.



Le processus ci-dessus prend ses valeurs dans $[0, 1]$ mais le schéma a une probabilité non nulle de sortir de l'intervalle, qui devient négligeable avec Δt suffisamment petit. Aux paramètres ci-dessus, mettre $\sigma = 0.75$ amène à ce genre de problèmes.

Il existe un second schéma, celui de Milstein, qui améliore l'approximation de l'intégrale stochastique et permet une convergence plus rapide de la trajectoire vers celle du processus, en supposant σ dérivable en espace. Le schéma est

$$\begin{aligned} X^n_{(j+1)\Delta t} = X^m_{j\Delta t} &+ \mu(j\Delta t, X^m_{j\Delta t})\Delta t + \sigma(j\Delta t, X^m_{j\Delta t})(W_{(j+1)\Delta t} - W_{j\Delta t}) \\ &+ \frac{1}{2}\sigma(j\Delta t, X^m_{j\Delta t})\partial_x\sigma(j\Delta t, X^m_{j\Delta t})\left((W_{(j+1)\Delta t} - W_{j\Delta t})^2 - \Delta t\right). \end{aligned}$$

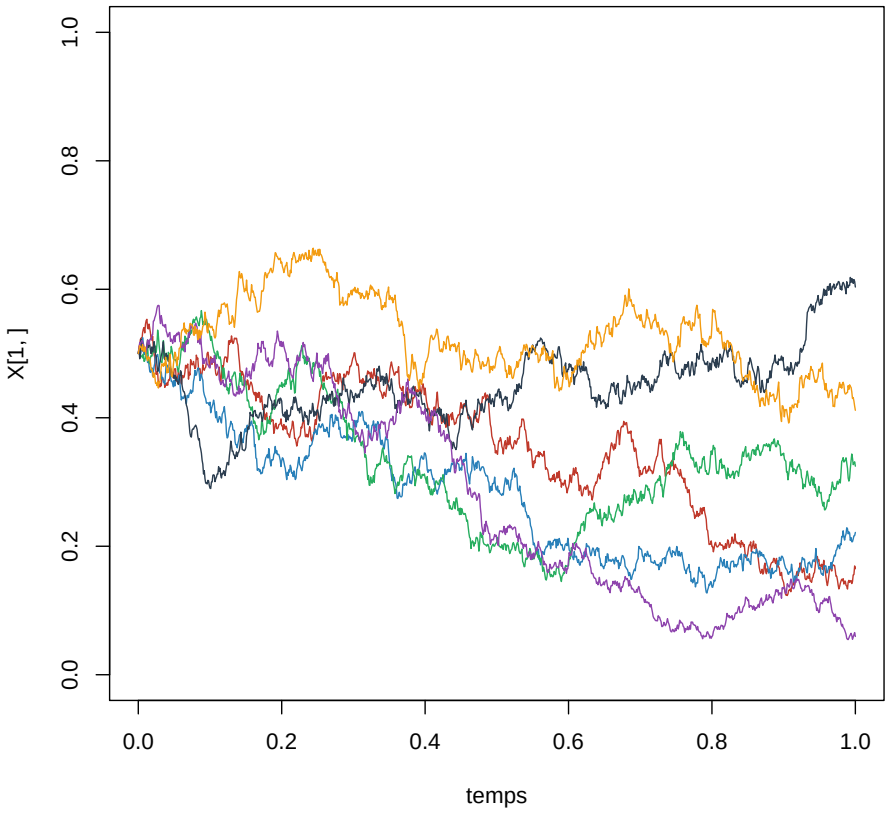
Nous illustrons en appliquant le schéma de Milstein au processus (2), le calcul de la dérivée donne

$$\left[\sigma\sqrt{x(1-x)}\right]' = \sigma\frac{1-2x}{2\sqrt{x(1-x)}}.$$

```
n ← 10^4
m ← 10^3
T ← 1
dt ← T/m; sdt ← sqrt(dt)

x ← 0.5; mu = 0.5
a ← 0.5; sigma ← 0.5

X ← matrix(x, n, m+1)
for(j in 1:m)
{
    Z ← rnorm(n, 0, sdt)
    Xj ← X[, j]
    S ← sigma*sqrt(Xj*(1-Xj))
    X[, j+1] ← Xj - a*(Xj - mu)*dt + sigma*sqrt(Xj*(1-Xj))*Z + 0.5*
       sigma^2*(1-2*Xj)*(Z^2 - dt)
}
```



Le processus obtenu par le schéma ci-dessus ne sort pas de l'intervalle $[0, 1]$ avec $\sigma = 0.75$ mais il pourra sortir avec $\sigma = 1$ (il serait obligatoire, dans ce cas, de réduire le pas).

3.2 Évaluation d'actif simple par Monte Carlo

Nous n'abordons pas les très importantes méthodes de réduction de variance, qui font partie d'un cours de Monte Carlo, on pourra se référer à [3].

Les méthodes de Monte Carlo s'appuient sur deux théorèmes fondamentaux :

- La **loi des grands nombres**, qui garantit la convergence de la moyenne empirique d'une suite de variables aléatoires i.i.d. intégrables ;
- Le **théorème de la limite centrale** qui permet d'obtenir une mesure de l'erreur de la moyenne empirique d'une suite de variables aléatoires i.d.d. de carré intégrable.

Pour estimer une quantité de la forme $\mathbb{E}(Z)$ avec Z une variable aléatoire, il suffit

- De simuler $(Z_i)_{1\leq i\leq n}$ des variables aléatoires i.i.d. de même loi que Z avec $n \geq 1$,
- De calculer l'estimateur $\hat{\mu}_n := \frac{1}{n}\sum_{i=1}^{n} Z_i$,
- De calculer son erreur potentielle, qui est $\hat{e}_n := q(1-\alpha/2)\hat{\sigma}(Z)/\sqrt{n}$ où $q(1-\alpha/2)$ est le quantile d'ordre α de la loi normale, qui peut être pris à 1,96 ou arrondi directement à 2 (environ 95% de confiance).

Ainsi, avec l'algorithme précédent, on obtient que $\mathbb{E}(Z)$ est égal à $\hat{\mu}_n \pm \hat{e}_n$ à une confiance de 95% (ou une confiance de $1-\alpha$ en fonction du niveau choisi).

Par exemple, pour estimer $\gamma := \mathbb{E}(-\log(X))$ avec $X \sim \mathcal{E}(1)$,

```r
n ← 10^6
Z ← log(rexp(n))
mu ← -mean(Z)
e ← 2*sd(Z)/sqrt(n)

cat(paste0("mu = ", round(mu, 6), " +/- ", round(e, 6), "\n"))
```

```
mu = -0.576789 ± 0.00256
```

Soit $(S_t)_{t\geq 0}$ un actif sous la probabilité risque-neure $\mathbb{Q}$ et $(r_t)_{t\geq 0}$ le taux d'intérêt sans risque sous $\mathbb{Q}$ (pouvant être constant, selon le modèle).
Pour estimer le prix d'une option européenne, par exemple d'un Call de prix d'exercice $K > 0$, on rappel que le prix $C(T)$ a la représentation

$$C(T) = \mathbb{E}^{\mathbb{Q}}\left[e^{-\int_0^T r_t dt}(S_T - K)^+\right].$$

Pour estimer $C(T)$, nous pouvons appliquer des simulation de Monte Carlo : il faut simuler S_T et $(r_t)_{t\geq 0}$.

4 R en assurance dommage

Cette section s'appuie sur [1], qu'on pourra utiliser pour approfondir.

4.1 Tarification

En assurance dommage, le coût des sinistres d'un assuré peut être représenté par une variable aléatoire de la forme

$$S = \sum_{n=1}^{N} Y_n,$$

où

- N est le nombre de sinistres, il s'agit d'une variable aléatoire entière éventuellement nulle ($N = 0 \Rightarrow S = 0$),
- $(Y_n)_{n\geq 1}$ est une suite de variables aléatoires positives et i.i.d. qui représente le coût des sinistres,
- N et $(Y_n)_{n\geq 0}$ sont indépendantes.

Sous ces hypothèses, on peut démontrer que (voir [4, Première formule de Wald])

$$\mathbb{E}(S) = \mathbb{E}(N)\mathbb{E}(Y). \tag{3}$$

On se placera toujours dans le cas où toutes les espérances sont finies. Cependant, il n'y a pas de raison que la loi de N (nombre de sinistres) et des $(X_n)_{n\geq 1}$ (coût des sinistres) soient identiques pour tous les assurés. Si on note $X^i = x^i \in \mathbb{R}^d$ les caractéristiques observables d'un assuré i, (3) devient

$$\mathbb{E}(S^i \mid X^i = x^i) = \mathbb{E}(N^i \mid X^i = x^i)\mathbb{E}(Y^i \mid X^i = x^i).$$

Nous estimons séparément $\mathbb{E}(N^i \mid X^i = x^i)$ et $\mathbb{E}(Y^i \mid X^i = x^i)$ grâce aux modèles linéaires généralisés. Nous le décrivons pour estimer $\mathbb{E}(N^i \mid X^i = x^i)$, c'est la même chose pour $\mathbb{E}(Y^i \mid X^i = x^i)$.

Dans un modèle linéaire généralisé, si on note $\mu_{N^i} := \mathbb{E}(N^i \mid X^i = x^i)$ et $\mu_{Y^i} := \mathbb{E}(Y^i \mid X^i = x^i)$, on suppose qu'il existe

- Une fonction $g_N : \mathbb{R} \mapsto \mathbb{R}$ inversible,
- Une constante $\beta_0^N \in \mathbb{R}$ et un vecteur $\beta^N \in \mathbb{R}^d$,

tels que

$$g_N(\mu_{N^i}) = \beta_0^N + x_i'\beta^N \left(\Longleftrightarrow \mu_{N^i} = g_N^{-1}(\beta_0^N + x_i'\beta^N)\right),$$

De plus, on suppose que la loi des N^i appartient à une famille $(\mathbb{P}_\theta)$ où $N^i \sim \mathbb{P}_{\mu_{N^i}}$, c'est à dire paramétrée (ou reparamétrée) par la moyenne. De manière plus générale, on peut introduire un facteur de déviance (qui est relié à la dispersion) commun et d'avoir $N^i \sim \mathbb{P}^\phi_{\mu_{N^i}}$ où ϕ pourra être à estimer (il est l'équivalent de la variance du bruit dans le modèle linéaire). La fonction g est choisie, et tout ceci posé, il est possible d'estimer les paramètres du modèle : β_0, β (et ϕ s'il y a lieu) par maximum de vraisemblance.

Nous commençons par voir sur **R** les modèles linéaires gaussiens, qui sont un cas particulier des modèles linéaires généralises, avec g l'identité $(x \mapsto x)$, et $\mathbb{P}^\phi_{\mu_i} = \mathcal{N}(\mu_i, \phi)$.

La fonction de **R** qui permet d'estimer les paramètres est la fonction **glm**.

Nous allons commencer par voir les modèles linéaires

4.1.1 Modèle linéaire avec R

La fonction de **R** qui permet d'estimer les paramètres d'un modèle linéaire est **lm**. Le moyen le plus simple de l'utiliser se fait avec

- Une *formule* au sens de **R** qui implique le nom des colonnes d'une data.frame ;
- De fournir la data.frame des données via l'argument `data`.

Dans **R** , il y a des données par défaut, par exemple les données `cars`.

```
cars
```

```
   speed dist speed2
1      4    2     16
2      4   10     16
3      7    4     49
4      7   22     49
5      8   16     64
6      9   10     81
        ...
```

Pour régresser la distance de freinage `dist` contre la vitesse `speed`, la formule est `dist ~ speed`.

```
lm(dist ~ speed, data = cars)
```

```
Call:
lm(formula = dist ~ speed, data = cars)
```

```
Coefficients:
(Intercept)        speed
    −17.579        3.932
```

`(Intercept)` est la constance β_0 de la régression. Pour régresser sur plusieurs variables, on sépare les régresseurs par `+` dans la formule.

```
cars$speed2 ← cars$speed^2
lm(dist ~ speed + speed2, data = cars)
```

```
Call:
lm(formula = dist ~ speed + speed2, data = cars)

Coefficients:
(Intercept)        speed       speed2
    2.47014      0.91329      0.09996
```

Lors qu'il y a beaucoup de variables dans la régression, et qu'on souhaite le faire sur toutes, on utilise le point `.` pour les représenter.

```
cars$speed2 ← cars$speed^2
lm(dist ~ . , data = cars)
```

```
Call:
lm(formula = dist ~ ., data = cars)

Coefficients:
(Intercept)        speed       speed2
    2.47014      0.91329      0.09996
```

Pour retirer des variables on utilise le signe moins −. La constante de régression s'appelle `1`, il est possible de la retirer.

```
lm(dist ~ . −1, data = cars)
```

```
Call:
lm(formula = dist ~ . − 1, data = cars)

Coefficients:
  speed   speed2
1.23903  0.09014
```

La fonction `lm` renvoie une liste. On peut connaître ses éléments en appliquant la fonction `names(lm(dist ~ speed, data = cars))`. La fonction `summary` permet de récupérer les tests statistiques des régresseurs et diverses information avec `summary(lm(dist ~ speed, data = cars))`. C'est également une liste dont nous pouvons connaître les éléments avec `names`.

4.1.2 Modèle linéaire généralisé avec R

La fonction de R qui permet d'estimer les paramètres d'un modèle linéaire généralisé est la fonction `glm`. Elle fonctionne de la même manière que la fonction `lm`, où il faut préciser en plus

- la fonction de lien g,
- la famille de loi $(\mathbb{P}_\theta)$.

Ces deux éléments sont donnés ensemble via un objet `family`. Pour retrouver le modèle linéaire, le cas particulier avec une famille normale et une fonction de lien qui est l'identité, on utilise `gaussian(link="identity")`.

```
glm(dist ~ speed, family = gaussian(link="identity"), data = cars)
```

```
Call:  glm(formula = dist ~ speed, family = gaussian(link = "
   identity"), data = cars)

Coefficients:
(Intercept)        speed
    -17.579        3.932

Degrees of Freedom: 49 Total (i.e. Null);  48 Residual
Null Deviance:        32540
Residual Deviance: 11350     AIC: 419.2
```

Ci-dessous, nous listons des exemples de familles de loi avec leur fonction de lien canonique. On pourra voir l'aide de aide avec `help(family)`.

Loi	Lien g	Code R	Représentation
$\mathcal{B}(\mu)$	`"logit"`	`binomial("logit")`	$\mu = \frac{\exp(\beta_0+x'\beta)}{1+\exp(\beta_0+x'\beta)}$
$\mathcal{P}(\mu)$	`"log"`	`poisson("log")`	$\mu = \exp(\beta_0 + x'\beta)$
$\mathcal{N}(\mu,\sigma^2)$	`"identity"`	`gaussian("identity")`	$\mu = \beta_0 + x'\beta$
$\mathcal{G}(\alpha,b)$	`"inverse"`	`Gamma("inverse")`	$\frac{\alpha}{b} = \frac{1}{\beta_0+x'\beta}$

On pourra changer la fonction de lien, la plupart acceptent `"identity"` ou la fonction `"log"`, ou des fonctions spécifiques, comme `"sqrt"` pour la loi de Poisson. Il sera préférable d'utiliser une fonction de lien telle que g^{-1} ait pour image l'ensemble de définition de la moyenne de la loi.

Nous faisons un exemple simulé exact, sur des nombres de sinistres, pour bien comprendre comment cela fonctionne et ce que cela représente.

```
beta0 ← -2
beta ← c(1, -0.4)

n ← 300
set.seed(0) #Pour fixer l'aléa et reproduire
X ← data.frame(X1 = rgamma(n, 5, 5), X2 = rbinom(n, 1, 0.2))

mu ← exp(beta0 + as.matrix(X) %*% as.matrix(beta))
X$freq ← rpois(n, mu)
```

```
(R ← glm(freq ~ ., family = poisson("log"), data = X))
```

```
Call:  glm(formula = freq ~ ., family = poisson("log"), data = X)

Coefficients:
(Intercept)           X1           X2
    −2.1506       1.0345       −0.7048

Degrees of Freedom: 299 Total (i.e. Null);  297 Residual
Null Deviance:        258.7
Residual Deviance: 234.2     AIC: 406.3
```

Puis nous pouvons afficher les statistiques associées aux coefficients de la régression.

```
summary(R)
```

```
Call:
glm(formula = freq ~ ., family = poisson("log"), data = X)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
−1.3036  −0.7712  −0.6377   0.3320   2.6235

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  −2.1506     0.2865  −7.506  6.1e−14 ***
X1            1.0345     0.2291   4.516  6.3e−06 ***
X2           −0.7048     0.3098  −2.275   0.0229 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for poisson family taken to be 1)

    Null deviance: 258.75  on 299  degrees of freedom
Residual deviance: 234.22  on 297  degrees of freedom
AIC: 406.34

Number of Fisher Scoring iterations: 6
```

Dans cet exemple, si un assuré a $X1 = 2$ et $X2 = 1$, le nombre moyen de sinistre estimé est :

```
exp(t(c(1, 2, 1)) %*% as.matrix(R$coefficient))
```

```
          [,1]
[1,] 0.4554772
```

ou plus simplement, avec la fonction **predict**

```
exp(predict(R, data.frame(X1 = 2, X2 = 1)))
```

```
          1
0.4554772
```

Il faudrait faire de même sur les coûts et multiplier ensemble pour obtenir la prime pure. Pour la loi binomiale négative, il y a la fonction dédiée **glm.nb** du paquet `MASS` qui est livré avec **R** .

4.2 Provisionnement

En assurance dommage, la prime est payée à la signature mais le paiement des sinistres éventuels à lieu dans l'année qui suit. Parfois, un sinistre qui a eu lieu n'est pas connu de l'assurance ou son montant n'est pas encore déterminé, et le coût total du sinistre est quelquefois connu que plusieurs années après. L'assureur est tenu d'estimer et de provisionner ce coût probable.
Nous prendrons l'exemple du modèle de Chain Ladder - Mack.

4.2.1 Triangles de liquation

Partant de données indviduels (l'évolution du coût de chaque sinistre), l'assureur peut construire deux triangle. On observe l'évolution du coût des sinistres sur n années. Le triangle incrémental :

i,j	1	$\cdots$	n
1	$X_{1,1}$	$\cdots$	$X_{1,n}$
$\vdots$	$\vdots$	$\ddots$	
n	$X_{n,1}$		

où $X_{i,j}$ est la variable aléatoire qui correspond au coût des sinistres survenus l'année i pour l'année de développement j.
Nous aurons besoin du triangle de charges cumulées, on définit pour $1 \leq i,j \leq n$,

$$C_{i,j} = \sum_{k=1}^{j} X_{i,k}.$$

i,j	1	$\cdots$	n
1	$C_{1,1}$	$\cdots$	$C_{1,n}$
$\vdots$	$\vdots$	$\ddots$	
n	$C_{n,1}$		

Nous pouvons passer de C à X via la multiplication d'une matrice simple, ce qui permettra d'avoir un code de passage efficace. Si on multiplie X par la matrice

$$XtoC = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Si `X` est le triangle incrémental, le code de passage est simplement :

```
XtoC ← upper.tri(X, TRUE)
C ← X %*% XtoC
```

où `upper.tri` est une fonction qui renvoie une matrice de booléens avec `TRUE` sur la partie supérieure à la diagonale.

Pour passer du triangle des coûts cumulés C à X, il suffit de multiplier à droite par l'inverse de $XtoC$. Son inverse est

```
solve(XtoC)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1   -1    0    0
[2,]    0    1   -1    0
[3,]    0    0    1   -1
[4,]    0    0    0    1
```

On peut la définir simplement.

```
n ← nrow(C)
CtoX ← diag(n)
CtoX[col(C) == row(C) + 1] ← -1
```

Puis le passage de C à X est simplement

```
X ← C %*% CtoX
```

On pourra importer les données d'exemple suivantes pour illustrer ci-après.

```
C ← read.table("https://nicolasbaradel.fr/enseignement/ressources/R
   /donneesMACK.txt", sep = "&")
n ← nrow(C)
```

4.2.2 Chain Ladder - Mack

Sous ce modèle, on suppose qu'en moyenne, l'évolution entre une année de développement j et $j+1$ est $\mathbb{E}(C_{i,j+1} \mid \texttt{passé}) = f_j C_{i,j}$. Avec les autres hypothèses, on en déduit (voir [1])

$$\hat{f}_j := \frac{\sum_{i=1}^{n-j} C_{i,j+1}}{\sum_{i=1}^{n-j} C_{i,j}}.$$

Nous pouvons les calculer simplement.

```
f_ ← function(C)
{
    f ← numeric(n-1)
    for(j in 1:(n-1))
        f[j] ← sum(C[1:(n-j), j + 1])/sum(C[1:(n-j), j])
    return(f)
}
f ← f_(C)
```

Grâce à ces facteurs de développement, nous pouvons ensuite développer le triangle des coûts de sinistres cumulés. On utilise la relation $\hat{C}_{i,j+1} = \hat{f}_j \hat{C}_{i,j}$ en partant de la diagonale.

```
hatC_ ← function(C, f)
{
    for(j in 1:(n−1))
        C[(n−j+1):n, j+1] ← f[j]∗C[(n−j+1):n, j]
    return(C)
}
C ← hat_C(C)
```

La provision associée à chaque année i est la différence entre le coût total estimé après tous les développement $\hat{C}_{i,n}$ et le coût actuel $C_{i,n-i+1}$, et la provision totale est la somme des provisions associée à chaque année.

```
R_ ← function(C)
        return(C[, n] − rev(C[row(C) + col(C) == n + 1]))
R ← R_(C)
```

Avec les données, la réserve totale est La réserve totale est $\hat{R} = 18680848$. Calculer la réserve n'est pas suffisant : il ne s'agit que de la moyenne estimée du coût des sinistres tardifs.
Le modèle de Mack suppose que $Var(C_{i,j+1} \mid \texttt{passé}) = \sigma_j^2 C_{i,j}$, et les estimateurs sans biais

$$\hat{\sigma}_j^2 := \frac{1}{n-j-1} \sum_{i=1}^{n-j} C_{i,j} \left(\frac{C_{i,j+1}}{C_{i,j}} - \hat{f}_j \right)^2 .$$

En R , cela se traduit par

```
s2_ ← function(C, f)
{
    s2 ← numeric(n−1)
    for(j in 1:(n−2))
        s2[j] ← sum( C[1:(n−j), j] ∗ (C[1:(n−j), j + 1]/C[1:(n−j),
           j] − f[j])^2 )/(n − j − 1)
    s2[n−1] ← min(s2[n−2]^2/s2[n−3], s2[n−3], s2[n−2])
    return(s2)
}
s2 ← s2_(C, f)
```

De là, on peut en déduire la volatilité sur le montant des réserves, incluant l'incertitude d'estimation des paramètres (voir [1, Section 4.2]) ou appliquer la méthode du boostrap pour estimer la distribution des réserves (voir [1, Section 6]).
On peut également utiliser le paquet `ChainLadder` qui possède les fonctions pour calculer directement les différentes quantités du modèle de Chain Ladder Mack.

5 R en assurance vie

5.1 Rentes viagères et capital décès

Le paquet qui permet de faire les calculs habituels en assurance-vie est `lifecontingencies`. Avec celui-ci il est possible de construire une table de mortalité, qui est un **objet** de R . L'objet est un `lifetable` et se créé avec la fonction `new`.

```
(TM ← new("lifetable", x=0:120, lx=round(100000*exp(-25*((0:120)
   /120)^9)), name="exemple"))
```

```
Life table exemple

      x     lx        px         ex
1     0 100000 1.0000000 78.9673600
                       ...
110 109      3 0.3333333  0.3333333
```

Ci-dessus, **lx** qui est donné dans la création de la table est le nombre de survivant à l'âge x, tandis que **px** est la probabilité de survivre à 1 an sachant que l'individu a l'âge x. Enfin, **ex** est l'espérance de vie résiduelle à l'âge x. Nous pouvons récupérer les noms des éléments.

```
slotNames(TM)
```

```
[1] "x"    "lx"   "name"
```

```
TM@lx
```

```
  [1] 100000 100000 100000 100000 100000 100000 100000 100000
     100000 100000
                       ...
[111]      1
```

De cette table, il est possible de calculer

- ${}_tp_x = \frac{l_{x+t}}{l_x} =$ **pxt(TM, x, t)**
- ${}_tq_x = 1 - {}_tp_x =$ **qxt(TM, x, t)**
- $e_x = \sum_{x=1}^{\infty} {}_tp_x =$ **exn(TM, x)**

En actuariat, pour le calcul de rente ou, de manière générale, de flux différés, il faut pouvoir intégrer un taux d'actualisation $\frac{1}{1+r}$ où r est le taux d'intérêt à 1 an.

Pour cela, nous créons un objet **actuarialtable** de la même manière qu'un **lifetable** sauf qu'on y ajoute le taux d'intérêt via le membre **interest**.

```
(TM ← new("actuarialtable", x=0:120, lx=round(100000*exp
   (-25*((0:120)/120)^9)), name="exemple", interest = 0.02))
```

```
Actuarial table  exemple interest rate  2 %

      x     lx           Dx           Nx          Cx           Mx
                  Rx
1     0 100000 1.000000e+05 4.029114e+06   0.0000000 2.099777e+04
   1.629732e+06
                       ...
111 110      1 1.132351e-01 1.132351e-01   0.1110148 1.110148e-01
   1.110148e-01
```

Toutes les quantités `Dx`, etc, sont les grandeurs actuarielles habituelles. Elles sont calculées à la volée, seul le taux d'intérêt a été ajouté à l'objet.

```
slotNames(TM)
```

```
[1] "interest" "x"        "lx"       "name"
```

De cette table actuarielle, il est possible de calculer les primes pures unitaires des rentes associées.

- $\ddot{a}_x = \sum_{t=0}^{\infty} \frac{{}_tp_x}{(1+r)^t}$ = `axn(TM, x)`
- $a_x = \sum_{t=1}^{\infty} \frac{{}_tp_x}{(1+r)^t}$ = `axn(TM, x, payment = "immediate")`
- ${}_{m|n}\ddot{a}_x = \sum_{t=m}^{m+n-1} \frac{{}_tp_x}{(1+r)^t}$ = `axn(TM, x, m=m, n=n)`
- ${}_{m|n}a_x = \sum_{t=m+1}^{m+n} \frac{{}_tp_x}{(1+r)^t}$ = `axn(TM, x, m=m, n=n, payment = "immediate")`

Sur la table actuarielle `TM`, nous pouvons calculer la prime pure unitaire du versement d'un capital en cas de décès.

- $A_x = \sum_{t=0}^{\infty} \frac{{}_tp_x \times q_{x+t}}{(1+r)^t}$ = `Axn(TM, x)`
- ${}_{m|n}A_x = \sum_{t=m}^{m+n-1} \frac{{}_tp_x \times q_{x+t}}{(1+r)^t}$ = `Axn(TM, x, m=m, n=n)`

Si l'assuré paie une prime P annuelle pour une assurance décès unitaire sur la vie entière, qu'il versera jusqu'au décès, alors la prime pure de ce qu'il versera à l'assureur est $\ddot{a}_x$ tandis que la prime pure de sa garantie est A_x. La prime d'équilibre est $\frac{A_x}{\ddot{a}_x}$ qui s'obtient avec `Axn(TM, x)/axn(TM, x)`. Nous pouvons en déduire aussi les provisions mathématiques. Par exemple, si l'assuré signe une garantie vie entière en cas de décès à l'âge x pour un capital K, et verse une rente vie entière d'un montant annuel P, on a $P\ddot{a}_x = KA_x$. Puis, s'il est toujours en vie après t années, la provision mathématique est ${}_tV_x = KA_{x+t} - P\ddot{a}_{x+t}$, c'est-à-dire `K*Axn(x+t)- P*axn(x+t)`.

5.2 Estimation de tables de mortalité

On observe les décès une population du même sexe et homogène. Chaque individu $1 \leq i \leq n$ a un âge différent au départ et on observe son éventuel décès en T_i. En revanche, il se peut qu'il sorte de l'observation en C_i (la censure). Ainsi, on observe le couple $(Y_i, \delta_i) = (\min(T_i, C_i), \mathbf{1}_{\{T_i \leq C_i\}})$ et on suppose généralement l'indépendance entre T_i et C_i.
Estimer une table de mortalité revient à estimer les

$$q_x = \mathbb{P}\left(T \in]x, x+1] \mid T > x\right), \quad x \geq 0.$$

Dans un premier temps, nous les estimons indépendamment les uns des autres, ce que nous appelons les taux bruts.

5.2.1 Taux bruts

Certains assurés arrivent dans le portefeuille à un âge non entier et d'autres quittent celui-ci sans observation de décès à un âge non entier.
Si nous faisons l'hypothèse approximative que les décès, dans un intervalle $]x, x+1]$, ont une force de sortie $\mu_{x+r} = \mu_x$ constante pour $r \in [0, 1[$, c'est à dire que, pour $t \in [0, 1]$

$${}_tp_x = e^{-\int_x^{x+t} \mu_s ds} = e^{-t\mu_x} = [e^{-\mu_x}]^t = p_x^t,$$

nous pouvons intégrer la troncature facilement (observation d'un assuré à partir d'un âge non entier). Ainsi, si on note

- τ_i la durée de troncature pour l'observation (l'assurée est observé à partir de l'âge $x + \tau_i$,
- δ_i l'observation d'un décès,
- Y_i la date de fin de l'observation (décès, sortie de l'intervalle, ou arrivée au bout de l'intervalle, dans ce dernier cas $Y_i = 1$),

dans ce cas, l'estimateur du maximum de vraisemblance de μ_x est

$$\hat{\mu}_x = \frac{\sum_{i=1}^n \delta_i}{\sum_{i=1}^n Y_i - \tau_i}$$

où $Y_i - \tau_i$ est la durée effective d'observation dans l'intervalle $[x, x+1[$.
Alors l'estimateur de q_x est

$$\hat{q}_x = 1 - \exp\left(-\frac{\sum_{i=1}^n \delta_i}{\sum_{i=1}^n Y_i - \tau_i}\right).$$

Nous faisons un petit exemple où nous simulons la troncature, censure, le décès, et calculons $\hat{q}_x$ pour x fixé.

```
qx ← 0.1
mux ← -log(1-qx)
n ← 10^6

#Date de début de l'observation dans l'âge x
tau ← rbinom(n, 1, 0.2)*runif(n)
T ← tau + rexp(n, mux)
#Si Z vaut 0, l'observation est censurée avant x+1
Z ← rbinom(n, 1, 0.8)
#Fin de l'observation en x+1 ou appliquation d'une censure
C ←  Z + (1-Z)*runif(n, tau, 1)
Y ← pmin(T, C)
delta ←  T <= C

(hat.qx ← 1-exp(-sum(delta)/sum(Y-tau)))
```

```
[1] 0.1000338
```

Nous pouvons aussi utiliser l'estimateur suivant, très proche, qui coïncide avec l'estimateur naturel sans troncature / censure.

```
#Autre version, qui coïncide avec le cas sans censure (qx = dx/nx)
Ybis ← Y
Ybis[T <= C] ← 1

(hat.qx ← sum(delta)/sum(Ybis-tau))
```

```
[1] 0.100055
```

5.2.2 Lissage Whittaker-Henderson

Nous savons que $x \mapsto q_x$ est une fonction croissante et régulière. En revanche, bien que l'estimateur $\hat{q}_x$ soit sans biais, la fonction $x \mapsto \hat{q}_x$ n'est en général pas croissante et peut être peu régulière là où il y a peu d'observations. Nous pouvons appliquer une méthode de lissage, comme celle de Whittaker-Henderson.

Pour pouvoir appliquer la méthode, construisons-nous une table de taux brut fictive. Dans un premier temps nous construisons des l_x fictifs desquels nous en déduisons des q_x.

```
lx ← 10000*exp(-25*((0:105)/120)^9)
qx ← 1-lx[-1]/lx[-length(lx)]
m ← length(qx)
```

Nous simulons ensuite des observations de mortalité suivant cette table et en déduisons les taux brut $\hat{q}_x$.

```
nx ← numeric(m)
dx ← hat.qx ← numeric(m)

set.seed(0)
nx[1] ← 10000
dx[1] ← rbinom(1, nx[1], qx[1])
for(i in 2:m)
{
    nx[i] ← nx[i-1] - dx[i-1]
    dx[i] ← rbinom(1, nx[i], qx[i])
}
hat.qx ← dx/nx
```

Le lissage de Whittaker-Henderson consiste à trouver $x \mapsto \tilde{q}_x$ qui est un arbitrage entre la fidélité aux taux bruts $x \mapsto \hat{q}_x$ et la régularité. Soit m l'âge maximum, la fidélité aux taux bruts est définie par la quantité :

$$F(\tilde{q}_x) = \sum_{x=0}^{m} w_x(\tilde{q}_x - \hat{q}_x)^2 = (\tilde{q} - \hat{q})'W(\tilde{q} - \hat{q}),$$

où $\tilde{q} = (\tilde{q}_x)_{0\leq x\leq m}$, $\hat{q} = (\hat{q}_x)_{0\leq x\leq m}$ et W est une matrice diagonale de diagonale $(w_x)_{0\leq x\leq m}$. Les poids sont généralement l'inverse de la variance de l'estimateur brut, c'est à dire

$$w_x := \frac{\sum_{i=1}^{n} E_x^i}{\hat{q}_x(1-\hat{q}_x)},$$

ou, s'il y a des problèmes numériques liés à $\hat{q}_x$ (s'annule ou très instable car peu de données),

$$w_x := \sum_{i=1}^{n} E_x^i.$$

```
W ← diag(nx)
```

La régularité d'ordre 2 est mesurée par

$$S(\tilde{q}) := \sum_{x=0}^{m-2} (\tilde{q}_{x+2} - 2\tilde{q}_{x+1} + \tilde{q}_x)^2 = \tilde{q}'K'K\tilde{q},$$

avec K une matrice $m-2 \times m$ définie par

$$K := \begin{pmatrix} 1 & -2 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & -2 & 1 & 0 & \cdots & 0 \\ & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 0 & 1 & -2 & 1 \end{pmatrix}$$

```
k ← c(1, −2, 1)
K ← matrix(0, m−2, m)
for(i in 1:(m−2))
    K[i, i:(i+2)] ← k
```

Le lissage consiste à minimiser la quantité

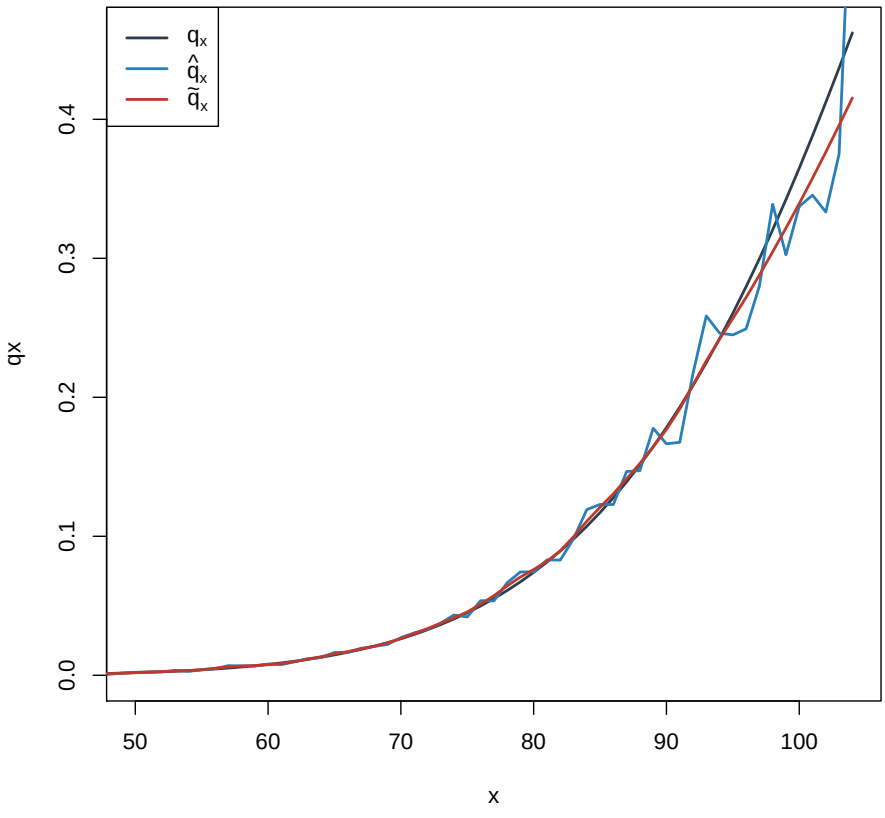
$$F(\tilde{q}) + hS(\tilde{q}),$$

où h est un paramètre à choisir : l'arbitrage entre fidélité et régularité. La solution est explicite est

$$\tilde{q} = (W + hK'K)^{-1}W\hat{q}.$$

```
h ← 10000
tilde.qx ← solve(W + h∗t(K)%∗%K, W%∗%hat.qx)
```

nous pouvons afficher les véritables $x \mapsto q_x$, avec $x \mapsto \hat{q}_x$ et $x \mapsto \tilde{q}_x$.



6 Utiliser du code compilé C pour accélérer R

Pour compiler une fonction R à partir d'un code C sur Windows, il faut installer Rtools : `https://cran.r-project.org/bin/windows/Rtools/` qui contient, en particulier, le compilateur et l'accès direct aux bibliothèques de R. Pour l'ajouter dans le path, on exécute la ligne suivante dans R.

```
writeLines('PATH="${RTOOLS40_HOME}\\usr\\bin;${PATH}"', con = "~/
   .Renviron")
```

Il faut ensuite ajouter R au *PATH*. Pour accéder au menu, on écrit *path* ou *variables d'environnement* dans le menu rechercher de Windows et on accède à une fenêtre où nous cliquons sur *Variables d'environnement...*. On édite *Path* et on y ajoute le chemin du dossier qui contient *R.exe* en 64 bits qui peut être, par exemple, `C:\Program Files\R\R-4.0.5\bin\x64`.

Les préparatifs sont terminés, pour s'assurer que tout cela fonctionne, on ouvre un Terminal dans RCode ou via Windows (*cmd* dans le menu recherché, ou depuis une fenêtre de l'explorateur Windows via le champs du dossier). Puis on écrit R et on éxécute. Cela doit lancer R dans le Terminal, il s'agit de celui dans le dossier renseigné dans le path, qu'on pourra changer ultérieurement si besoin. Cela permettra d'exécuter R en ligne de commande pour compiler une fonction ou un package.

6.1 Initiation avec `.C()`

Nous présentons la compilation d'une fonction C par un exemple. Dans un fichier *fonction.c*, nous écrivons le code C suivant, détaillé ci-après.

```
void somme(double * x, double * y, double * z)
{
    *z = *x + *y;
}
```

Pour compiler cette fonction, nous pouvons faire, au choix

- Dans un Terminal, après avoir fait pointé le chemin vers le dossier qui contient le fichier *fonction.c*, pour le compiler, il suffit de d'exécuter `R CMD SHLIB fonction.c`
- Depuis R, la fonction `system` permet d'exécuter une commande via le Terminal.

Avec `system`, si la fonction est dans le projet dans un dossier *C*, il suffit d'exécuter dans R

```
system("R CMD SHLIB C/fonction.c")
```

```
[1] 0
```

La valeur de retour `0` traduit un bon déroulement. Un fichier *fonction.dll* (Windows) ou *fonction.so* (Mac) apparaît.

On charge la bibliothèque de fonction compilées avec la fonction `dyn.load`.

```
dyn.load("C/fonction.dll")
```

Nous sommes prêts à appeler notre fonction dans R. Cela se fait avec `.C`. Le premier argument est le nom de la fonction, et ensuite on fournit, en les nommants, les autres arguments.

```
.C("somme", x=1, y=pi, z=0)
```

```
$x
[1] 1

$y
[1] 3.141593

$z
[1] 4.141593
```

La fonction renvoie une liste avec la (nouvelle) valeur de chacune des variables en argument de la fonction C. Nous pouvons encapsuler l'appel de `.C` avec une fonction R.

```
somme ← function(x, y)
    return(.C("somme", x = x, y = y, z = 0)$z)

somme(1, pi)
```

```
[1] 4.141593
```

En revanche, il faut être très prudent sur le type de l'objet envoyé. Si la fonction C attend un *double* et qu'on lui envoie un autre type, cela peut faire planter la fonction et R tout entier qui devra redémarrer, ou alors renvoyer un résultat aléatoire, comme le montre l'exemple suivant.

```
somme(1L, 0)
```

```
[1] 1.188318e-312
```

Pour protéger la fonction, on pourra forcer la conversion en double avec `as.double`.

```
somme ← function(x, y)
    return(.C("somme", x = as.double(x), y = as.double(y), z = 0)$z
       )

somme(1L, 0)
```

```
[1] 1
```

Si on souhaite compiler à nouveau le fichier *C/fonction.c*, il faut au préalable décharger la bibliothèque de fonctions, sinon le compilateur ne pourra pas l'écraser et retournera une erreur.

```
dyn.unload("C/fonction.dll")
```

Expliquons la fonction `somme` écrite en C. Ici, le type de retour est **`void`**, cela veut dire qu'elle ne renvoie rien (pas de **`return`**. Le type des variables doit être précisé en C, il s'agit dans notre exemple de **`double`**. L'étoile qui suit est obligatoire dans la création d'une fonction C pour R via cette méthode. Cela indique que la mémoire de la variable sera directement modifiée (mais R fait une copie de la variable envoyée, la variable R sera inchangée). En conséquence, pour modifier ou appeler la variable `x` dans le corps de la fonction, on l'appel avec `*x`.

Un exemple plus intéressant est de pouvoir par exemple simuler la suite i.i.d. de $(S_i)_{1\leq i\leq n}$ de variables aléatoires définies par

$$S_i = \sum_{k=1}^{N_i} X_k^i, \quad 1 \leq i \leq n,$$

où $(N_i)_{1\leq i\leq n}$ est une suite i.i.d. de variables aléatoires à valeurs dans $\mathbb{N}$ et $(X_k^i)_{k\geq 1, 1\leq i\leq n}$ sont des variables aléatoires i.d.d. à valeurs dans $\mathbb{R}$ et indépendantes des (N_i). Par exemple, si $(N_i) \overset{i.i.d.}{\sim} \mathcal{P}(\lambda)$ et $(X_k^i) \overset{i.i.d.}{\sim} \mathcal{LN}(\mu, \sigma^2)$. Ce qui est passé en C via `double * x` peut être un vecteur. Dans ce cas, on accède aux éléments via `[]` comme en R, à la différence que l'indexation commence à `0`. Et `*x` est équivalent à `x[0]`. Il faut toutefois passer la taille des vecteurs en argument, il n'est pas possible de la connaître dans le code C sinon.

Pour simuler les variables aléatoires en C, nous pouvons appeler les fonctions de simulation de R codées en C, il suffit d'ajouter les bons `#include` en préambule.

```
#include <R.h>
#include <Rmath.h>

void rsum(int * n, double * lambda, double * mu, double* sigma,
   double * S)
{
    int N;
    GetRNGstate();
    for(int i = 0; i != *n ; ++i)
    {
        N = rpois(*lambda);
        for(int k = 0; k != N ; k++)
            S[i] += rlnorm(*mu, *sigma);
    }
    PutRNGstate();
}
```

Ci-dessus, nous pouvons accéder aux fonctions C natives de R pour les simulations. Ce sont les mêmes que celles de R, à la différence qu'elles ne demandent pas le nombre de simulations (elles n'en renvoient qu'une). Toutes les fonctions de `#include <Rmath.h>` sont consultables en téléchargeant le code source de R et en allant dans `src/nmath`. Dans le code ci-dessus, `S` est un vecteur de taille `n` et doit être entré comme tel dans la fonction C via l'appel de R . Enfin, `GetRNGstate` doit être appelé pour s'assurer que le générateur est sur la graine de R et `PutRNGstate` à la fin pour transmettre l'état à R : sans cela, on sera confronté à des bugs (simulations nulles ou identiques à chaque appel).

Ecrivons maintenant la fonction R associée.

```
rsum ← function(n, lambda, mu, sigma)
    return(.C("rsum",
        n = as.integer(n),
        lambda = as.double(lambda),
        mu = as.double(mu),
        sigma = as.double(sigma),
        S = numeric(n)
    )$S)

rsum(6, 5, 1, 1)
```

```
[1] 38.318945  2.306568  3.202413 17.568550 37.865063 69.789027
```

Il faut ensuite tester sa fonction, on peut par exemple vérifier la moyenne et la variance.

Comparons le gain en temps de calcul par rapport à deux approches sans code C. Nous définissons deux fonctions, `rsumR` qui est l'approche la plus simple qui comporte une unique boucle sur les simulations.

```
rsumR ← function(n, lambda, mu, sigma)
{
    S ← numeric(n)
    for(i in 1:n)
        S[i] ← sum(rlnorm(rpois(1, lambda), mu, sigma))
    return(S)
}
```

Puis `rsumR2` qui ne comporte aucune boucle.

```
rsumR2 ← function(n, lambda, mu, sigma)
{
    S ← numeric(n)
    N ← rpois(n, lambda)
    X ← matrix(0, n, max(N))
    X[col(X) <= N] ← rlnorm(sum(N), mu, sigma)
    return(rowSums(X))
}
```

Nous allons comparer le temps de calcul grâce au paquet **microbenchmark**. Il ajoute la fonction **microbenchmark**, celle-ci prend un code R, par défaut l'execute 100 fois, et renvoie le temps de calcul médian. Comparons les 3 fonctions.

```
microbenchmark(rsum(10^4, 5, 1, 1))
```

```
Unit: milliseconds
                expr   min    lq   mean median     uq   max neval
 rsum(10^4, 5, 1, 1) 5.499 5.645 5.8364 5.7814 6.0081 6.770   100
```

```
microbenchmark(rsumR(10^4, 5, 1, 1))
```

```
Unit: milliseconds
                 expr   min    lq   mean median    uq   max neval
 rsumR(10^4, 5, 1, 1) 36.23 37.54 40.175 40.829 41.40 60.28   100
```

```
microbenchmark(rsumR2(10^4, 5, 1, 1))
```

```
Unit: milliseconds
                  expr   min    lq   mean median    uq   max neval
 rsumR2(10^4, 5, 1, 1) 7.465 8.076 8.3527 8.1723 8.298 13.03   100
```

Si nous comparons les temps de calcul médians dans cet exemple, la fonction C prend environ `5.8ms` contre `40.8ms` par la fonction **R** simple et `8.2ms` la fonction **R** sans boucle. On remarque qu'un code **R** très bien pensé permet de se rapprocher du temps de calcul de la foncton C, sans l'atteindre mais en étant sous un facteur 2. Tandis que le code **R** simple est environ 7 fois plus lent que la fonction C (et 5 fois que la fonction R sans boucle).

En revanche, dans cet exemple, la fonction R sans boucle consomme beaucoup plus de mémoire que les autres pendant le calcul. Elle demande à construire une matrice $n \times max(N)$ et si lambda est élevé, celle-ci peut être gigantesque. La fonction C cumule l'avantage d'être la plus efficace avec la consommation de mémoire la plus faible durant le calcul.

6.2 Manipulation d'objet de R avec `.Call()`

Dans la section précédente, il n'était possible que de passer des vecteurs via pointeurs, et de récupérer une version modifiée de ces derniers. Il est également possible d'envoyer directement des objets de R et de renvoyer un objet de R nouvellement créé. Nous allons utiliser la structure interne des objets de R , les structure **SEXP**. Il faudra rajouter en entête **#include** <**Rinternals.h**> qui permet d'accéder aux structures de R . Reprenons l'exemple de la somme de deux éléments.

```
#include <R.h>
#include <Rinternals.h>
#include <Rmath.h>

SEXP somme(SEXP x, SEXP y)
{
    SEXP z;
    PROTECT(z = allocVector(REALSXP, 1));
    REAL(z)[0] = REAL(x)[0] + REAL(y)[0];
    UNPROTECT(1);
    return(z);
}
```

Dans le code ci-dessus

- **SEXP** est la structure des objets de R, la fonction en renverra un et en prendra deux en argument. Tous les objets de R sont des **SEXP**, du vecteur de taille 1 à la liste.
- **allocVector** permet de construire un vecteur, le mot-clé **REALSXP** précise qu'il s'agit d'un type réel (**double**), et le chiffre 1 est la taille du vecteur. De plus, l'appel dans se faire dans l'environnement **PROTECT**.
- La fonction **REAL** permet d'accéder au tableau de données du vecteur z. Comme z est une structure avec d'autres informations (comme la taille du vecteur), cela permet d'accéder aux élément du vecteur. Ensuite, il faut spécifier lequel, même de taille 1, c'est pourquoi on utilise **[0]**.
- Enfin, on appelle **UNPROTECT(1)** pour sortir de l'environnement de protection et on peut renvoyer le vecteur **z**.

Après avoir chargé la biblothèque de fonctions, l'appel dans R se fait comme avant, sauf qu'on utilise la fonction **.Call**.

```
somme <- function(x, y)
    return(.Call("somme", as.double(x), as.double(y)))

somme(1, pi)
```

```
[1] 4.141593
```

Adaptons maintenant la fonction `rsum` précédente à ce format.

```
SEXP rsum(SEXP n, SEXP lambda, SEXP mu, SEXP sigma)
{
    int N, _n = INTEGER(n)[0];
    double _lambda = REAL(lambda)[0], _mu = REAL(mu)[0], _sigma =
       REAL(sigma)[0];

    SEXP S;
    PROTECT(S = allocVector(REALSXP, _n));
    double * pS = REAL(S);

    GetRNGstate();
    for(int i = 0; i != _n ; ++i)
    {
        N = rpois(_lambda);
        for(int k = 0; k != N ; k++)
            pS[i] += rlnorm(_mu, _sigma);
    }
    PutRNGstate();

    UNPROTECT(1);
    return(S);
}
```

Pour tous les vecteurs de taille 1, nous récupérons la valeur directement dans une variable de la même manière que précédemment. Pour le vecteur `S`, nous récupérons le pointeur : c'est ce qui permet d'accéder aux valeurs de `S` via un tableau après, avec `pS[i]`. Le reste du code est identique.

La fonction d'appel dans R est simplement :

```
rsum ← function(n, lambda, mu, sigma)
    return(.Call("rsum",
        as.integer(n),
        as.double(lambda),
        as.double(mu),
        as.double(sigma)
    ))

rsum(6, 5, 1, 1)
```

```
[1] 18.326962  4.962594 30.957576 28.276227  6.486745 25.729138
```

Le temps de calcul est très semblable à la version précédente. L'intérêt réside dans le fait qu'il est possible d'avoir en argument des matrices, des listes, d'utiliser cette structure dans C, et de renvoyer également une structure complexe dans R en retour.

Références

[1] Nicolas Baradel. Assurance dommage. https://nicolasbaradel.fr/enseignement/ressources/cours_assurance_dommage.pdf.

[2] Nicolas Baradel. Introduction au langage r. https://nicolasbaradel.fr/enseignement/ressources/cours_r.pdf.

[3] Nicolas Baradel. Méthodes numériques en finance. https://nicolasbaradel.fr/enseignement/ressources/cours_methodes_numeriques_finance.pdf.

[4] Nicolas Baradel. Théorie du risque. https://nicolasbaradel.fr/enseignement/ressources/cours_theorie_du_risque.pdf.

[5] Nicolas Baradel. *Langage R : Introduction à la Statistique, à l'Actuariat et à la Finance*. Economica, 2015.