Initiation au logiciel R

March 3, 2019

Table des matières

Les blocs d'instructions

Un bloc d'instruction est un ensemble d'instructions délimitées par des accolades :

- une accolade ouvrante pour indiquer le début du bloc.
- une accolade fermante pour indiquer la fin du bloc.

Les blocs d'instructions

Un bloc d'instruction est un ensemble d'instructions délimitées par des accolades :

- une accolade ouvrante pour indiquer le début du bloc.
- une accolade fermante pour indiquer la fin du bloc.

L'indentation du contenu d'un bloc est optionnel, mais indispensable pour une meilleur lisibilité du programme. Il faut donc écrire :

```
{
  a <- 10
  b <- 20
  {
    c <- a*b
  }
}
```

et non pas (illisible)

```
{
a <- 10
b <- 20
{
c <- a*b
}
}
```

Les blocs d'instructions

Un bloc d'instruction est un ensemble d'instructions délimitées par des accolades :

- une accolade ouvrante pour indiquer le début du bloc.
- une accolade fermante pour indiquer la fin du bloc.

L'indentation du contenu d'un bloc est optionnel, mais indispensable pour une meilleur lisibilité du programme. Il faut donc écrire :

```
{
  a <- 10
  b <- 20
  {
    c <- a*b
  }
}
```

et non pas (illisible)

```
{
a <- 10
b <- 20
{
c <- a*b
}
}
```

Structure conditionnelle : if/else

La syntaxe formelle d'un if /else est la suivante :

```
if( condition ) {
  # bloc d' instructions  1

}else{
  # bloc d' instructions  2
}
```

Structure conditionnelle : if/else

La syntaxe formelle d'un if /else est la suivante :

```
if( condition ) {
  # bloc d' instructions 1

}else{
  # bloc d' instructions 2
}
```

La clause else est facultative. L'écriture ci-dessous est tout à fait valide :

```
if ( x<0 ) {
  x <- -x
}
print(x)
```

Structure conditionnelle : if/else

La syntaxe formelle d'un if /else est la suivante :

```
if( condition ) {
  # bloc d' instructions  1

}else{
  # bloc d' instructions  2
}
```

La clause else est facultative. L'écriture ci-dessous est tout à fait valide :

```
if ( x<0 ) {
  x <- -x
}
print(x)
```

Structure conditionnelle : if/else

La syntaxe formelle d'un if /else est la suivante :

```
if( condition ) {
  # bloc d' instructions 1

}else{
  # bloc d' instructions 2
}
```

La clause else est facultative. L'écriture ci-dessous est tout à fait valide :

```
if ( x<0 ) {
  x <- -x
}
print(x)
```

Les if /else peuvent être imbriquées comme suit :

```
if (cond1) {
  # bloc d' instructions 1
} else if (cond2) {
  # bloc d' instructions 2
} else if(cond3) {
  # bloc d' instructions 3
} else {
  # bloc d' instructions 4
}
```

Structure conditionnelle : if/else

La syntaxe formelle d'un if /else est la suivante :

```
if( condition ) {
  # bloc d' instructions 1

}else{
  # bloc d' instructions 2
}
```

La clause else est facultative. L'écriture ci-dessous est tout à fait valide :

```
if ( x<0 ) {
  x <- -x
}
print(x)
```

Les if /else peuvent être imbriquées comme suit :

```
if (cond1) {
  # bloc d' instructions 1
} else if (cond2) {
  # bloc d' instructions 2
} else if(cond3) {
  # bloc d' instructions 3
} else {
  # bloc d' instructions 4
}
```

Structure répétitive : while

La boucle TANT QUE (WHILE en anglais) est un peu comme une structure si répétitive. Son écriture en R est la suivante :

```
while( condition ) {
   #instructions
}
```

Structure répétitive : while

La boucle TANT QUE (WHILE en anglais) est un peu comme une structure si répétitive. Son écriture en R est la suivante :

```
while( condition ) {
   #instructions
}
```

Exemple :

```
v <- c(2, 34, 6.78, 9.2)
i <- 1
while( i<=length(v) ) {
   print( v[i] )
   i <- i+1
}
```

Cet exemple illustre le parcours d'un tableau en R en réalisant l'affichage.

Structure répétitive : while

La boucle TANT QUE (WHILE en anglais) est un peu comme une structure si répétitive. Son écriture en R est la suivante :

```
while( condition ) {
   #instructions
}
```

Exemple :

```
v <- c(2, 34, 6.78, 9.2)
i <- 1
while( i<=length(v) ) {
   print( v[i] )
   i <- i+1
}
```

Cet exemple illustre le parcours d'un tableau en R en réalisant l'affichage.

Structure répétitive : while

Le déroulement normal d'une boucle peut être interrompu à l'aide des instructions suivantes :

- break : sort immédiatement de la boucle,
- next : arrête le traitement de l'itération courante pour revenir directement en haut du while.

Structure répétitive : while

Le déroulement normal d'une boucle peut être interrompu à l'aide des instructions suivantes :

- break : sort immédiatement de la boucle,
- next : arrête le traitement de l'itération courante pour revenir directement en haut du while.

Exemple :

```
x <- c(37, -6, 41, 0, 10)
i <- 1
while( i < length(x) ) {
  if ( x[i] == 0 ) {
    break
  }
  if ( x[i] < 0 ) {
    i <- i+1 #Attention : ne pas oublier d'incrementer i avant le next !!
    next
  }
  print( x[i] )

  i <- i+1
}
```

Ce programme affiche : 37 et 41.

Structure répétitive : while

Le déroulement normal d'une boucle peut être interrompu à l'aide des instructions suivantes :

- break : sort immédiatement de la boucle,
- next : arrête le traitement de l'itération courante pour revenir directement en haut du while.

Exemple :

```
x <- c(37, -6, 41, 0, 10)
i <- 1
while( i < length(x) ) {
  if ( x[i] == 0 ) {
    break
  }
  if ( x[i] < 0 ) {
    i <- i+1 #Attention : ne pas oublier d'incrementer i avant le next !!
    next
  }
  print( x[i] )

  i <- i+1
}
```

Ce programme affiche : 37 et 41.

La structure répétitive : for

La structure for est une structure qui permet de boucler en faisant évoluer une variable, appelée itérateur, sur un ensemble de valeurs données.

```
for( i in v ) {
  #instructions pour traiter i
}
```

où :

- v est un vecteur définissant les valeurs à parcourir (valeurs successivement prises par la variable i,
- i est l'itérateur : il prend successivement, à chaque tour de boucle, les valeurs du vecteur v.

La structure répétitive : for

La structure for est une structure qui permet de boucler en faisant évoluer une variable, appelée itérateur, sur un ensemble de valeurs données.

```
for( i in v ) {
  #instructions pour traiter i
}
```

où :

- v est un vecteur définissant les valeurs à parcourir (valeurs successivement prises par la variable i,
- i est l'itérateur : il prend successivement, à chaque tour de boucle, les valeurs du vecteur v.

Exemple :

```
for( i in c(3.4, 5.6, 7) ) {
  print(i)
}
```

Ce programme affiche : 3.4, 5.6 et 7.

La structure répétitive : for

La structure for est une structure qui permet de boucler en faisant évoluer une variable, appelée itérateur, sur un ensemble de valeurs données.

```
for( i in v ) {
  #instructions pour traiter i
}
```

où :

- v est un vecteur définissant les valeurs à parcourir (valeurs successivement prises par la variable i,
- i est l'itérateur : il prend successivement, à chaque tour de boucle, les valeurs du vecteur v.

Exemple :

```
for( i in c(3.4, 5.6, 7) ) {
  print(i)
}
```

Ce programme affiche : 3.4, 5.6 et 7.

Parcours d'une séquence d'entiers

Le cas le plus classique d'utilisation est un itérateur qui parcourt une séquence d'entiers.

```
for( it in 1:100 ) {
  print(it)
}
```

Ce programme a une équivalence directe avec un while :

```
it <- 1
while( it <=100 ){
  print(it)
  it <- it+1
}
```

En programmation, ceci correspond à un besoin très courant : les parcours de vecteurs et de matrices. Tous les parcours de tableaux ou de matrice peuvent être simplement écrits avec des for.

Définition d'une fonction

La définition d'une nouvelle fonction suit la syntaxe suivante :

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

avec

- name est le nom que l'on décide de donner à la fonction,
- arguments sont les paramètres de la fonction,
- expression est le corps de la fonction

Définition d'une fonction

La définition d'une nouvelle fonction suit la syntaxe suivante :

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

avec

- name est le nom que l'on décide de donner à la fonction,
- arguments sont les paramètres de la fonction,
- expression est le corps de la fonction

La fonction est appelée à l'aide de l'expression name().
Par exemple

```
carre <- function(x) x^2

# Le carré de 2
carre(2)

## [1] 4

# Le carré de -3
carre(-3)

## [1] 9
```

Définition d'une fonction

La définition d'une nouvelle fonction suit la syntaxe suivante :

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

avec

- name est le nom que l'on décide de donner à la fonction,
- arguments sont les paramètres de la fonction,
- expression est le corps de la fonction

La fonction est appelée à l'aide de l'expression name().
Par exemple

```
carre <- function(x) x^2

# Le carré de 2
carre(2)

## [1] 4

# Le carré de -3
carre(-3)

## [1] 9
```

Le corps d'une fonction est constitué d'une seule ou plusieurs instructions. Il est nécessaire de les entourner par des accolades. Le résultat est la valeur de la dernière commande contenue dans le corps de la fonction.

```
f <- function(x) {
  x^2
  y <- x
  y
}
f(2)
```

Le corps d'une fonction est constitué d'une seule ou plusieurs instructions. Il est nécessaire de les entourner par des accolades. Le résultat est la valeur de la dernière commande contenue dans le corps de la fonction.

```
f <- function(x) {
  x^2
  y <- x
  y
}
f(2)
```

Si on souhaite retourner une valeur autre part qu'à la dernière ligne, il faut utiliser la fonction return().

```
f <- function(x) {
  return(x^2)
  # Un commentaire de dernière ligne
}
f(2)
```

Il est possible de retourner une liste, pouvant contenir autant d'objet que l'on souhaite.

```
# Calculer la moyenne et l'écart-type pour un vecteur
stat_des <- function(x) {
  list(moyenne = mean(x), ecart_type = sd(x))
}
x <- runif(10)
stat_des(x)
```

Le corps d'une fonction est constitué d'une seule ou plusieurs instructions. Il est nécessaire de les entourner par des accolades. Le résultat est la valeur de la dernière commande contenue dans le corps de la fonction.

```
f <- function(x) {
  x^2
  y <- x
  y
}
f(2)
```

Si on souhaite retourner une valeur autre part qu'à la dernière ligne, il faut utiliser la fonction return().

```
f <- function(x) {
  return(x^2)
  # Un commentaire de dernière ligne
}
f(2)
```

Il est possible de retourner une liste, pouvant contenir autant d'objet que l'on souhaite.

```
# Calculer la moyenne et l'écart-type pour un vecteur
stat_des <- function(x) {
  list(moyenne = mean(x), ecart_type = sd(x))
}
x <- runif(10)
stat_des(x)
```

Exemples :

Exemple 1 : Variable locale d'une fonction

```
rm( list =ls() )

a <- c(4, 5, 6)

f <- function() {
  a <- 3
  print( a )
}

f();
print( a );
```

Ce programme affiche : 3 et 4 5 6.
Dans cet exemple, on commence par définir une variable a et une fonction f dans la session R.

- La fonction f modifie la valeur d'une variable locale a et l'affiche.
- Cette variable n'est pas la même que celle qui a été définie dans la session R de sorte que si on appelle la fonction puis qu'on affiche a (dans la session R) alors l'affichage obtenu est l'affichage de la variable a d'origine.
- En résumé, la modification de a dans la fonction est sans effet pour a dans la session.

Exemples :

Exemple 2 : Passage de paramètres et instruction return

```
#declaration de la fonction
MaFonction <- function(p1)
{
  a <- 3*p1
  return(a)
}

#appel de fonction
v <- Mafonction(34)
print(v)
```

```
> var <- 10
> a <- MaFonction( var )
> print( a )
[1]  30
> a <- MaFonction( 1:5 )
> print(a)
[1]  3  6  9  12  15
```

Exercice 1 : (Boucle while)

1. À l'aide de la fonction while(), créer une boucle qui permet de calculer la factorielle d'un nombre ;
2. Réutiliser le code de la question précédente pour en faire une fonction qui, lorsqu'on lui donne un nombre, retourne sa factorielle. Comparer le résultat avec la fonction factorial().

Exercice 1 : (Boucle while)

1. À l'aide de la fonction while(), créer une boucle qui permet de calculer la factorielle d'un nombre ;
2. Réutiliser le code de la question précédente pour en faire une fonction qui, lorsqu'on lui donne un nombre, retourne sa factorielle. Comparer le résultat avec la fonction factorial().

1)

```
i <- 10
res <- 1
while(i>1){
  res <- res * i
  i <- i-1
}
```

Exercice 1 : (Boucle while)

1. À l'aide de la fonction while(), créer une boucle qui permet de calculer la factorielle d'un nombre ;
2. Réutiliser le code de la question précédente pour en faire une fonction qui, lorsqu'on lui donne un nombre, retourne sa factorielle. Comparer le résultat avec la fonction factorial().

1)

```
i <- 10
res <- 1
while(i>1){
  res <- res * i
  i <- i-1
}
```

Exercice 1 : (Boucle while)

1. À l'aide de la fonction while(), créer une boucle qui permet de calculer la factorielle d'un nombre ;
2. Réutiliser le code de la question précédente pour en faire une fonction qui, lorsqu'on lui donne un nombre, retourne sa factorielle. Comparer le résultat avec la fonction factorial().

1)

```
i <- 10
res <- 1
while(i>1){
  res <- res * i
  i <- i-1
}
```

2)

```
factorielle <- function(x){
  res <- 1
  while(x>1){
    res <- res * x
    x <- x-1
  }
  res
}# Fin de factorielle()
```

Exercice 1 : (Boucle while)

1. À l'aide de la fonction while(), créer une boucle qui permet de calculer la factorielle d'un nombre ;
2. Réutiliser le code de la question précédente pour en faire une fonction qui, lorsqu'on lui donne un nombre, retourne sa factorielle. Comparer le résultat avec la fonction factorial().

1)

```
i <- 10
res <- 1
while(i>1){
  res <- res * i
  i <- i-1
}
```

2)

```
factorielle <- function(x){
  res <- 1
  while(x>1){
    res <- res * x
    x <- x-1
  }
  res
}# Fin de factorielle()
```

Exercice 2 : (Boucles while et for)

1. Choisir un nombre mystère entre 1 et 100, et le stocker dans un objet que l'on nommera nombre.mystere. Ensuite, créer une boucle qui à chaque itération effectue un tirage aléatoire d'un entier compris entre 1 et 100. Tant que le nombre tiré est différent du nombre mystère, la boucle doit continuer. À la sortie de la boucle, une variable que l'on appellera nb.tirages contiendra le nombre de tirages réalisés pour obtenir le nombre mystère ;

Exercice 2 : (Boucles while et for)

1. Choisir un nombre mystère entre 1 et 100, et le stocker dans un objet que l'on nommera nombre.mystere. Ensuite, créer une boucle qui à chaque itération effectue un tirage aléatoire d'un entier compris entre 1 et 100. Tant que le nombre tiré est différent du nombre mystère, la boucle doit continuer. À la sortie de la boucle, une variable que l'on appellera nb.tirages contiendra le nombre de tirages réalisés pour obtenir le nombre mystère ;

2. Utiliser le code de la question précédente pour réaliser la fonction trouver.nombre, qui, lorsqu'on lui donne un nombre compris entre 1 et 100, retourne le nombre de tirages aléatoires d'entiers compris entre 1 et 100 nécessaires avant de tirer le nombre mystère ;

Exercice 2 : (Boucles while et for)

1. Choisir un nombre mystère entre 1 et 100, et le stocker dans un objet que l'on nommera nombre.mystere. Ensuite, créer une boucle qui à chaque itération effectue un tirage aléatoire d'un entier compris entre 1 et 100. Tant que le nombre tiré est différent du nombre mystère, la boucle doit continuer. À la sortie de la boucle, une variable que l'on appellera nb.tirages contiendra le nombre de tirages réalisés pour obtenir le nombre mystère ;

2. Utiliser le code de la question précédente pour réaliser la fonction trouver.nombre, qui, lorsqu'on lui donne un nombre compris entre 1 et 100, retourne le nombre de tirages aléatoires d'entiers compris entre 1 et 100 nécessaires avant de tirer le nombre mystère ;

Exercice 2 : (Boucles while et for)

1. Choisir un nombre mystère entre 1 et 100, et le stocker dans un objet que l'on nommera nombre.mystere. Ensuite, créer une boucle qui à chaque itération effectue un tirage aléatoire d'un entier compris entre 1 et 100. Tant que le nombre tiré est différent du nombre mystère, la boucle doit continuer. À la sortie de la boucle, une variable que l'on appellera nb.tirages contiendra le nombre de tirages réalisés pour obtenir le nombre mystère ;

2. Utiliser le code de la question précédente pour réaliser la fonction trouver.nombre, qui, lorsqu'on lui donne un nombre compris entre 1 et 100, retourne le nombre de tirages aléatoires d'entiers compris entre 1 et 100 nécessaires avant de tirer le nombre mystère ;

3. En utilisant une boucle for, faire appel 1000 fois à la fonction trouver.nombre() qui vient d'être créée. À chaque itération, stocker le résultat dans un élément d'un vecteur que l'on appellera nb.essais.rep. Enfin, afficher la moyenne du nombre de tirages nécessaires pour retrouver le nombre magique.

```
nb_essais_rep <- rep(NA, 1000)
```

Exercice 2 : (Boucles while et for)

1. Choisir un nombre mystère entre 1 et 100, et le stocker dans un objet que l'on nommera nombre.mystere. Ensuite, créer une boucle qui à chaque itération effectue un tirage aléatoire d'un entier compris entre 1 et 100. Tant que le nombre tiré est différent du nombre mystère, la boucle doit continuer. À la sortie de la boucle, une variable que l'on appellera nb.tirages contiendra le nombre de tirages réalisés pour obtenir le nombre mystère ;

2. Utiliser le code de la question précédente pour réaliser la fonction trouver.nombre, qui, lorsqu'on lui donne un nombre compris entre 1 et 100, retourne le nombre de tirages aléatoires d'entiers compris entre 1 et 100 nécessaires avant de tirer le nombre mystère ;

3. En utilisant une boucle for, faire appel 1000 fois à la fonction trouver.nombre() qui vient d'être créée. À chaque itération, stocker le résultat dans un élément d'un vecteur que l'on appellera nb.essais.rep. Enfin, afficher la moyenne du nombre de tirages nécessaires pour retrouver le nombre magique.

```
nb_essais_rep <- rep(NA, 1000)
```

Solution exercice 2 : (Boucles while et for)

1)

```
nombre_mystere <- 30

nb_tirages <- 0
while(sample(x = seq_len(100), size = 1) != nombre_mystere){
  nb_tirages <- nb_tirages + 1
}
nb_essais
```

2)

```
trouver_nombre <- function(x){
  nb_tirages <- 1
  while(sample(x = seq_len(100), size = 1) != x){
    nb_tirages <- nb_tirages + 1
  }
  nb_tirages
}
```

3)

```
for(i in seq_len(1000)) nb_essais_rep[i] <- trouver_nombre(10)

mean(nb_essais_rep)
```

Exercice 3 : (Suite de Fibonacci)

Utiliser une boucle for pour reproduire la suite de Fibonacci jusqu'à son dixième terme (la séquence F_n est définie par la relation de récurrence suivante :

$$F_n = F_{n-1} + F_{n-2};$$

les valeurs initiales sont : $F_0 = 0$ et $F_1 = 1$).

Exercice 3 : (Suite de Fibonacci)

Utiliser une boucle for pour reproduire la suite de Fibonacci jusqu'à son dixième terme (la séquence F_n est définie par la relation de récurrence suivante :

$$F_n = F_{n-1} + F_{n-2};$$

les valeurs initiales sont : $F_0 = 0$ et $F_1 = 1$).

Solution :

```
end <- 10
res <- rep(NA, end)
res[1] <- 0
res[2] <- 1
for(i in 3:end){
  res[i] <- res[i-1] + res[i-2]
}
res
```

Exercice 3 : (Suite de Fibonacci)

Utiliser une boucle for pour reproduire la suite de Fibonacci jusqu'à son dixième terme (la séquence F_n est définie par la relation de récurrence suivante :

$$F_n = F_{n-1} + F_{n-2};$$

les valeurs initiales sont : $F_0 = 0$ et $F_1 = 1$).

Solution :

```
end <- 10
res <- rep(NA, end)
res[1] <- 0
res[2] <- 1
for(i in 3:end){
  res[i] <- res[i-1] + res[i-2]
}
res
```

Exercice 4 :

Dans l'exercice ci-dessous, écrire une fonction R pour faire le calcul demandé.

Étant donné un vecteur d'observations $\mathbf{x} = (x_1, \ldots, x_n)$ et un vecteur de poids correspondants $\mathbf{w} = (w_1, \ldots, w_n)$, 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

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

et

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

Exercice 4 :

Dans l'exercice ci-dessous, écrire une fonction R pour faire le calcul demandé.

Étant donné un vecteur d'observations $\mathbf{x} = (x_1, \ldots, x_n)$ et un vecteur de poids correspondants $\mathbf{w} = (w_1, \ldots, w_n)$, 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

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

et

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

Solution :

```
mp <- function(x,w){
  return(sum(w*x)/sum(w))
}
x <- c(7, 13, 3, 8, 12, 12, 20, 11)
w <- c (0.15, 0.04, 0.05, 0.06, 0.17, 0.16, 0.11, 0.09)
mp(x,w)
```

Exercice 4 :

Dans l'exercice ci-dessous, écrire une fonction R pour faire le calcul demandé.

Étant donné un vecteur d'observations $\mathbf{x} = (x_1, \ldots, x_n)$ et un vecteur de poids correspondants $\mathbf{w} = (w_1, \ldots, w_n)$, 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

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

et

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

Solution :

```
mp <- function(x,w){
  return(sum(w*x)/sum(w))
}
x <- c(7, 13, 3, 8, 12, 12, 20, 11)
w <- c (0.15, 0.04, 0.05, 0.06, 0.17, 0.16, 0.11, 0.09)
mp(x,w)
```

Exercice 5 :

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

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

alors que l'estimateur biaisé est

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

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

Exercice 5 :

Solution :

```
variance <- function(x,biased=FALSE){
  n <- length(x)
  m <- mean(x)
  if(biased == FALSE){
    s2 <- sum((x-m)^2)/(n-1)
    }else{
    s2 <- sum((x-m)^2)/n
    }
  return(s2)
}
x <- rnorm(1000) #1000 valeurs tirés selon la loi normale centrée et réduite
variance(x)
var(x)
variance(x, biased=TRUE)
```

Exercice 6 :

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`.

Exercice 6 :

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`.

Solution :

```
phi <- function(x)
{
    exp(-x^2/2) / sqrt(2 * pi)
}
```

Exercice 6 :

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`.

Solution :

```
phi <- function(x)
{
    exp(-x^2/2) / sqrt(2 * pi)
}
```