Table des matières

4 Exemples résolus

Objectifs du chapitre

- Mettre en pratique les connaissances acquises dans les chapitres précédents.
- Savoir tirer profit de l'arithmétique vectorielle de R pour effectuer des calculs complexes sans boucles.
- Utiliser l'initialisation de vecteurs et leur indiçage de manière à réduire le temps de calcul.

Ce chapitre propose de faire le point sur les concepts étudiés jusqu'à maintenant par le biais de quelques exemples résolus. On y met particulièrement en évidence les avantages de l'approche vectorielle du langage R.

La compréhension du contexte de ces exemples requiert quelques connaissances de base en mathématiques financières et en théorie des probabilités.

4.1 Calcul de valeurs présentes

De manière générale, la valeur présente d'une série de paiements $P_1, P_2, \ldots, P_n$ à la fin des années $1, 2, \ldots, n$ est

$$\sum_{j=1}^{n} \prod_{k=1}^{j} (1+i_k)^{-1} P_j, \tag{4.1}$$

où i_k est le taux d'intérêt effectif annuellement durant l'année k. Lorsque le taux d'intérêt est constant au cours des n années, cette formule se simplifie en

$$\sum_{j=1}^{n} (1+i)^{-j} P_j. \tag{4.2}$$

Un prêt est remboursé par une série de cinq paiements, le premier étant dû dans un an. On doit trouver le montant du prêt pour chacune des hypothèses ci-dessous.

a) Paiement annuel de 1 000, taux d'intérêt de 6 % effectif annuellement.

Avec un paiement annuel et un taux d'intérêt constants, on utilise la formule (4.2) avec $P_j = P = 1\,000$:

```
> 1000 * sum((1 + 0.06)^(-(1:5)))
[1] 4212.364
```

Remarquer comme l'expression R se lit exactement comme la formule mathématique.

b) Paiements annuels de 500, 800, 900, 750 et 1 000, taux d'intérêt de 6 % effectif annuellement.

Les paiements annuels sont différents, mais le taux d'intérêt est toujours le même. La formule (4.2) s'applique donc directement :

```
> sum(c(500, 800, 900, 750, 1000) * (1 + 0.06)^(-(1:5)))
[1] 3280.681
```

c) Paiements annuels de 500, 800, 900, 750 et 1 000, taux d'intérêt de 5 %, 6 %, 5,5 %, 6,5 % et 7 % effectifs annuellement.

Avec différents paiements annuels et des taux d'intérêt différents, il faut employer la formule (4.1). Le produit cumulatif des taux d'intérêt est obtenu avec la fonction `cumprod` :

```
> sum(c(500, 800, 900, 750, 1000) /
+     cumprod(1 + c(0.05, 0.06, 0.055, 0.065, 0.07)))
[1] 3308.521
```

4.2 Fonctions de masse de probabilité

On doit calculer toutes ou la majeure partie des probabilités de deux lois de probabilité, puis vérifier que la somme des probabilités est bien égale à 1.

Cet exemple est quelque peu artificiel dans la mesure où il existe dans R des fonctions internes pour calculer les principales caractéristiques des lois de probabilité les plus usuelles. Nous utiliserons d'ailleurs ces fonctions pour vérifier nos calculs.

a) Calculer toutes les masses de probabilité de la distribution binomiale pour des valeurs des paramètres n et p quelconques. La fonction de masse de probabilité de la binomiale est

$$f(x) = \binom{n}{x} p^x (1-p)^{n-x}, \quad x = 0, \ldots, n.$$

Soit $n = 10$ et $p = 0{,}8$. Les coefficients binomiaux sont calculés avec la fonction `choose` :

```
> n <- 10
> p <- 0.8
> x <- 0:n
> choose(n, x) * p^x * (1 - p)^rev(x)
 [1] 0.0000001024 0.0000040960 0.0000737280
 [4] 0.0007864320 0.0055050240 0.0264241152
 [7] 0.0880803840 0.2013265920 0.3019898880
[10] 0.2684354560 0.1073741824
```

On vérifie les réponses obtenues avec la fonction interne `dbinom` :

```
> dbinom(x, n, prob = 0.8)
 [1] 0.0000001024 0.0000040960 0.0000737280
 [4] 0.0007864320 0.0055050240 0.0264241152
 [7] 0.0880803840 0.2013265920 0.3019898880
[10] 0.2684354560 0.1073741824
```

On vérifie enfin que les probabilités somment à 1 :

```
> sum(choose(n, x) * p^x * (1 - p)^rev(x))
[1] 1
```

b) Calculer la majeure partie des masses de probabilité de la distribution de Poisson, dont la fonction de masse de probabilité est

$$f(x) = \frac{\lambda^x e^{-\lambda}}{x!}, \quad x = 0, 1, \ldots,$$

où $x! = x(x-1)\cdots 2 \cdot 1$.
La loi de Poisson ayant un support infini, on calcule les probabilités en $x = 0, 1, \ldots, 10$ seulement avec $\lambda = 5$. On calcule les factorielles avec la fonction `factorial`. On notera au passage que `factorial(x) == gamma(x + 1)`, où la fonction R `gamma` calcule les valeurs de la fonction mathématique du même nom

$$\Gamma(n) = \int_0^\infty x^{n-1} e^{-x}\, dx = (n-1)\Gamma(n-1),$$

avec $\Gamma(0) = 1$. Pour n entier, on a donc $\Gamma(n) = (n-1)!$.

```
> lambda <- 5
> x <- 0:10
> exp(-lambda) * (lambda^x / factorial(x))
```

```
 [1] 0.006737947 0.033689735 0.084224337
 [4] 0.140373896 0.175467370 0.175467370
 [7] 0.146222808 0.104444863 0.065278039
[10] 0.036265577 0.018132789
```

Vérification avec la fonction interne `dpois` :

```
> dpois(x, lambda)
 [1] 0.006737947 0.033689735 0.084224337
 [4] 0.140373896 0.175467370 0.175467370
 [7] 0.146222808 0.104444863 0.065278039
[10] 0.036265577 0.018132789
```

Pour vérifier que les probabilités somment à 1, il faudra d'abord tronquer le support infini de la Poisson à une «grande» valeur. Ici, 200 est suffisamment éloigné de la moyenne de la distribution, 5. Remarquer que le produit par $e^{-\lambda}$ est placé à l'extérieur de la somme pour ainsi faire un seul produit plutôt que 201.

```
> x <- 0:200
> exp(-lambda) * sum((lambda^x / factorial(x)))
[1] 1
```

4.3 Fonction de répartition de la loi gamma

La loi gamma est fréquemment utilisée pour la modélisation d'événements ne pouvant prendre que des valeurs positives et pour lesquels les petites valeurs sont plus fréquentes que les grandes. Par exemple, on utilise parfois la loi gamma en sciences actuarielles pour la modélisation des montants de sinistres. Nous utiliserons la paramétrisation où la fonction de densité de probabilité est

$$f(x) = \frac{\lambda^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-\lambda x}, \quad x > 0, \tag{4.3}$$

où $\Gamma(\cdot)$ est la fonction gamma définie dans l'exemple précédent.

Il n'existe pas de formule explicite de la fonction de répartition de la loi gamma. Néanmoins, la valeur de la fonction de répartition d'une loi gamma de paramètre α entier et $\lambda = 1$ peut être obtenue à partir de la formule

$$F(x;\alpha,1) = 1 - e^{-x} \sum_{j=0}^{\alpha-1} \frac{x^j}{j!}. \tag{4.4}$$

a) Évaluer $F(4;5,1)$.

Cet exercice est simple puisqu'il s'agit de calculer une seule valeur de la fonction de répartition avec un paramètre α fixe. Par une application directe de (4.4), on a :

```
> alpha <- 5
> x <- 4
> 1 - exp(-x) * sum(x^(0:(alpha - 1))/gamma(1:alpha))
[1] 0.3711631
```

Vérification avec la fonction interne `pgamma` :

```
> pgamma(x, alpha)
[1] 0.3711631
```

On peut aussi éviter de générer essentiellement la même suite de nombres à deux reprises en ayant recours à une variable intermédiaire. Au risque de rendre le code un peu moins lisible (mais plus compact !), l'affectation et le calcul final peuvent même se faire dans une seule expression.

```
> 1 - exp(-x) * sum(x^(-1 + (j <- 1:alpha))/gamma(j))
[1] 0.3711631
```

b) Évaluer $F(x;5,1)$ pour $x = 2,3,\ldots,10$ en une seule expression.

Cet exercice est est beaucoup plus compliqué qu'il n'y paraît au premier abord. Ici, la valeur de α demeure fixe, mais on doit calculer, en une seule expression, la valeur de la fonction de répartition en plusieurs points. Or, cela exige de faire d'un coup le calcul x^j pour plusieurs valeur de x et plusieurs valeurs de j. C'est un travail pour la fonction `outer` :

```
> x <- 2:10
> 1 - exp(-x) * colSums(t(outer(x, 0:(alpha-1), "^")) /
+                       gamma(1:alpha))
[1] 0.05265302 0.18473676 0.37116306 0.55950671
[5] 0.71494350 0.82700839 0.90036760 0.94503636
[9] 0.97074731
```

Vérification avec la fonction interne `pgamma` :

```
> pgamma(x, alpha)
[1] 0.05265302 0.18473676 0.37116306 0.55950671
[5] 0.71494350 0.82700839 0.90036760 0.94503636
[9] 0.97074731
```

Il est laissé en exercice de déterminer pourquoi la transposée est nécessaire dans l'expression ci-dessus. Exécuter l'expression étape par étape, de l'intérieur vers l'extérieur, pour mieux comprendre comment on arrive à faire le calcul en (4.4).

4.4 Algorithme du point fixe

Trouver la racine d'une fonction g — c'est-à-dire le point x où $g(x) = 0$ — est un problème classique en mathématiques. Très souvent, il est possible de reformuler le problème de façon à plutôt chercher le point x où $f(x) = x$. La solution d'un tel problème est appelée *point fixe*.

L'algorithme du calcul numérique du point fixe d'une fonction $f(x)$ est très simple :

1. choisir une valeur de départ x_0 ;
2. calculer $x_n = f(x_{n-1})$ pour $n = 1, 2, \ldots$;
3. répéter l'étape 2 jusqu'à ce que $|x_n - x_{n-1}| < \epsilon$ ou $|x_n - x_{n-1}|/|x_{n-1}| < \epsilon$.

On doit trouver, à l'aide de la méthode du point fixe, la valeur de i telle que

$$a_{\overline{10}|} = \frac{1-(1+i)^{-10}}{i} = 8{,}21,$$

c'est à dire le taux de rendement d'une série de 10 versements de 1 pour laquelle on a payé un montant de 8,21.

Puisque, d'une part, nous ignorons combien de fois la procédure itérative devra être répétée et que, d'autre part, il faut exécuter la procédure au moins une fois, le choix logique pour la structure de contrôle à utiliser dans cette procédure itérative est `repeat`. De plus, il faut comparer deux valeurs successives du taux d'intérêt, nous devrons donc avoir recours à deux variables. On a :

```
> i <- 0.05
> repeat
+ {
+     it <- i
+     i <- (1 - (1 + it)^(-10))/8.21
+     if (abs(i - it)/it < 1E-10)
+         break
+ }
> i
[1] 0.03756777
```

Vérification :

```
> (1 - (1 + i)^(-10))/i
[1] 8.21
```

Nous verrons au chapitre 5 comment créer une fonction à partir de ce code.

4.5 Suite de Fibonacci

La suite de Fibonacci est une suite de nombres entiers très connue. Les deux premiers termes de la suite sont 0 et 1 et tous les autres sont la somme des deux termes précédents. Mathématiquement, les valeurs de la suite de Fibonacci sont données par la fonction

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 1 \\ f(n) &= f(n-1) + f(n-2), \quad n \geq 2. \end{aligned}$$

Le quotient de deux termes successifs converge vers $(1+\sqrt{5})/2$, le nombre d'or.

On veut calculer les $n > 2$ premiers termes de la suite de Fibonacci. Ce problème étant intrinsèquement récursif, nous devons utiliser une boucle.

Voici une première solution pour $n = 10$:

```
> n <- 10
> x <- c(0, 1)
> for (i in 3:n) x[i] <- x[i - 1] + x[i - 2]
> x
 [1]  0  1  1  2  3  5  8 13 21 34
```

La procédure ci-dessus a un gros défaut : la taille de l'objet x est constamment augmentée pour stocker une nouvelle valeur de la suite. Tentons une analogie alimentaire pour cette manière de procéder. Pour ranger des biscuits frais sortis du four, 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 biscuit soit épuisé.

C'est exactement ce qui se passe dans la mémoire de l'ordinateur avec la procédure ci-dessus, l'odeur des bons biscuits chauds en moins. En effet, le système

doit constamment allouer de la nouvelle mémoire et déplacer les termes déjà sauvegardés au fur et à mesure que le vecteur `x` grandit. On aura compris qu'une telle façon de faire est à éviter absolument lorsque c'est possible — et ça l'est la plupart du temps.

Quand on sait quelle sera la longueur d'un objet, comme c'est le cas dans cet exemple, il vaut mieux créer un contenant vide de la bonne longueur et le remplir par la suite. Cela nous donne une autre façon de calculer la suite de Fibonacci :

```
> n <- 10
> x <- numeric(n)      # contenant créé
> x[2] <- 1            # x[1] vaut déjà 0
> for (i in 3:n) x[i] <- x[i - 1] + x[i - 2]
> x
 [1]  0  1  1  2  3  5  8 13 21 34
```

Dans les exemples du chapitre 5, nous composeront des fonctions avec ces deux exemples et nous comparerons les temps de calcul pour n grand.

4.6 Exercices

Dans chacun des exercices ci-dessous, écrire une expression R pour faire le calcul demandé. Parce qu'elles ne sont pas nécessaires, il est interdit d'utiliser des boucles.

4.1 Calculer la valeur présente d'une série de paiements fournie dans un vecteur `P` en utilisant les taux d'intérêt annuels d'un vecteur `i`.

4.2 Étant donné un vecteur d'observations $\boldsymbol{x} = (x_1, \ldots, x_n)$ et un vecteur de poids correspondants $\boldsymbol{w} = (w_1, \ldots, w_n)$, calculer la moyenne pondérée des observations,

$$\sum_{i=1}^{n} \frac{w_i}{w_\Sigma} x_i,$$

où $w_\Sigma = \sum_{i=1}^{n} w_i$. Tester l'expression avec les vecteurs de données

$$\boldsymbol{x} = (7, 13, 3, 8, 12, 12, 20, 11)$$

et

$$\boldsymbol{w} = (0{,}15,\ 0{,}04,\ 0{,}05,\ 0{,}06,\ 0{,}17,\ 0{,}16,\ 0{,}11,\ 0{,}09).$$

4.3 Soit un vecteur d'observations $\boldsymbol{x} = (x_1, \ldots, x_n)$. Calculer la moyenne harmonique de ce vecteur, définie comme

$$\frac{n}{\frac{1}{x_1} + \cdots + \frac{1}{x_n}}.$$

Tester l'expression avec les valeurs de l'exercice 4.2.

4.4 Calculer la fonction de répartition en $x = 5$ d'une loi de Poisson avec paramètre $\lambda = 2$, qui est donnée par

$$\sum_{k=0}^{5} \frac{2^k e^{-2}}{k!},$$

où $k! = 1 \cdot 2 \cdots k$.

4.5 a) Calculer l'espérance d'une variable aléatoire X dont le support est $x = 1, 10, 100, ..., 1\,000\,000$ et les probabilités correspondantes sont $\frac{1}{28}, \frac{2}{28}, ..., \frac{7}{28}$, dans l'ordre.

b) Calculer la variance de la variable aléatoire X définie en a).

4.6 Calculer le taux d'intérêt nominal composé quatre fois par année, $i^{(4)}$, équivalent à un taux de $i = 6$ % effectif annuellement.

4.7 La valeur présente d'une série de n paiements de fin d'année à un taux d'intérêt i effectif annuellement est

$$a_{\overline{n}|} = v + v^2 + \cdots + v^n = \frac{1 - v^n}{i},$$

où $v = (1 + i)^{-1}$. Calculer en une seule expression, toujours sans boucle, un tableau des valeurs présentes de séries de $n = 1, 2, \ldots, 10$ paiements à chacun des taux d'intérêt effectifs annuellement $i = 0{,}05, 0{,}06, \ldots, 0{,}10$.

4.8 Calculer la valeur présente d'une annuité croissante de 1 $ payable annuellement en début d'année pendant 10 ans si le taux d'actualisation est de 6 %. Cette valeur présente est donnée par

$$I\ddot{a}_{\overline{10}|} = \sum_{k=1}^{10} k v^{k-1},$$

toujours avec $v = (1 + i)^{-1}$.

4.9 Calculer la valeur présente de la séquence de paiements 1, 2, 2, 3, 3, 3, 4, 4, 4, 4 si les paiements sont effectués en fin d'année et que le taux d'actualisation est de 7 %.

4.10 Calculer la valeur présente de la séquence de paiements définie à l'exercice 4.9 en supposant que le taux d'intérêt d'actualisation alterne successivement entre 5 % et 8 % chaque année, c'est-à-dire que le taux d'intérêt est de 5 %, 8 %, 5 %, 8 %, etc.

5 Fonctions définies par l'usager

Objectifs du chapitre

- Savoir définir une fonction R, ses divers arguments et, le cas échéant, les valeurs par défaut de ceux-ci.
- Être en mesure de déboguer une fonction R.
- Adopter un style de codage correspondant à la pratique en R.

La possibilité pour l'usager de définir facilement et rapidement de nouvelles fonctions — et donc des extensions au langage — est une des grandes forces de R. Les fonctions personnelles définies dans l'espace de travail ou dans un package sont traitées par le système exactement comme les fonction internes.

Ce court chapitre passe en revue la syntaxe et les règles pour créer des fonctions dans R. On discute également brièvement de débogage et de style de codage.

5.1 Définition d'une fonction

On définit une nouvelle fonction avec la syntaxe suivante :

```
fun <- function(arguments) expression
```

où

- `fun` est le nom de la fonction (les règles pour les noms de fonctions étant les mêmes que celles présentées à la section 2.2 pour tout autre objet) ;
- *`arguments`* est la liste des arguments, séparés par des virgules ;
- *`expression`* constitue le corps de la fonction, soit une expression ou un groupe d'expressions réunies par des accolades.

5.2 Retourner des résultats

La plupart des fonctions sont écrites dans le but de retourner un résultat. Or, les règles d'interprétation d'un groupe d'expressions présentées à la section 2.1 s'appliquent ici au corps de la fonction.

- Une fonction retourne tout simplement le résultat de la *dernière expression* du corps de la fonction.
- On évitera donc que la dernière expression soit une affectation, car la fonction ne retournera alors rien et on ne pourra utiliser une construction de la forme `x <- f()` pour affecter le résultat de la fonction à une variable.
- Si on doit retourner un résultat sans être à la dernière ligne de la fonction (à l'intérieur d'un bloc conditionnel, par exemple), on utilise la fonction `return`. *L'utilisation de* `return` *à la toute fin d'une fonction est tout à fait inutile et considérée comme du mauvais style en R.*
- Lorsqu'une fonction doit retourner plusieurs résultats, il est en général préférable d'avoir recours à une liste nommée.

5.3 Variables locales et globales

Comme la majorité des langages de programmation, R comporte des concepts de variable locale et de variable globale.

- Toute variable définie dans une fonction est locale à cette fonction, c'est-à-dire qu'elle :
  - n'apparaît pas dans l'espace de travail ;
  - n'écrase pas une variable du même nom dans l'espace de travail.
- Il est possible de définir une variable dans l'espace de travail depuis une fonction avec l'opérateur d'affectation `<<-`. Il est très rare — et généralement non recommandé — de devoir recourir à de telles variables globales.
- On peut définir une fonction à l'intérieur d'une autre fonction. Cette fonction sera locale à la fonction dans laquelle elle est définie.

Le lecteur intéressé à en savoir plus pourra consulter les sections de la documentation de R portant sur la portée lexicale (*lexical scoping*). C'est un sujet important et intéressant, mais malheureusement trop avancé pour ce document d'introduction à la programmation en R.

```
fp <- function(k, n, start = 0.05, TOL = 1E-10)
{
    ## Fonction pour trouver par la méthode du point
    ## fixe le taux d'intérêt pour lequel une série de
    ## 'n' paiements vaut 'k'.
    ##
    ## ARGUMENTS
    ##
    ##     k: la valeur présente des paiements
    ##     n: le nombre de paiements
    ## start: point de départ des itérations
    ##   TOL: niveau de précision souhaité
    ##
    ## RETOURNE
    ##
    ## Le taux d'intérêt

    i <- start
    repeat
    {
        it <- i
        i <- (1 - (1 + it)^(-n))/k
        if (abs(i - it)/it < TOL)
            break
    }
    i
}
```

FIG. 5.1: Exemple de fonction de point fixe

5.4 Exemple de fonction

Le code développé pour l'exemple de point fixe de la section 4.4 peut être intégré dans une fonction ; voir la figure 5.1.

- Le nom de la fonction est `fp`.
- La fonction compte quatre arguments : `k`, `n`, `start` et `TOL`.

- Les deux derniers arguments ont des valeurs par défaut de 0,05 et 10^{-10}, respectivement.
- La fonction retourne la valeur de la variable `i`.

5.5 Fonctions anonymes

Il est parfois utile de définir une fonction sans lui attribuer un nom — d'où la notion de *fonction anonyme*. Il s'agira en général de fonctions courtes utilisées dans une autre fonction. Par exemple, pour calculer la valeur de xy^2 pour toutes les combinaisons de x et y stockées dans des vecteurs du même nom, on pourrait utiliser la fonction `outer` ainsi :

```
> x <- 1:3; y <- 4:6
> f <- function(x, y) x * y^2
> outer(x, y, f)
     [,1] [,2] [,3]
[1,]   16   25   36
[2,]   32   50   72
[3,]   48   75  108
```

Cependant, si la fonction `f` ne sert à rien ultérieurement, on peut se contenter de passer l'objet fonction à `outer` sans jamais lui attribuer un nom :

```
> outer(x, y, function(x, y) x * y^2)
     [,1] [,2] [,3]
[1,]   16   25   36
[2,]   32   50   72
[3,]   48   75  108
```

On a alors utilisé dans `outer` une fonction anonyme.

5.6 Débogage de fonctions

Il est assez rare d'arriver à écrire un bout de code sans bogue du premier coup. Par conséquent, qui dit programmation dit séances de débogage.

Les techniques de débogages les plus simples et naïves sont parfois les plus efficaces et certainement les plus faciles à apprendre. Loin d'un traité sur le débogage de code R, nous offrons seulement ici quelques trucs que nous utilisons régulièrement.

- Les erreurs de syntaxe sont les plus fréquentes (en particulier l'oubli de virgules). Lors de la définition d'une fonction, une vérification de la syntaxe est effectuée par l'interprète R. Attention, cependant : une erreur peut prendre sa source plusieurs lignes avant celle que l'interprète pointe comme faisant problème.
- Les messages d'erreur de l'interprète ne sont pas toujours d'un grand secours. . .tant que l'on n'a pas appris à les reconnaître. Un exemple de message d'erreur fréquemment rencontré :

  ```
  valeur manquante là où TRUE / FALSE est requis
  ```

  Cela provient généralement d'une commande `if` dont l'argument vaut `NA` plutôt que `TRUE` ou `FALSE`. La raison : des valeurs manquantes se sont faufilées dans les calculs à notre insu.
- Lorsqu'une fonction ne retourne pas le résultat attendu, placer des commandes `print` à l'intérieur de la fonction, de façon à pouvoir suivre les valeurs prises par les différentes variables.

  Par exemple, la modification suivante à la boucle de la fonction `fp` permet d'afficher les valeurs successives de la variable `i` et de détecter une procédure itérative divergente :

  ```
  repeat
  {
      it <- i
      i <- (1 - (1 + it)^(-n))/k
      print(i)
      if (abs((i - it)/it < TOL))
          break
  }
  ```
- Quand ce qui précède ne fonctionne pas, ne reste souvent qu'à exécuter manuellement la fonction. Pour ce faire, définir dans l'espace de travail tous les arguments de la fonction, puis exécuter le corps de la fonction ligne par ligne. La vérification du résultat de chaque ligne permet généralement de retrouver la ou les expressions qui causent problème.

5.7 Styles de codage

Si tous conviennent que l'adoption d'un style propre et uniforme favorise le développement et la lecture de code, il existe plusieurs chapelles dans le monde des programmeurs quant à la «bonne façon» de présenter et, surtout, d'indenter le code informatique.

Par exemple, Emacs reconnaît et supporte les styles de codage suivants, entre autres :

C++/Stroustrup
```
for (i in 1:10)
{
    expression
}
```

K&R (1TBS)
```
for (i in 1:10){
     expression
}
```

Whitesmith
```
for (i in 1:10)
     {
     expression
     }
```

GNU
```
for (i in 1:10)
  {
    expression
  }
```

- Pour des raisons générales de lisibilité et de popularité, le style C++, avec les accolades sur leurs propres lignes et une indentation de quatre (4) espaces est considéré comme standard pour la programmation en R.
- La version de GNU Emacs distribuée par l'auteur est déjà configurée pour utiliser ce style de codage.
- Consulter la documentation de votre éditeur de texte pour savoir si vous pouvez configurer le niveau d'indentation. La plupart des bons éditeurs pour programmeurs le permettent.
- Surtout, éviter de ne pas du tout indenter le code.

5.8 Exemples

```
### POINT FIXE

## Comme premier exemple de fonction, on réalise une mise en
## oeuvre de l'algorithme du point fixe pour trouver le taux
## d'intérêt tel que a_angle{n} = k pour 'n' et 'k' donnés.
## Cette mise en oeuvre est peu générale puisqu'il faudrait
## modifier la fonction chaque fois que l'on change la
```

```
## fonction f(x) dont on cherche le point fixe.
fp1 <- function(k, n, start = 0.05, TOL = 1E-10)
{
    i <- start
    repeat
    {
        it <- i
        i <- (1 - (1 + it)^(-n))/k
        if (abs(i - it)/it < TOL)
            break
    }
    i
}

fp1(7.2, 10)               # valeur de départ par défaut
fp1(7.2, 10, 0.06)         # valeur de départ spécifiée
i                          # les variables n'existent pas...
start                      # ... dans l'espace de travail

## Généralisation de la fonction 'fp1': la fonction f(x) dont
## on cherche le point fixe (c'est-à-dire la valeur de 'x'
## tel que f(x) = x) est passée en argument. On peut faire
## ça? Bien sûr, puisqu'une fonction est un objet comme un
## autre en R. On ajoute également à la fonction un argument
## 'echo' qui, lorsque TRUE, fera en sorte d'afficher à
## l'écran les valeurs successives de 'x'.
##
## Ci-dessous, il est implicite que le premier argument, FUN,
## est une fonction.
fp2 <- function(FUN, start, echo = FALSE, TOL = 1E-10)
{
    x <- start
    repeat
    {
        xt <- x

        if (echo)       # inutile de faire 'if (echo == TRUE)'
            print(xt)

        x <- FUN(xt)    # appel de la fonction

        if (abs(x - xt)/xt < TOL)
            break
    }
    x
```

```
}

f <- function(i) (1 - (1+i)^(-10))/7.2 # définition de f(x)
fp2(f, 0.05)               # solution
fp2(f, 0.05, echo = TRUE)  # avec résultats intermédiaires
fp2(function(x) 3^(-x), start = 0.5) # avec fonction anonyme

## Amélioration mineure à la fonction 'fp2': puisque la
## valeur de 'echo' ne change pas pendant l'exécution de la
## fonction, on peut éviter de refaire le test à chaque
## itération de la boucle. Une solution élégante consiste à
## utiliser un outil avancé du langage R: les expressions.
##
## L'objet créé par la fonction 'expression' est une
## expression non encore évaluée (comme si on n'avait pas
## appuyé sur Entrée à la fin de la ligne). On peut ensuite
## évaluer l'expression (appuyer sur Entrée) avec 'exec'.
fp3 <- function(FUN, start, echo = FALSE, TOL = 1E-10)
{
    x <- start

    ## Choisir l'expression à exécuter plus loin
    if (echo)
        expr <- expression(print(xt <- x))
    else
        expr <- expression(xt <- x)

    repeat
    {
        eval(expr)         # évaluer l'expression

        x <- FUN(xt)       # appel de la fonction

        if (abs(x - xt)/xt < TOL)
            break
    }
    x
}

fp3(f, 0.05, echo = TRUE)  # avec résultats intermédiaires
fp3(function(x) 3^(-x), start = 0.5) # avec une fonction anonyme

### SUITE DE FIBONACCI

## On a présenté au chapitre 4 deux manières différentes de
```

```
## pour calculer les 'n' premières valeurs de la suite de
## Fibonacci. On crée d'abord des fonctions à partir de ce
## code. Avantage d'avoir des fonctions: elles sont valides
## pour tout 'n' > 2.
##
## D'abord la version inefficace.
fib1 <- function(n)
{
    res <- c(0, 1)
    for (i in 3:n)
        res[i] <- res[i - 1] + res[i - 2]
    res
}
fib1(10)
fib1(20)

## Puis la version supposéement plus efficace.
fib2 <- function(n)
{
    res <- numeric(n)      # contenant créé
    res[2] <- 1            # res[1] vaut déjà 0
    for (i in 3:n)
        res[i] <- res[i - 1] + res[i - 2]
    res
}
fib2(5)
fib2(20)

## A-t-on vraiment gagné en efficacité? Comparons le temps
## requis pour générer une longue suite de Fibonacci avec les
## deux fonctions.
system.time(fib1(10000))   # version inefficace
system.time(fib2(10000))   # version efficace

## Variation sur un même thème: une fonction pour calculer non
## pas les 'n' premières valeurs de la suite de Fibonacci,
## mais uniquement la 'n'ième valeur.
##
## Mais il y a un mais: la fonction 'fib3' est truffée
## d'erreurs (de syntaxe, d'algorithmique, de conception). À
## vous de trouver les bogues. (Afin de préserver cet
## exemple, copier le code erroné plus bas ou dans un autre
## fichier avant d'y faire les corrections.)
fib3 <- function(nb)
{
```

```
    x <- 0
    x1 _ 0
    x2 <- 1
    while (n > 0))
x <- x1 + x2
x2 <- x1
x1 <- x
n <- n - 1
}
fib3(1)                    # devrait donner 0
fib3(2)                    # devrait donner 1
fib3(5)                    # devrait donner 3
fib3(10)                   # devrait donner 34
fib3(20)                   # devrait donner 4181
```

5.9 Exercices

5.1 La fonctions `var` calcule l'estimateur sans biais de la variance d'une population à partir de l'échantillon donné en argument. Écrire une fonction `variance` qui calculera l'estimateur biaisé ou sans biais selon que l'argument `biased` sera `TRUE` ou `FALSE`, respectivement. Le comportement par défaut de `variance` devrait être le même que celui de `var`. L'estimateur sans biais de la variance à partir d'un échantillon $X_1, \ldots, X_n$ est

$$S^2_{n-1} = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X})^2,$$

alors que l'estimateur biaisé est

$$S^2_n = \frac{1}{n}\sum_{i=1}^{n}(X_i - \bar{X})^2,$$

où $\bar{X} = n^{-1}(X_1 + \cdots + X_n)$.

5.2 Écrire une fonction `matrix2` qui, contrairement à la fonction `matrix`, remplira par défaut la matrice par ligne. La fonction *ne doit pas* utiliser `matrix`. Les arguments de la fonction `matrix2` seront les mêmes que ceux de `matrix`, sauf que l'argument `byrow` sera remplacé par `bycol`.

5.3 Écrire une fonction `phi` servant à calculer la fonction de densité de probabilité d'une loi normale centrée réduite, soit

$$\phi(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}, \quad -\infty < x < \infty.$$

La fonction devrait prendre en argument un vecteur de valeurs de x. Comparer les résultats avec ceux de la fonction `dnorm`.

5.4 Écrire une fonction `Phi` servant à calculer la fonction de répartition d'une loi normale centrée réduite, soit

$$\Phi(x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-y^2/2}\, dy, \quad -\infty < x < \infty.$$

Supposer, pour le moment, que $x \geq 0$. L'évaluation numérique de l'intégrale ci-dessus peut se faire avec l'identité

$$\Phi(x) = \frac{1}{2} + \phi(x) \sum_{n=0}^{\infty} \frac{x^{2n+1}}{1 \cdot 3 \cdot 5 \cdots (2n+1)}, \quad x \geq 0.$$

Utiliser la fonction `phi` de l'exercice 5.3 et tronquer la somme infinie à une grande valeur, 50 par exemple. La fonction ne doit pas utiliser de boucles, mais peut ne prendre qu'une seule valeur de x à la fois. Comparer les résultats avec ceux de la fonction `pnorm`.

5.5 Modifier la fonction `Phi` de l'exercice 5.4 afin qu'elle admette des valeurs de x négatives. Lorsque $x < 0$, $\Phi(x) = 1 - \Phi(-x)$. La solution simple consiste à utiliser une structure de contrôle `if ... else`, mais les curieux chercheront à s'en passer.

5.6 Généraliser maintenant la fonction de l'exercice 5.5 pour qu'elle prenne en argument un vecteur de valeurs de x. Ne pas utiliser de boucle. Comparer les résultats avec ceux de la fonction `pnorm`.

5.7 Sans utiliser l'opérateur `%*%`, écrire une fonction `prod.mat` qui effectuera le produit matriciel de deux matrices seulement si les dimensions de celles-ci le permettent. Cette fonction aura deux arguments (`mat1` et `mat2`) et devra tout d'abord vérifier si le produit matriciel est possible. Si celui-ci est impossible, la fonction retourne un message d'erreur.

a) Utiliser une structure de contrôle `if ... else` et deux boucles.

b) Utiliser une structure de contrôle `if ... else` et une seule boucle.

Dans chaque cas, comparer le résultat avec l'opérateur `%*%`.

5.8 Vous devez calculer la note finale d'un groupe d'étudiants à partir de deux informations : 1) une matrice contenant la note sur 100 de chacun des étudiants à chacune des évaluations, et 2) un vecteur contenant la pondération de chacune des évaluations. Un de vos collègues a composé la fonction `notes.finales` ci-dessous afin de faire le calcul de la note finale pour chacun de ses étudiants.

Votre collègue vous mentionne toutefois que sa fonction est plutôt lente et inefficace pour de grands groupes d'étudiants. Modifiez la fonction afin d'en réduire le nombre d'opérations et faire en sorte qu'elle n'utilise aucune boucle.

```
notes.finales <- function(notes, p)
{
    netud <- nrow(notes)
    neval <- ncol(notes)
    final <- (1:netud) * 0
    for(i in 1:netud)
    {
        for(j in 1:neval)
        {
            final[i] <- final[i] + notes[i, j] * p[j]
        }
    }
    final
}
```

5.9 Trouver les erreurs qui empêchent la définition de la fonction ci-dessous.

```
AnnuiteFinPeriode <- function(n, i)
{{
    v <- 1/1 + i)
    ValPresChaquePmt <- v^(1:n)
    sum(ValPresChaquepmt)
}
```

5.10 La fonction ci-dessous calcule la valeur des paramètres d'une loi normale, gamma ou Pareto à partir de la moyenne et de la variance, qui sont connues par l'utilisateur.

```
param <- function(moyenne, variance, loi)
{
    loi <- tolower(loi)
    if (loi == "normale")
        param1 <- moyenne
        param2 <- sqrt(variance)
        return(list(mean = param1, sd = param2))
    if (loi == "gamma")
        param2 <- moyenne/variance
        param1 <- moyenne * param2
        return(list(shape = param1, scale = param2))
```

```
    if (loi == "pareto")
        cte <- variance/moyenne^2
        param1 <- 2 * cte/(cte-1)
        param2 <- moyenne * (param1 - 1)
        return(list(alpha = param1, lambda = param2))
    stop("La loi doit etre une de \"normale\",
\"gamma\" ou \"pareto\"")
}
```

L'utilisation de la fonction pour diverses lois donne les résultats suivants :

```
> param(2, 4, "normale")
$mean
[1] 2

$sd
[1] 2

> param(50, 7500, "gamma")

Erreur dans param(50, 7500, "gamma") : Objet "param1"
non trouvé

> param(50, 7500, "pareto")

Erreur dans param(50, 7500, "pareto") : Objet "param1"
non trouvé
```

a) Expliquer pour quelle raison la fonction se comporte ainsi.

b) Appliquer les correctifs nécessaires à la fonction pour que celle-ci puisse calculer les bonnes valeurs. (Les erreurs ne se trouvent pas dans les mathématiques de la fonction.) *Astuce* : tirer profit du moteur d'indentation de votre éditeur de texte pour programmeur.

Bibliographie

Abelson, H., G. J. Sussman et J. Sussman. 1996, *Structure and Interpretation of Computer Programs*, 2e éd., MIT Press, ISBN 0-26201153-0.

Becker, R. A. 1994, «A brief history of S», cahier de recherche, AT&T Bell Laboratories. URL http://cm.bell-labs.com/cm/ms/departments/sia/doc/94.11.ps.

Becker, R. A. et J. M. Chambers. 1984, *S: An Interactive Environment for Data Analysis and Graphics*, Wadsworth, ISBN 0-53403313-X.

Becker, R. A., J. M. Chambers et A. R. Wilks. 1988, *The New S Language: A Programming Environment for Data Analysis and Graphics*, Wadsworth & Brooks/Cole, ISBN 0-53409192-X.

Braun, W. J. et D. J. Murdoch. 2007, *A First Course in Statistical Programming with R*, Cambridge University Press, ISBN 978-0-52169424-7.

Cameron, D., J. Elliott, M. Loy, E. S. Raymond et B. Rosenblatt. 2004, *Leaning GNU Emacs*, 3e éd., O'Reilly, Sebastopol, CA, ISBN 0-59600648-9.

Chambers, J. M. 1998, *Programming with Data: A Guide to the S Language*, Springer, ISBN 0-38798503-4.

Chambers, J. M. 2000, «Stages in the evolution of S», URL http://cm.bell-labs.com/cm/ms/departments/sia/S/history.html.

Chambers, J. M. 2008, *Software for Data Analysis: Programming with R*, Springer, ISBN 978-0-38775935-7.

Chambers, J. M. et T. J. Hastie. 1992, *Statistical Models in S*, Wadsworth & Brooks/-Cole, ISBN 0-53416765-9.

Hornik, K. 2011, «The R FAQ», URL http://cran.r-project.org/doc/FAQ/R-FAQ.html, ISBN 3-90005108-9.

Iacus, S. M., S. Urbanek et R. J. Goedman. 2011, «R for Mac OS X FAQ», URL http://cran.r-project.org/bin/macosx/RMacOSX-FAQ.html.

IEEE. 2003, *754-1985 IEEE Standard for Binary Floating-Point Arithmetic*, IEEE, Piscataway, NJ.

Ihaka, R. et R. Gentleman. 1996, «R: A language for data analysis and graphics», *Journal of Computational and Graphical Statistics*, vol. 5, nº 3, p. 299–314.

Ligges, U. 2003, «R-winedt», dans *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)*, édité par K. Hornik, F. Leisch et A. Zeileis, TU Wien, Vienna, Austria, ISSN 1609-395X. URL http://www.ci.tuwien.ac.at/Conferences/DSC-2003/Proceedings/.

Redd, A. 2010, «Introducing NppToR: R interaction for Notepad++», *R Journal*, vol. 2, nº 1, p. 62–63. URL http://journal.r-project.org/archive/2010-1/RJournal_2010-1.pdf.

Ripley, B. D. et D. J. Murdoch. 2011, «R for Windows FAQ», URL http://cran.r-project.org/bin/windows/base/rw-FAQ.html.

Venables, W. N. et B. D. Ripley. 2000, *S Programming*, Springer, New York, ISBN 0-38798966-8.

Venables, W. N. et B. D. Ripley. 2002, *Modern Applied Statistics with S*, 4e éd., Springer, New York, ISBN 0-38795457-0.

Venables, W. N., D. M. Smith et R Development Core Team. 2011, *An Introduction to R*, R Foundation for Statistical Computing. URL http://cran.r-project.org/doc/manuals/R-intro.html.

Index

Les numéros de page en caractères gras indiquent les pages où les concepts sont introduits, définis ou expliqués.

ISBN
978-2-98