INITIATION À R

HERVÉ PERDRY

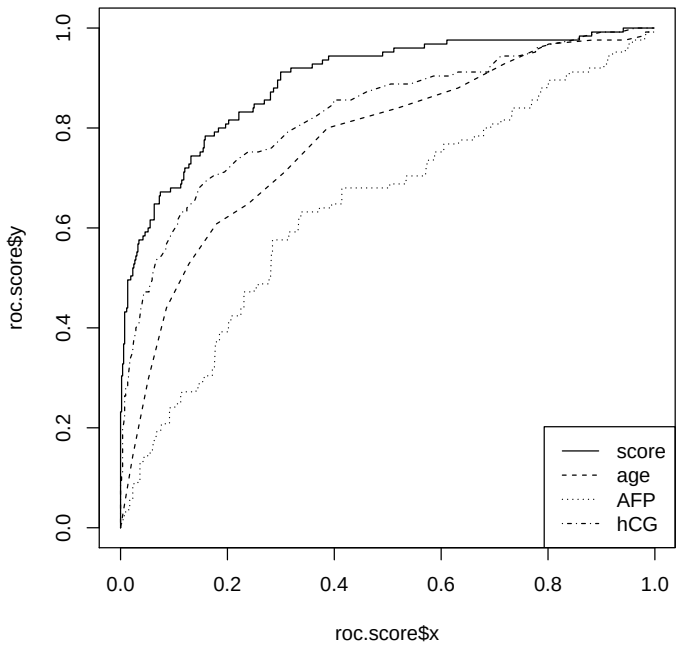


Table des matières

1 Premier contact avec R

R est un logiciel libre et gratuit, utilisé pour le traitement de données statistiques. Vous pouvez l'installer sur votre ordinateur personnel sans délier bourse ni vous livrer à quelque manœuvre illicite ! Voyez le site de R : http://r-project.org.

Le caractère « libre » de R fait sa grande popularité dans le domaine universitaire. De nouvelles méthodes sont programmées sans cesse en R, en faisant un outil toujours plus riche (cf http://cran.r-project.org). Par suite, son utilisation dans les entreprises privées est de plus en plus importante également ; il concurrence les logiciels SAS et STATA.

1.1 Documentation

Beaucoup de documentation est fournie avec R, en particulier un document appelé *An introduction to R*, beaucoup plus complet que celui-ci. Sa lecture est chaudement recommandée, ainsi que celle du très bon poly « R pour les débutants » d'Emmanuel Paradis. L'aide mémoire du site duclert.org peut également être utile. Vous trouverez facilement sur internet des forums d'entraide d'utilisateurs R.

Des pages d'aides sont également inclues dans le logiciel. Cf section 1.4 pour une lecture guidée d'une de ces pages.

Le présent document est très incomplet : non seulement nous ne présentons qu'une infime partie des commandes R, mais nous ne présentons pas non plus tous les détails de l'utilisation de ces commandes. Regarder leur page de manuel est un exercice implicite tout au long de ce document.

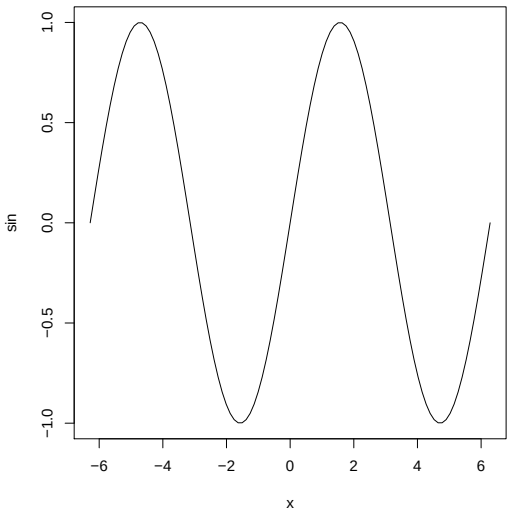
1.2 Calculs en ligne de commande, utilisation de variables

Le débutant commencera par se familiariser à R à travers son interface dite de « ligne de commande ». On commencera par s'en servir comme calculette :

```
> 1+1
[1] 2
> 3*7
[1] 21
> pi
[1] 3.141593
> sin(pi/4)
[1] 0.7071068
> sqrt(2)/2
[1] 0.7071068
```

Et même comme calculette graphique :

```
> plot(sin, xlim=c(-2*pi,2*pi))
```



1.2.1 Utiliser des variables

On peut stocker le résultat d'un calcul dans une variable. L'opérateur d'affectation est une petite flèche : `<-`

```
> a <- 2
> a
[1] 2
```

Toute affectation détruit la valeur précédente (en jargon d'informaticien, on dit que la valeur précédente est *écrasée* par la nouvelle) :

```
> a <- 1
> a
[1] 1
```

La petite flèche existe aussi dans l'autre sens : `->`

```
> 3+2 -> a
> a
[1] 5
```

On peut utiliser utiliser la variable `a` dans un calcul, par exemple :

```
> b <- a + a
> b
[1] 10
```

Ici R commence par remplacer `a + a` par sa valeur ; rappelons que `a` défini juste au-dessus contient `5` ; l'opération `b <- a + a` est donc évaluée comme `b <- 10`.

Exercice : que contient `a` après les commandes suivantes :

```
> a <- 4
> a <- a+1
```

Solution : Comme ci-dessus, R remplace `a+1` par sa valeur, soit `4+1`, soit `5`, et l'opération `a <- a+1` est évaluée comme `a <- 5`.

Remarque : Le point-virgule permet de séparer des commandes sur une même ligne :

```
> a <- 5; a <- a+1; a <- 2*a; a <- a+1;
> a
[1] 13
```

Exercice : Assurez-vous de bien comprendre le résultat ci-dessus.

1.2.2 Type d'une variable (ou d'un objet)

La notion de *type* d'un objet est illustrée par les exemples suivants :

```
> a <- 5.4
> b <- 1:5
> c <- 6
> Prenom <- "Marie-Charlotte"
> z <- TRUE
> typeof(a)
[1] "double"
> typeof(b)
[1] "integer"
> typeof(c)
[1] "double"
> typeof(z)
[1] "logical"
> typeof(Prenom)
[1] "character"
```

Ainsi, un nombre est de type `double` ou `integer`, une chaîne de caractères de type `character` ; notez enfin l'apparition de `TRUE` et `FALSE`, dont le type est `logical` :

```
> typeof(FALSE)
[1] "logical"
```

Que, dans l'exemple ci-dessus, le nombre `6` soit de type `double` et non `integer` peut surprendre. Il faut utiliser la syntaxe `6L` pour obtenir un `integer` :

```
> typeof(6L)
[1] "integer"
```

On peut effectuer des conversions d'un type à l'autre avec des commandes du type `as.quelquechose()`...

```
> as.character(10)
[1] "10"
> "100.01" -> x
> x
[1] "100.01"
> as.double(x)
[1] 100.01
> as.integer(x)
[1] 100
```

La conversion du type `logical` vers `integer` (ou `double`) nous intéressera tout particulièrement :

```
> as.integer(TRUE)
[1] 1
> as.integer(FALSE)
[1] 0
```

De plus, dès qu'on utilise des opérateurs arithmétiques sur des objets de type `logical` la conversion est implicite :

```
> 1 + TRUE
```

```
[1] 2
> TRUE + TRUE + FALSE
[1] 2
> typeof(1 + TRUE)
[1] "double"
> typeof(TRUE + TRUE + FALSE)
[1] "integer"
```

Note La fonction `mode` donne des informations un peu différentes, le mode `numeric` rassemblant les types `integer` et `double`. La fonction `as.numeric()` est synonyme de `as.double()`, et l'auteur l'emploie régulièrement en lieu et place de cette dernière.

1.2.3 Les vecteurs

Nous avons jusqu'à présent créé des variables contenant une seule valeur. R est en fait conçu pour travailler principalement sur des *vecteurs*, en langage familier des « listes » de valeurs. Il est important de noter que dans un vecteur on ne mélange pas des nombres et des chaînes de caractères, toutes les valeurs d'un vecteur ont le même type – le type du vecteur.

En fait, toutes les variables que nous avons créées jusqu'ici sont des vecteurs de longueur 1 !

Créer des vecteurs

On crée un vecteur d'entiers ou de `double` de taille donnée avec les fonctions `integer()` ou `double()` :

```
> integer(5)
[1] 0 0 0 0 0
> double(5)
[1] 0 0 0 0 0
> integer(5) -> a
```

L'objet `a` contient 5 valeurs, numérotées de 1 à 5 ; ce numéro est appelé *indice* ; e vecteur nouvellement créé contient des `0`. On accède et on modifie une de ces valeurs, par exemple la seconde, en utilisant des crochets : `a[2]`

```
> a[2] <- 9L
> a
[1] 0 9 0 0 0
> typeof(a)
[1] "integer"
```

Les fonctions `character()` et `logical()` permettent de créer des vecteurs de types respectifs `character` et `logical`.

La fonction `numeric()` est synonyme de `double()`, et ainsi qu'on l'a déjà signalé pour `as.numeric()`, elle sera souvent employée dans la suite.

Concaténation de vecteurs

La *concaténation* est l'opération qui consiste à coller des vecteurs bout-à-bout pour en créer de nouveaux. Elle se réalise avec la fonction `c()` :

```
> b <- numeric(2)
> b[1] <- 1
> b[2] <- 21
> b
[1]  1 21
> c(a,b)
```

```
[1]  0  9  0  0  0  1 21
```

Une valeur isolée étant un vecteur de longueur 1, on peut créer des vecteurs par concaténation de scalaires :

```
> y <- c(1,2,6,7,0,-1)
> y
[1]  1  2  6  7  0 -1
```

1.3 Quelques commandes pour gérer la session courante

1.3.1 Les objets créés

On obtient la liste des objets qui ont été créés avec la commande `ls()` :

```
> ls()
[1] "a"      "b"      "c"      "Prenom" "x"      "y"      "z"
```

Remarque en passant : des parenthèses après un nom indiquent à R qu'il s'agit du nom d'une fonction, et qu'il doit exécuter cette fonction. Ainsi, si vous tapez `lapin(2)`, R cherche s'il existe une fonction nommée `lapin`. S'il existe un objet nommé `lapin` mais que ce n'est pas une fonction, R protestera avec la dernière énergie :

```
> lapin <- "Oryctolagus cuniculus"
> lapin(1.618)
Erreur : impossible de trouver la fonction "lapin"
```

Que se passe-t-il si on omet les parenthèses après le nom d'une fonction qui existe, comme `ls`? **Essayez!**
La fonction `ls.str()` donne également le contenu (ou un aperçu du contenu) des variables.

```
> ls.str()
a :  int [1:5] 0 9 0 0 0
b :  num [1:2] 1 21
c :  num 6
lapin :  chr "Oryctolagus cuniculus"
Prenom :  chr "Marie-Charlotte"
x :  chr "100.01"
y :  num [1:6] 1 2 6 7 0 -1
z :  logi TRUE
```

Et `rm()` permet d'effacer des variables :

```
> ls()
[1] "a"      "b"      "c"      "lapin"  "Prenom" "x"      "y"      "z"
> rm("a")
> ls()
[1] "b"      "c"      "lapin"  "Prenom" "x"      "y"      "z"
> rm("Prenom")
> ls()
[1] "b"     "c"     "lapin" "x"     "y"     "z"
> rm(list = c("b","x"))
> ls()
[1] "c"     "lapin" "y"     "z"
```

Enfin, pour effacer toutes les variables, on procédera ainsi :

```
> rm(list=ls())
> ls()
character(0)
```

Remarque : Cette dernière valeur, `character(0)`, est simplement un vecteur de type `character` et de longueur nulle.

1.3.2 Sauver des objets dans un fichier, relire un fichier

Il faut tout d'abord, avant d'écrire un fichier, modifier le répertoire courant, ou répertoire de travail (*working directory*) : c'est le rôle de la fonction `setwd()`. La fonction `getwd()` permet d'obtenir la valeur du répertoire courant.
Sous Windows, on peut changer de répertoire de travail « à la souris », ce qui peut être perçu comme considérablement plus simple : menu Fichier > Changer le répertoire courant.

Voyons comment sauver deux variables dans un fichier :

```
> x <- c(1,2,3)
> y <- c(10,100,1000)
> save(x,y, file="data.rda")
```

Effaçons maintenant toutes les variables définies dans la session

```
> rm(list=ls())
> ls()
character(0)
```

Et relisons le fichier :

```
> load("data.rda")
> ls()
[1] "x" "y"
> x
[1] 1 2 3
> y
[1]   10  100 1000
```

La fonction `save.image()` est un raccourci pour la sauvegarde de toutes les variables définies. C'est ce qui est proposé en fin de session, quand vous fermez R ; dans ce cas R crée un fichier « caché » `.RData`, ainsi qu'un fichier `.Rhistory` pour l'historique des commandes (cf plus loin).

Exercice : Essayez de regarder le contenu du fichier `data.rda` avec un éditeur de texte (sous windows : le bloc-note). Est-ce compréhensible ? Il existe des fonctions qui permettent d'écrire un fichier texte lisible (et éditable) :

```
> write(x, "donnees.txt")
```

Inspectez le contenu de `donnees.txt` : vous y retrouverez les valeurs `1 2 3`. Pour relire un tel fichier, la fonction `scan()` s'impose :

```
> scan("donnees.txt")
[1] 1 2 3
```

La fonction `read.table()` qui permet de lire des fichiers de données structurés en tables, qui sera la plus utile pour les analyses de données statistiques.

1.3.3 Historique des commandes tapées

Vous avez peut-être remarqué que la touche ↑ permet de « rappeller » les dernières commandes tapées. Vous pouvez également les obtenir avec `history()`. On peut les sauver dans un fichier avec `savehistory()`, et les relire avec `loadhistory()`. C'est proposé en fin de session en même temps que la sauvegarde des variables définies, dans ce cas R crée un fichier « caché » `.Rhistory` en même temps que `.RData` qui contient les variables définies dans la session.

1.4 Lire une page d'aide

Pour terminer ce chapitre, voici comment obtenir de l'aide sur une commande de R, par exemple `sd` :

```
> ?sd
> help("sd")
```

Certains mots clefs doivent être entre guillemets quand on utilise le point d'interrogation :

```
> ?"for"
```

Notez que les pages d'aides ont toute la même structure ; d'abord une courte description :

```
Description:

     This function computes the standard deviation of the values in
     'x'.  If 'na.rm' is 'TRUE' then missing values are removed before
     computation proceeds.
```

Nous apprenons notament que cette fonction permet de calculer la *standard deviation* des valeurs contenues dans `x`, c'est-à-dire l'écart-type. Ensuite, la section `Usage` donne la façon d'appeler la fonction :

```
Usage:

     sd(x, na.rm = FALSE)
```

On note en particulier que le paramètre `na.rm` a une valeur par défaut, qui est `FALSE` ; on peut donc taper simplement `sd(x)`, qui sera équivalent à `sd(x, na.rm = FALSE)`. Les paramètres ont souvent un nom informatif, ici avec un peu d'habitude ont pourra décoder `na.rm` en *NA remove* : supprimer les `NA`.

Vient ensuite une section `Arguments`, qui décrit ce qu'on doit mettre dans les paramètres en question, et une section `Details` qui nous en dit davantage sur le rôle de la fonction.

La section `See also` devrait particulièrement retenir votre attention :

```
See Also:

     'var' for its square, and 'mad', the most robust alternative.
```

Elle nous oriente sur des fonctions en rapport avec celle que nous regardons, ici en particulier `var` qui calcule la variance.

Enfin, la section `Examples` permet souvent d'y voir plus clair, en fournissant des exemples pertinents.

Voyez aussi les commandes suivantes, qui permettent une recherche par mot-clef dans l'aide :

```
> help.search("mean")
> ??"mean"
> apropos("mean")
```

2 Vecteurs

Nous allons approfondir notre connaissance des vecteurs : arithmétique des vecteurs, tests, fonctions...

2.1 Vecteurs

Souvenons-nous des façons simples de créer un vecteur :

```
> numeric(3)
[1] 0 0 0
> double(4)
[1] 0 0 0 0
> c(1,2,4)
[1] 1 2 4
```

On l'a dit, un vecteur ne peut contenir des valeurs que d'un seul type. Si on essaie de passer outre, R va effectuer des conversions arbitraires :

```
> a <- 1:5
> typeof(a)
[1] "integer"
> typeof( c(a,7) )
[1] "double"
> c(TRUE,1,"a")
[1] "TRUE" "1"    "a"
```

2.1.1 Création de vecteurs particuliers

Le raccourci suivant est très utile pour créer des vecteurs en comptant de 1 en 1 :

```
> 1:4
[1] 1 2 3 4
> 4:1
[1] 4 3 2 1
> -1:3
[1] -1  0  1  2  3
> -(1:3)
[1] -1 -2 -3
```

La fonction `seq()` permet de progresser par pas arbitraires, en fournissant soit le pas, soit la longueur désirée :

```
> seq(from = 1, to = 3, by = 0.2)
 [1] 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0
> seq(1, 3, by = 0.2)
 [1] 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0
> seq(1, 3, length = 7)
[1] 1.000000 1.333333 1.666667 2.000000 2.333333 2.666667 3.000000
```

La fonction `rep()` permet de répéter les valeurs contenues dans un vecteur :

```
> rep(1, 3)
[1] 1 1 1
> rep(1:3, 2)
[1] 1 2 3 1 2 3
> rep(1:3, each = 2)
[1] 1 1 2 2 3 3
```

Présentons enfin la fonction `sample(x, size)` qui permet de réaliser des tirages aléatoires dans un vecteur. Tirage « sans remise » :

```
> sample(0:9, 5)
[1] 1 2 4 8 6
> sample(0:9, 10)
 [1] 9 8 5 4 7 6 0 3 1 2
```

Tirage « avec remise »

```
> sample(0:9, 10, replace=TRUE)
 [1] 3 2 5 2 8 2 9 2 9 2
> sample(0:9, 20, replace=TRUE)
 [1] 8 0 5 7 8 8 8 0 8 5 3 5 4 1 8 2 8 5 5 0
```

Exercice : Lire l'aide de `sample`, et s'assurer de bien avoir compris le rôle du paramètre facultatif `prob`. Dans un (grand) étang, il y a 80% de truites et 20% de brochets. On y pêche 20 poissons : simuler cette opération avec `sample`.

2.1.2 Extraction d'une partie d'un vecteur

On se souvient de l'indiçage d'un vecteur :

```
> y <- sample(0:9, 6)
> y
[1] 3 7 1 4 5 6
> y[1]
[1] 3
> y[2]
[1] 7
> y[2] <- 10
> y
[1]  3 10  1  4  5  6
```

Mais on peut également extraire plusieurs valeurs d'un coup, en fournissant les indices des éléments qu'on veut extraire.

```
> y[1:3]
[1]  3 10  1
> y[c(1,3)]
[1] 3 1
```

Et cela fonctionne également pour affecter des valeurs :

```
> y[c(1,3)] <- c(0,20);
> y
[1]  0 10 20  4  5  6
```

Recyclage de valeurs

Ici on a donné un vecteur de longueur deux pour remplacer deux valeurs de `y`. Si on donne une seule valeur, elle est recyclée :

```
> y[1:4] <- 2
> y
[1] 2 2 2 2 5 6
```

On peut également, pour remplacer quatre valeurs, en fournir deux, qui seront pareillement recyclées :

```
> y[1:4] <- c(1,3)
> y
[1] 1 3 1 3 5 6
```

Voici enfin ce qui se produit quand on fournit un nombre de valeurs qui n'est pas un diviseur du nombre de valeurs à remplacer :

```
> a <- 1:10
> a[1:5] <- c(1,0)
Message d'avis :
In a[1:5] <- c(1, 0) :
  le nombre d'objets à remplacer n'est pas multiple de la taille du remplacement
> a
 [1]  1  0  1  0  1  6  7  8  9 10
```

Le recyclage a lieu, mais un message d'avertissement est affiché.

Exclure une partie d'un vecteur

Les indices négatifs sont interprétés comme des demandes d'exclusion :

```
> y
[1] 1 3 1 3 5 6
> y[-1]
[1] 3 1 3 5 6
> y[-3:-5]
[1] 1 3 6
```

L'existence de cette syntaxe va de pair avec le fait que les indices des vecteurs en R commencent à 0 !

Utilisation de vecteurs booléens

Enfin, une autre façon importante d'extraire une partie d'un vecteur est de fournir un vecteur de type `logical` (appelé vecteur booléen), de même longueur que le vecteur. On extrait ainsi les valeurs du vecteur aux indices où on a `TRUE`; rien ne vaut un exemple :

```
> y
[1] 1 3 1 3 5 6
> y[ c(TRUE, FALSE, TRUE, TRUE, FALSE, TRUE) ]
[1] 1 1 3 6
```

Si on donne un vecteur booléen plus court, ses valeurs sont recyclées :

```
> y[ c(TRUE,FALSE) ]
[1] 1 1 5
```

À nouveau, cela fonctionne également pour affecter de nouvelles valeurs :

```
> y[ c(TRUE,FALSE) ] <- 2
> y
[1] 2 3 2 3 2 6
```

2.2 L'arithmétique de vecteurs

La plupart des opérations se font composante par composante.

```
> a <- 1:3
> b <- 4:6
> a + b
[1] 5 7 9
> a * b
[1]  4 10 18
> a/b
[1] 0.25 0.40 0.50
```

Recyclage de valeurs

On peut aussi faire agir un scalaire sur toutes les composantes d'un vecteur : c'est un nouveau cas du recyclage déjà rencontré.

```
> a + 1
[1] 2 3 4
> a * 2
[1] 2 4 6
> 1/a
[1] 1.0000000 0.5000000 0.3333333
```

Le recyclage est possible dès que la taille du plus petit vecteur divise celle du plus grand :

```
> a <- 1:8
> b <- c(-1,1)
> a*b
[1] -1  2 -3  4 -5  6 -7  8
> a+b
[1] 0 3 2 5 4 7 6 9
```

Et on a un avertissement dans le cas contraire :

```
> a <- 1:3
> a * 1:2
[1] 1 4 3
Message d'avis :
In a * 1:2 :
  la taille d'un objet plus long n'est pas multiple de la taille
  d'un objet plus court
```

Ainsi qu'on l'a vu dans le cas particulier des vecteurs de longueur 1, si on utilise un vecteur logique dans une opération arithmétique, les `TRUE` sont convertis en 1, les `FALSE` en 0.

```
> b <- c(TRUE,FALSE)
> b
[1]  TRUE FALSE
> b + 2
[1] 3 2
```

2.3 Tests

Si on applique un opérateur de test à un vecteur, on obtient un vecteur booléen :

```
> sample(1:5,20,replace=TRUE) -> Y
> Y
 [1] 2 5 2 3 2 2 1 1 4 3 2 3 2 3 4 5 5 1 2 1
> Y > 2
 [1] FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE  TRUE
[13] FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
> Y == 3
 [1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE
[13] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
```

Les opérateurs de test usuels sont récapitulés dans la table ci-dessous.

Opérateur	sens
>	supérieur
<	inférieur
>=	supérieur ou égal
<=	inférieur ou égal
==	égal
!=	différent

Étant donné que `Y > 2` produit un vecteur de type `logical` avec des `TRUE` aux endroits où les éléments de `Y` sont supérieurs à 2, on peut l'utiliser pour *extraire* les éléments de `Y` qui vérifie cette condition :

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

Il y a également des opérateurs logiques :

Opérateur	sens
!	non
|	ou
&	et
xor()	ou exclusif

Assurez-vous de bien comprendre ce qui suit :

```
> !TRUE
[1] FALSE
> !FALSE
[1] TRUE
> TRUE | FALSE
[1] TRUE
> TRUE & FALSE
[1] FALSE
> sample(0:4,10,replace=TRUE) -> Y
> Y != 0
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
> !(Y == 0)
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
> (Y > 1) & (Y < 4)
 [1] FALSE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE FALSE
> (Y == 2) | (Y == 3)
```

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

Exercice : comment extraire de `Y` les éléments pairs ? Les éléments plus grands que 2 et plus petits que 4 ?

Exercice : Égalité entre valeurs numériques Attention aux tests d'égalité entre valeurs numériques ! Comment interprétez-vous le résultat suivant ? Comment remédier à ce problème ?

```
> (1.1-0.2) == 0.9
[1] FALSE
```

2.4 Fonctions agissant sur les vecteurs booléens

La fonction `any(x)` teste si au moins un des éléments de `x` est `TRUE`, alors que `all(x)` teste si tous les éléments de `x` sont `TRUE` ?

```
> x <- c(TRUE, FALSE)
> any(x)
[1] TRUE
> all(x)
[1] FALSE
```

Ainsi, pour tester l'égalité de deux vecteurs, on pourra procéder comme suit :

```
> (1:4)**2 == (1:4)*(1:4)
[1] TRUE TRUE TRUE TRUE
> all((1:4)**2 == (1:4)*(1:4))
[1] TRUE
```

La fonction `which(x)` renvoie les indices où `x` vaut `TRUE` :

```
> sample(0:4,10,replace=TRUE) -> Y
> Y
 [1] 2 1 1 4 1 0 2 0 3 3
> which(Y > 2)
[1]  4  9 10
```

La fonction `ifelse(test, yes, no)` renvoie un vecteur de même longueur que le vecteur booléen `test`, dont les valeurs sont prises dans les vecteurs `yes` et `no` selon que les éléments de `test` prennent les valeurs `TRUE` et `FALSE` :

```
> ifelse( c(TRUE,FALSE,TRUE,TRUE), 11:14, 21:24)
[1] 11 22 13 14
```

On peut expliquer le résultat ci-dessus par le tableau suivant : quand dans la colonne `test`, il y a `TRUE`, on pioche dans la colonne `yes`, et quand il y a `FALSE`, on pioche dans la colonne `no` :

test	yes	no
TRUE	**11**	21
FALSE	12	**22**
TRUE	**13**	23
TRUE	**14**	24

2.5 Fonctions et vecteurs numériques

Présentons rapidement quelques fonctions qui agissent sur les vecteurs numériques, notamment la somme et le produit :

```
> y <- 1:8
> sum(y)
[1] 36
> prod(y)
[1] 40320
```

Et à nouveau, ces fonctions arithmétiques, quand elles sont appliquées à des vecteurs de type `logical` provoquent une conversion vers le type `integer`.

```
> sample(0:4,10,replace=TRUE) -> Y
> Y
 [1] 1 0 2 0 0 2 3 4 3 0
> Y > 2
 [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE
> sum(Y > 2)
[1] 3
```

On voit que cette dernière commande a permis de compter les élements de `Y` supérieurs à 2.

2.5.1 Fonctions probabilistes

Voici quelques fonctions « probabilistes », permettant de réaliser des tirages aléatoires. On a déjà vu `sample(x, size)` qui tire au hasard `size` éléments de `x` :

```
> sample(1:100,10)
 [1] 90  6 65 50 30 57 42 11 44 83
```

La fonction `rbinom()` permet de générer des valeurs aléatoires suivant une loi binomiale :

```
> rbinom(n=10,size=100,prob=0.4)
 [1] 40 33 39 38 41 36 34 35 35 37
```

La fonction `rnorm()` permet de générer des valeurs aléatoires suivant une loi normale (centrée réduite par défaut) :

```
> rnorm(n=10)
 [1]  0.2734716  1.1858087  0.5652742 -1.6545016 -2.5016061  1.4642132
 [7]  0.3106972 -0.4226029  0.4067051 -0.2448710
```

Exercice : Lisez l'aide de `rnorm()`. Générer des valeurs suivant une loi normale de moyenne 1 et de variance 4.

La fonction `rpois()` permet de générer des valeurs suivant une loi de Poisson :

```
> rpois(n=10, lambda=1)
 [1] 4 2 0 0 0 0 1 1 2 1
```

3 Autres types de données de R

Nous passons brièvement en revue quelques types de données qui n'ont pas encore été présentés.

3.1 Facteurs

Les facteurs sont à considèrer dans le contexte d'une analyse statistique : par exemple le sexe des individus qui participent à une étude ou le centre de traitement (dans une étude multi-centrique) sont à considérer comme des facteurs discrets qui vont avoir un effet sur la réponse.

3.1.1 Facteurs à niveaux

Les facteurs ressemblent à des vecteurs dont les valeurs ont un certain nombre de *niveaux* possibles. Ces niveaux sont ordonnés.

```
> x <- factor( c("b", "a", "b", "b"), levels = c("a", "b", "c") )
> x
[1] b a b b
Levels: a b c
> levels(x)
[1] "a" "b" "c"
> x[4] <- "c"
> x
[1] b a b c
Levels: a b c


> x[3] <- "d"
Warning message:
In `[<-.factor`(`*tmp*`, 3, value = "d") :
  invalid factor level, NA generated


> x
[1] b    a    <NA> c
Levels: a b c
```

Les niveaux d'un facteur peuvent être obtenus par la fonction `levels()` :

```
> levels(x)
[1] "a" "b" "c"
```

La conversion vers le type `integer` permet de retrouver pour chaque valeur de `x` quelle est la position de son niveau dans `levels(x)` :

```
> as.integer(x)
[1]  2  1 NA  3
```

3.1.2 Réordonner les niveaux

La fonction `relevel()` permet de changer le *niveau de référence*, c'est à dire le premier niveau du facteur.

```
> relevel(x, "b")
[1] b    a    <NA> c
Levels: b a c
```

Il n'y a pas de fonction qui réordonne complètement les niveaux d'un facteur (ou je n'en ai pas trouvé). On peut en bricoler une vite fait sur le gaz (décortiquer le fonctionnement de la fonction suivante est laissé en exercice aux lecteurs les plus alertes) :

```
> relevels <- function(x, levels)
+             Reduce( function(x, level) relevel(x, level), rev(levels), init = x)

> x <- factor( c("a", "c", "b", "b", "d"), levels = c("d", "c", "a", "b") )
> x
[1] a c b b d
Levels: d c a b
> x <- relevels(x, c("a", "b", "c", "d"))
> x
[1] a c b b d
Levels: a b c d
```

3.1.3 Rebaptiser les niveaux

On peut changer le nom des niveaux en faisant une affectation à `levels(x)` :

```
> x
[1] a c b b d
Levels: a b c d
> levels(x)[1] <- "HOP"
> x
[1] HOP c   b   b   d
Levels: HOP b c d
> levels(x) <- c("HOP", "BOUM", "LA", "TSOIN")
> x
[1] HOP   LA    BOUM  BOUM  TSOIN
Levels: HOP BOUM LA TSOIN
```

3.1.4 Oublier des niveaux

La fonction `droplevels` peut être utile quand on extrait une partie d'un facteur.

```
> x
[1] HOP   LA    BOUM  BOUM  TSOIN
Levels: HOP BOUM LA TSOIN
> y <- x[1:3]
> y
[1] HOP  LA   BOUM
Levels: HOP BOUM LA TSOIN
> y <- droplevels(y)
> y
[1] HOP  LA   BOUM
Levels: HOP BOUM LA
```

3.1.5 Caveat

Il arrive qu'en lisant un fichier de données formaté en colonnes, une colonne comptenant des valeurs numériques soit par erreur lue comme un facteur. Attention au résultat de la conversion avec `as.numeric()`! Voyez l'exemple suivant. Comment s'explique ce résultat ?

```
> x <- factor( sample(100:109, 20, replace=TRUE) )
> x
 [1] 100 103 108 103 100 109 109 108 100 104 100 109 105 103 104 104 101 100 106
[20] 107
Levels: 100 101 103 104 105 106 107 108 109
> as.numeric(x)
 [1] 1 3 8 3 1 9 9 8 1 4 1 9 5 3 4 4 2 1 6 7
```

3.2 Listes

Les listes permettent de mélanger des objets de classes différentes. Voici une liste dont les composantes sont nommées.

```
> X <- list( a = 12, b = 1:20, lapin = "bugs bunny", z = TRUE)
> X
$a
[1] 12

$b
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20

$lapin
[1] "bugs bunny"

$z
[1] TRUE
```

3.2.1 Indexation des listes

On peut accéder aux composantes d'une liste ainsi :

```
> X$a
[1] 12
> X$b
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

Ou ainsi :

```
> X[["lapin"]]
[1] "bugs bunny"
> X[[2]]
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

Attention ! Si on ne met qu'un crochet on obtient une liste. Ceci sert à extraire des sous-listes de `X` :

```
> X[1]
$a
[1] 12
> X[1:2]
```

```
$a
[1] 12

$b
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
> X[c("a","lapin")]
$a
[1] 12

$lapin
[1] "bugs bunny"
```

3.2.2 Composantes nommées ou anonymes

Les composantes d'une liste ne sont pas forcément nommées :

```
> Y <- list( 1:5, "un" )
> Y
[[1]]
[1] 1 2 3 4 5

[[2]]
[1] "un"
```

On accède aux noms des composantes avec `names()` :

```
> names(X)
[1] "a"     "b"     "lapin" "z"
> names(Y)
NULL
```

Et on peut les modifier :

```
> names(X)[1] <- "aah"
> X
$aah
[1] 12

$b
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20

$lapin
[1] "bugs bunny"

$z
[1] TRUE
> names(Y) <- c("a", "z")
> Y
$a
[1] 1 2 3 4 5

$z
[1] "un"
```

3.2.3 Liste de longueur prescrite

Pour créer une liste de longueur prescrite on utilise... `vector()`.

```
> X <- vector("list", 4)
> X
[[1]]
NULL

[[2]]
NULL

[[3]]
NULL

[[4]]
NULL
> for(i in 1:4) X[[i]] <- 1:i
> X
[[1]]
[1] 1

[[2]]
[1] 1 2

[[3]]
[1] 1 2 3

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

3.2.4 Remarque finale

Il peut arriver qu'une liste contienne une autre liste :

```
> DATA <- list( patient1 = list( a = runif(5), b = runif(7) ),
+               patient2 = list( a = runif(2), b = NA ),
+               patient3 = list( a = runif(5), b = runif(4) ) )
> DATA
$patient1
$patient1$a
[1] 0.4816502 0.5197211 0.9439088 0.9616066 0.4113572

$patient1$b
[1] 0.8486628 0.2914763 0.3145141 0.1627338 0.5331078 0.5427067 0.2862448


$patient2
$patient2$a
[1] 0.2148851 0.6098648

$patient2$b
[1] NA


$patient3
```

```
$patient3$a
[1] 0.97932207 0.68866006 0.02146414 0.88325778 0.94096106

$patient3$b
[1] 0.1168734 0.2071129 0.3620045 0.5459311
> DATA$patient2$a
[1] 0.2148851 0.6098648
```

Cette façon de structurer des données peut demander un peu de gymnastique cérébrale mais elle est à envisager sérieusement quand on a des données volumineuses et « un peu compliquées ».

3.3 Data frames

En première approximation, les data frames sont des listes dont toutes les composantes sont des vecteurs ou des facteurs de même longueur. Ils sont affichés comme des tableaux.

```
> T <- data.frame( a = 1:10, b = runif(10),
+       sex = factor( rep(c("M", "F"), each = 5), levels = c("F", "M") ) )
> T
    a          b sex
1   1 0.18905123   M
2   2 0.93964603   M
3   3 0.10811901   M
4   4 0.04087236   M
5   5 0.88197067   M
6   6 0.34190555   F
7   7 0.44013160   F
8   8 0.41347807   F
9   9 0.88258754   F
10 10 0.24182143   F
```

Nous manipulerons des data frames au chapitre sur la lecture des fichiers de données.

3.4 Matrices

3.4.1 Création de matrices, opérations simples

On peut créer une matrice comme ceci :

```
> A <- matrix( c(1,2,3,4,5,6), nrow=3)
> A
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

Pour que les éléments donnés servent à remplir la matrice ligne par ligne et non colonne par colonne, on utilise

```
> matrix( c(1,2,3,4,5,6), nrow= 3, byrow=TRUE)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

Il y a une fonction `diag()` pour créer des matrices diagonales :

```
> diag( c(4,6,8) )
     [,1] [,2] [,3]
[1,]    4    0    0
[2,]    0    6    0
[3,]    0    0    8
```

On peut extraire des vecteurs lignes et colonnes ainsi :

```
> A[1,]
[1] 1 4
> A[,2]
[1] 4 5 6
```

Pour que le résultat reste sous forme matricielle, on peut ajouter `drop=FALSE` :

```
> A[1,,drop=FALSE]
     [,1] [,2]
[1,]    1    4
> A[,2,drop=FALSE]
     [,1]
[1,]    4
[2,]    5
[3,]    6
```

La transposée s'obtient avec la fonction `t()` :

```
> A
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> t(A)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

La somme de matrices est facile à obtenir :

```
> B <- matrix( c(-1,0, 0,2, -1,-3), nrow=3)
> B
     [,1] [,2]
[1,]   -1    2
[2,]    0   -1
[3,]    0   -3
> A+B
     [,1] [,2]
[1,]    0    6
[2,]    2    4
[3,]    3    3
```

3.4.2 Produits

Il y a une subtilité pour le produit matriciel, il s'obtient avec l'opérateur spécial `%*%` :

```
> A
```

```
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> t(B)
     [,1] [,2] [,3]
[1,]   -1    0    0
[2,]    2   -1   -3
> A %*% t(B)
     [,1] [,2] [,3]
[1,]    7   -4  -12
[2,]    8   -5  -15
[3,]    9   -6  -18
```

Les vecteurs « simples », créés avec `c()`, seront considérés comme des vecteurs lignes ou colonnes selon les circonstances :

```
> A <- matrix( c(1,2,3, 2,0,1, 3,1,4), nrow=3)
> A
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    2    0    1
[3,]    3    1    4
> u <- c(0,1,2)
> A %*% u
     [,1]
[1,]    8
[2,]    2
[3,]    9
> u %*% A
     [,1] [,2] [,3]
[1,]    8    2    9
```

Du coup la forme quadratique u^tAu se calcule simplement comme ceci :

```
> u %*% A %*% u
     [,1]
[1,]   20
```

3.4.3 Inverse, déterminant, trace

L'inverse d'une matrice s'obtient avec la fonction `solve()` :

```
> C <- matrix( c(2,2,3, 1,2,3, -2,3,4), nrow=3)
> C
     [,1] [,2] [,3]
[1,]    2    1   -2
[2,]    2    2    3
[3,]    3    3    4
> solve(C)
     [,1] [,2] [,3]
[1,]    1   10   -7
[2,]   -1  -14   10
[3,]    0    3   -2
> C %*% solve(C)
```

```
     [,1]          [,2]          [,3]
[1,]    1 -8.881784e-16  0.000000e+00
[2,]    0  1.000000e+00 -8.881784e-16
[3,]    0  1.776357e-15  1.000000e+00
```

Autre exemple :

```
> D <- matrix( c(5,0,1, 4,0,1, 2,-2,1), nrow=3)
> D
     [,1] [,2] [,3]
[1,]    5    4    2
[2,]    0    0   -2
[3,]    1    1    1
> solve(D)
     [,1] [,2] [,3]
[1,]    1 -1.0   -4
[2,]   -1  1.5    5
[3,]    0 -0.5    0
```

et le déterminant avec `det()` :

```
> det(C)
[1] -1
> det(D)
[1] 2
```

La trace peut s'obtenir ainsi :

```
> sum(diag(C))
[1] 8
```

3.4.4 Valeurs propres

Les vecteurs propres et les valeurs propres d'une matrice s'obtiennent avec `eigen` (allemand pour « propre » ; en anglais, les vecteurs et valeurs propres sont appelés *eigenvectors* et *eigenvalues*).

```
> A <- matrix( c(1,6,-1, 2,-1,-2, 1,0,-1), nrow=3)
> x <- eigen(A)
> x
eigen() decomposition
$values
[1] -4.000000e+00  3.000000e+00  9.616736e-17

$vectors
           [,1]       [,2]       [,3]
[1,]  0.4082483 -0.4850713 -0.0696733
[2,] -0.8164966 -0.7276069 -0.4180398
[3,] -0.4082483  0.4850713  0.9057529
```

Les valeurs propres sont dans `x$values`, et les vecteurs propres dans les colonnes de `x$vectors`.

```
> u <- x$vectors[,1]
> u
[1]  0.4082483 -0.8164966 -0.4082483
> A %*% u
```

```
          [,1]
[1,] -1.632993
[2,]  3.265986
[3,]  1.632993
> x$values[1] * u
[1] -1.632993  3.265986  1.632993
```

3.5 Tableaux

Les matrices sont des tableaux à deux dimensions, mais on peut faire des tableaux à un nombre quelconque de dimensions. Ici, un tableau à 3 dimensions, la première indexée de 1 à 3, les deux dernières de 1 à 2.

```
> A <- array( 1:12, dim=c(3,2,2) )
> A
, , 1

     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

, , 2

     [,1] [,2]
[1,]    7   10
[2,]    8   11
[3,]    9   12
> A[3,2,1]
[1] 6
> A[,1,1]
[1] 1 2 3
> A[,2,]
     [,1] [,2]
[1,]    4   10
[2,]    5   11
[3,]    6   12
```

4 Lire un fichier de données

Nous allons voir dans ce chapitre comment charger un fichier de données structuré en colonnes, et réaliser quelques manipulations dessus.

4.1 Lire le fichier

Nous disposons d'un fichier `data.txt` qui se présente comme ceci (disponible sur http://www.g2s.u-psud.fr/data/).

```
Ce fichier contient les prénoms, sexes, poids et tailles de 20 individus adultes

prénom        sexe     taille     poids
Marie         F        159        53,7
Jacques       M        175        71,3
Yves          M        182        37,8
Marc          M        175        .
Emmanuelle    F        158        49,6
Hervé         M        181        41,8
Claude        F        155        54,5
Michel        M        172        51,2
René          M        175        68,4
Jacques       M        180        79,1
Éric          M        162        58,1
Nathalie      F        164        44,6
Sophie        F        172        64,8
Olivier       M        172        45,9
Françoise     F        164        47,2
André         M        185        .
Hélène        F        163        50,6
Martin        M        171        71,2
Jennyfer      F        155        38,1
Joëlle        F        157        60,9
```

Le point présent pour certains poids (Marc et André) est à interpréter comme une valeur manquante.
Pour lire ce fichier, nous procédons comme suit :

```
> X <- read.table("data.txt", skip = 2, header = TRUE, na.strings=".", dec=",")
> head(X)
      prénom sexe taille poids
1      Marie    F    159  53.7
2    Jacques    M    175  71.3
3       Yves    M    182  37.8
4       Marc    M    175    NA
5 Emmanuelle    F    158  49.6
6      Hervé    M    181  41.8
```

Les options données ont la signification suivante :

— `skip = 2` : sauter deux lignes avant de commencer à lire
— `header = TRUE` : la première ligne lue est une en-tête (les noms des colonnes)
— `na.strings = "."` : la chaîne de caractères `"."` est à interpréter comme valeur manquante

— `dec = ","` : le séparateur décimal est une virgule

La fonction `read.table()` est très souple ; il existe de nombreuses autres options qui permettent de modifier son comportement. Il est donc nécessaire de lire la page d'aide... Si le fichier est fourni par un utilisateur d'un système d'exploitation de type unix, le texte est certainement encodé avec la norme UTF-8. Sous Windows, qui n'est toujours pas, aux dernières nouvelles, passé à l'UTF-8, il sera nécessaire d'ajouter l'option `encoding = "UTF-8"'`.

Le résultat est un « data frame » : c'est une structure de données dont les colonnes sont des vecteurs (ou des facteurs). Ces vecteurs peuvent avoir des types différents. Examinons-les les uns après les autres dans notre exemple :

```
> X$prénom
 [1] Marie      Jacques    Yves       Marc       Emmanuelle Hervé      Claude
 [8] Michel     René       Jacques    Éric       Nathalie   Sophie     Olivier
[15] Françoise  André      Hélène     Martin     Jennyfer   Joëlle
19 Levels: André Claude Emmanuelle Éric Françoise Hélène Hervé Jacques ... Yves
> X$sexe
 [1] F M M M F M F M M M M F F M F M F M F F
Levels: F M
> X[,"sexe"]
 [1] F M M M F M F M M M M F F M F M F M F F
Levels: F M
> X[,2]
 [1] F M M M F M F M M M M F F M F M F M F F
Levels: F M
> X$taille
 [1] 159 175 182 175 158 181 155 172 175 180 162 164 172 172 164 185 163 171 155 157
> X$poids
 [1] 53.7 71.3 37.8   NA 49.6 41.8 54.5 51.2 68.4 79.1 58.1 44.6 64.8 45.9 47.2   NA
[17] 50.6 71.2 38.1 60.9
```

Notez en passant les trois façons d'accéder à une colonne de `X` : `X$sexe`, ou `X[,"sexe"]`, ou `X[,2]`.

Il y a quelque chose qui ne va pas dans les résultats ci-dessus : s'il est pertinent que la colonne `sexe` soit interprétée comme un facteur à deux modalités `M` et `F`, ça n'est pas souhaitable pour la colonne prénom. Une des solutions est d'utiliser une option supplémentaire qui permet de spécifier la classe de chaque colonne.

```
> X <- read.table("data.txt", skip = 2, header = TRUE, na.strings=".", dec=",",
      colClasses=c("character","factor","numeric","numeric"))
> head(X)
      prénom sexe taille poids
1      Marie    F    159  53.7
2    Jacques    M    175  71.3
3       Yves    M    182  37.8
4       Marc    M    175    NA
5 Emmanuelle    F    158  49.6
6      Hervé    M    181  41.8
> X$prénom
 [1] "Marie"      "Jacques"    "Yves"       "Marc"       "Emmanuelle" "Hervé"
 [7] "Claude"     "Michel"     "René"       "Jacques"    "Éric"       "Nathalie"
[13] "Sophie"     "Olivier"    "Françoise"  "André"      "Hélène"     "Martin"
[19] "Jennyfer"   "Joëlle"
```

On voit dans la colonne poids que les valeurs manquantes sont notées `NA` en R (comme *Non Available*). Il est indispensable en statistiques de les prendre très au sérieux! La fonction `is.na()` fait le test que son nom laisse deviner.

```
> is.na(X$poids)
 [1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[14] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
> X$poids[ !is.na(X$poids) ]
 [1] 53.7 71.3 37.8 49.6 41.8 54.5 51.2 68.4 79.1 58.1 44.6 64.8 45.9 47.2 50.6 71.2
[17] 38.1 60.9
```

4.2 Calcul de moyennes, variance, écart-type, représentations graphiques...

On peut obtenir une information sur chacune des colonnes avec `summary()` :

```
> summary(X)
    prénom          sexe       taille          poids
 Length:20          F: 9   Min.   :155.0   Min.   :37.80
 Class :character   M:11   1st Qu.:161.2   1st Qu.:46.23
 Mode  :character          Median :171.5   Median :52.45
                           Mean   :168.8   Mean   :54.93
                           3rd Qu.:175.0   3rd Qu.:63.83
                           Max.   :185.0   Max.   :79.10
                                           NA's   :2
```

Notons que cette fonction ne précise pas si les variables numériques sont de type `integer` ou `double`.

Calculons le poids moyen de l'ensemble des individus, puis le poids moyen des hommes et celui des femmes :

```
> mean( X$poids )
[1] NA
> mean( X$poids, na.rm = TRUE )
[1] 54.93333
> mean( X$poids[X$sexe == "M"], na.rm = TRUE )
[1] 58.31111
> mean( X$poids[X$sexe == "F"] )
[1] 51.55556
```

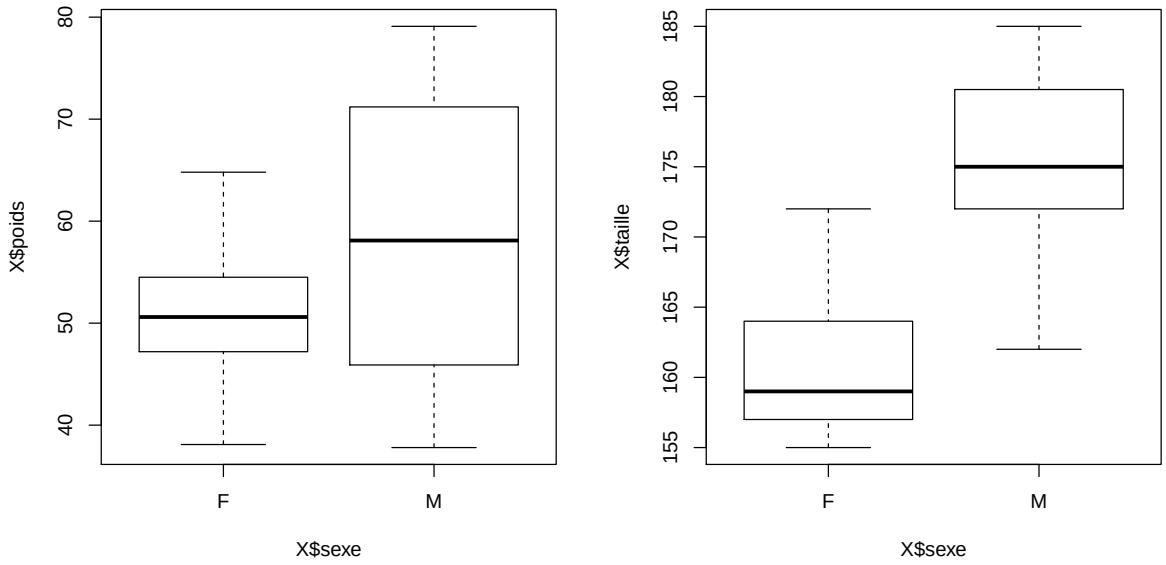
On ne peut pas calculer la moyenne des poids sans spécifier qu'il faut omettre les données manquantes : c'est l'option `na.rm = TRUE`. Elle est utilisée par de nombreuses fonctions en R.

On peut également calculer variance et écart-type :

```
> var( X$poids[X$sexe == "F"] )
[1] 66.14278
> sd( X$poids[X$sexe == "F"] )
[1] 8.132821
```

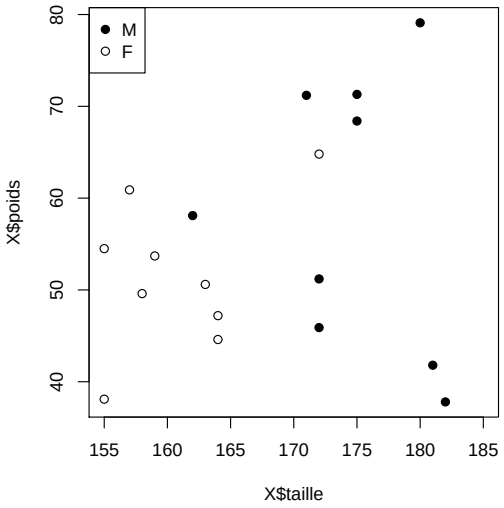
On peut faire un « bar plot » pour visualiser les différences de poids et de taille entre les sexes :

```
> par(mfrow=c(1,2))
> plot( X$poids ~ X$sexe)
> plot( X$taille ~ X$sexe)
```



Et on peut tracer également le poids en fonction de la taille, avec un type de point différent pour chaque sexe :

```
> plot( X$taille, X$poids, pch=ifelse(X$sexe == "M", 19, 1) )
> legend("topleft", c("M","F"), pch=c(19,1))
```



On peut ajouter dans le tableau une colonne pour l'IMC :

```
> X$imc <- X$poids/(X$taille/100)**2
> head(X)
      prénom sexe taille poids      imc
1      Marie    F    159  53.7 21.24125
2    Jacques    M    175  71.3 23.28163
3       Yves    M    182  37.8 11.41167
4       Marc    M    175    NA       NA
5 Emmanuelle    F    158  49.6 19.86861
6      Hervé    M    181  41.8 12.75907
> summary(X$imc)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  11.41   16.76   20.55   19.61   22.60   24.71       2
```

4.3 Extraction de parties du tableau, maximum, minimum, tri

Quel est le plus petit élément du vecteur `X$poids` ?

```
> min(X$poids)
[1] NA
```

On peut également trier un vecteur :

```
> sort(X$poids)
 [1] 37.8 38.1 41.8 44.6 45.9 47.2 49.6 50.6 51.2 53.7 54.5 58.1 60.9 64.8 68.4 71.2
[17] 71.3 79.1
> sort(X$poids, decreasing=TRUE)
 [1] 79.1 71.3 71.2 68.4 64.8 60.9 58.1 54.5 53.7 51.2 50.6 49.6 47.2 45.9 44.6 41.8
[17] 38.1 37.8
> sort(X$prénom)
 [1] "André"      "Claude"     "Emmanuelle" "Éric"       "Françoise"  "Hélène"
 [7] "Hervé"      "Jacques"    "Jacques"    "Jennyfer"   "Joëlle"     "Marc"
[13] "Marie"      "Martin"     "Michel"     "Nathalie"   "Olivier"    "René"
[19] "Sophie"     "Yves"
```

Les fonctions `which.min()` et `which.max()` renvoient l'indice du plus petit et du plus grand élément. Ceci permet de déterminer l'individu le plus lourd, le plus léger :

```
> which.max(X$poids)
[1] 10
> which.min(X$poids)
[1] 3
> X$prénom[ which.min(X$poids) ]
[1] "Yves"
> X$prénom[ which.max(X$poids) ]
[1] "Jacques"
```

On peut également extraire la ligne de `X` qui correspond à ces individus :

```
> X[ which.min(X$poids), ]
  prénom sexe taille poids      imc
3   Yves    M    182  37.8 11.41167
```

Pour trier les individus par taille croissante, on utisera `order()`, qui renvoie les indices des éléments du vecteur dans l'ordre où il faut les prendre pour que le vecteur soit trié.

```
> X[ order(X$taille), ]
       prénom sexe taille poids      imc
7      Claude    F    155  54.5 22.68470
19   Jennyfer    F    155  38.1 15.85848
20     Joëlle    F    157  60.9 24.70688
5  Emmanuelle    F    158  49.6 19.86861
1       Marie    F    159  53.7 21.24125
11       Éric    M    162  58.1 22.13839
17     Hélène    F    163  50.6 19.04475
12   Nathalie    F    164  44.6 16.58239
```

```
15  Françoise    F    164  47.2 17.54908
18     Martin    M    171  71.2 24.34937
8      Michel    M    172  51.2 17.30665
13     Sophie    F    172  64.8 21.90373
14    Olivier    M    172  45.9 15.51514
2     Jacques    M    175  71.3 23.28163
4        Marc    M    175    NA       NA
9        René    M    175  68.4 22.33469
10    Jacques    M    180  79.1 24.41358
6       Hervé    M    181  41.8 12.75907
3        Yves    M    182  37.8 11.41167
16      André    M    185    NA       NA
```

Enfin, on peut procéder ainsi pour extraire les individus dont l'IMC dépasse 22 :

```
> X[ which(X$imc > 23), ]
    prénom sexe taille poids      imc
2  Jacques    M    175  71.3 23.28163
10 Jacques    M    180  79.1 24.41358
18  Martin    M    171  71.2 24.34937
20  Joëlle    F    157  60.9 24.70688
```

Il est tentant d'utiliser `X[ X$imc > 23, ]`, mais il y a un problème avec la présence d'IMC manquants, qui produisent des `NA` dans le vecteur `X$imc > 23` :

```
> X[ X$imc > 23, ]
      prénom sexe taille poids      imc
2    Jacques    M    175  71.3 23.28163
NA      <NA> <NA>     NA    NA       NA
10   Jacques    M    180  79.1 24.41358
NA.1    <NA> <NA>     NA    NA       NA
18    Martin    M    171  71.2 24.34937
20    Joëlle    F    157  60.9 24.70688
> X$imc > 23
 [1] FALSE  TRUE FALSE    NA FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
[14] FALSE FALSE    NA FALSE  TRUE FALSE  TRUE
```

Une solution possible est la suivante :

```
> X[ !is.na(X$imc) & X$imc > 23, ]
    prénom sexe taille poids      imc
2  Jacques    M    175  71.3 23.28163
10 Jacques    M    180  79.1 24.41358
18  Martin    M    171  71.2 24.34937
20  Joëlle    F    157  60.9 24.70688
```

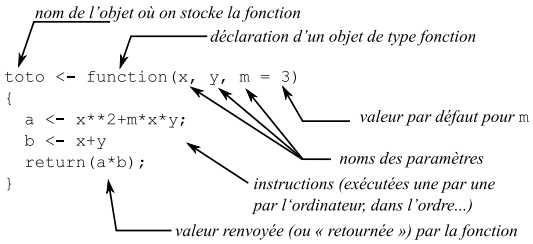
Mais la fonction `subset` est encore ce qu'on fait de plus élégant :

```
> subset(X, imc > 23)
    prénom sexe taille poids      imc
2  Jacques    M    175  71.3 23.28163
10 Jacques    M    180  79.1 24.41358
18  Martin    M    171  71.2 24.34937
20  Joëlle    F    157  60.9 24.70688
```

5 Fonctions

5.1 Anatomie d'une fonction

Voici la déclaration d'une nouvelle fonction, qu'on a choisi d'appeler `toto`.



Que se passe-t-il lors d'un appel à la fonction comme `toto(3,2,1)` ?
Les paramètres `x`, `y`, `m` prennent les valeurs `x=3`, `y=2`, `m=1`. À la première ligne de la fonction, une variable `a` est créée, qui contient le résultat du calcul `x**2+2*m*x`, soit `15`. À la ligne suivante, une variable `b` est créée, qui contient la résultat de `x+y`, soit `5`. Enfin, la fonction renvoie la valeur `a*b`, c'est-à-dire `75`.

```
> toto(3,2,1)
[1] 15  5
[1] 75
```

L'instruction `return(value)` interrompt l'exécution de la fonction, qui renvoie `value` à l'utilisateur.

Dans le *prototype* de la fonction ci-dessus, on voit apparaître une valeur par défaut pour `m`. Ceci permet de faire des appels comme `toto(1,2)` : aucune valeur n'étant fournie pour `m`, on aura `m=3`. Quel sera le résultat ?

Exercice : Quel est le résultat de `toto(0:2,0:2)` ?

(Essayer de répondre sans utiliser R. Ne taper la fonction qu'en désespoir de cause.)

Remarque : R permet d'écrire des fonctions courtes, faisant un seul calcul, sans utiliser les accolades ni la fonction `return()` :

```
toto2 <- function(x, y, m = 3) (x**2+m*x*y)*(x+y);
```

5.2 Fonctions simples

5.2.1 Somme

Écrire une fonction `somme(a,b)` qui renvoie la valeur `a+b`.

5.2.2 Surface d'Elephas Maximus Indicus

Metabolic rate can be expressed as a function of the total surface area during life.
K.P. Sreekumar, G. Nirmala

Le prix IgNobel de mathématiques a été attribué en 2002 à K.P. Sreekumar et G. Nirmalan de la Kerala Agricultural University (India), pour leur rapport intitulé Estimation of the Total Surface Area in Indian Elephants (Elephas Maximus Indicus), *Veterinary Research Communications*, 1990, 14(1) 5-17.

Ces auteurs établissent (entre autres) la formule

$$S = -8.245 + 6.807H + 7.073FFC$$

où H est la hauteur à l'épaule et FFC *(fore footpad circumference)* est la circonférence de la base du pied de devant. H et FFC sont en mètres, S est en mètres carrés. Quelles sont les unités des constantes dans la formule ci-dessus ?

Écrire une fonction `Elephas(H, FFC)` qui calcule la surface d'un éléphant.

5.3 Structures de contrôle

5.3.1 Branchements avec `if`, `else`

Exécuter des instructions dans l'ordre, c'est bien, mais ça ne suffit pas toujours : dans la vie, il faut faire des choix. C'est ce que permet la structure
`if(`*condition*`)`
`{`
instructions
`}`
L'évaluation de la *condition* doit produire `FALSE` ou `TRUE`. Le bloc *instructions* n'est exécuté que si la *condition* est `TRUE`. Voici un exemple :

```
Heaviside <- function(x) {
  if(x<0) {
    return(0);
  }
  return(1);
}
```

Ainsi, le bloc d'instructions entre les accolades qui suivent `if(x<0)` n'est exécuté que si le résultat de `x<0` est `TRUE`. La fonction de Heaviside est

$$\mathrm{H}(x) = \begin{cases} 0 & \text{si } x < 0 \\ 1 & \text{si } x \geq 0. \end{cases}$$

On peut ajouter un `else` (en français *sinon*) pour introduire un bloc d'instructions qui sera exécuté si la condition est fausse. Voici un exemple :

```
F <- function(x,y) {
  if(x<y) {
    z <- x;
  } else {
    z <- y;
  }
  return(z**2+z+1);
}
```

Que fait cette fonction ?

Travailler sur un vecteur : `ifelse`

Nous avons déjà rencontré cette fonction : rappelons que `ifelse(test, yes, no)` renvoie un vecteur de même longueur que le vecteur booléen `test`, dont les valeurs sont prises dans les vecteurs `yes` et `no` selon que les éléments de `test` prennent les valeurs `TRUE` et `FALSE`.

On peut réécrire la fonction d'Heaviside ainsi (notez que si la longueur de `x` est plus grande que 1, les vecteurs `yes = 0` et `no = 1` seront recyclés, comme il est usuel lorsqu'on travaille avec des vecteurs) :

```
Heaviside2 <- function(x) {
  res <- ifelse(x<0, 0, 1);
  return(res);
}
```

Contrairement à la structure `if() else` , cette fonction ne contrôle pas le flux d'exécution ; ses arguments sont des vecteurs et non des blocs d'instructions !!

Exercice : Comparez `plot(Heaviside, xlim=c(-1,1))` et `plot(Heaviside2, xlim=c(-1,1))`. Comparez `Heaviside(-2:2)` et `Heaviside2(-2:2)`. Comprenez-vous ce qui se passe ?

Exercice : Utilisez `sapply()` pour tracer le graphe de la fonction `Heaviside()`.

Exercice : Écrire une fonction qui fait la même chose que la fonction `F` donnée en exemple plus haut, mais en utilisant `ifelse()`.

5.3.2 Boucles `for`

La structure `for` de R est un peu différente de la structure homologue dans les langages classiques (C, Perl, etc). Elle s'utilise ainsi :

`for(`*var* `in` *vecteur*`)`
`{`
Instructions pouvant utiliser var
`}`

Par exemple, la fonction suivante calcule la somme des éléments du vecteur `x`.

```
ma_somme <- function(x) {
  S <- 0;
  for(i in x) {
    S <- S+i;
  }
}
```

...mais il est beaucoup plus rapide d'utiliser `sum(x)` comme le démontre l'essai suivant :

```
> x <- runif(1e6)
> system.time( ma_somme(x) )
   user  system elapsed
  0.041   0.004   0.045
> system.time( sum(x) )
   user  system elapsed
  0.001   0.000   0.001
```

5.3.3 Boucles `while`

La structure
`while(`*expression A*`)`
`{`
instructions B
`}`
permet d'exécuter répétitivement le bloc *instructions B*, et ce *tant que* l'évaluation de *expression A* produit `TRUE`.

Cette fonction renvoie la liste des puissances de 2 plus petites que n.

```
F <- function(n) {
  res <- c();
  i <- 1;
  while(i<n) {
    res<-c(res,i);
    i <- i*2;
  }
  return(res);
}
```

5.3.4 `for` ou `while` ?

Quand on a le choix, la boucle `for` est à préférer ; disons sans plus de détails que c'est de la « bonne pratique de programmation ». La boucle `while` n'est indispensable quand on ne sait pas à l'avance combien de « tours de boucles » vont être réalisés (cf section 5.4.6).

Une question connexe est celle de la construction d'un vecteur dans une boucle, qui sert à retenir des résultats intermédiaires. Deux méthodes sont possibles.

— `x <- numeric(n)` suivie d'une boucle d'affectations `x[i] <- ...`

— `x <- c()` suivie d'une boucle de concaténations `x <- c(x,...)`

Quand c'est possible (et à nouveau, ça l'est quand on connaît à l'avance la longueur du vecteur qui va être créé), la première construction doit être préférée. Comparer avec `system.time()` les performances des fonctions suivantes :

```
a <- function(n) {
  x <- numeric(n);
  for(i in 1:n) {
    x[i] <- i;
  }
  return(x);
}
```

```
b <- function(n) {
  x <- c();
  for(i in 1:n) {
    x <- c(x,i);
  }
  return(x);
}
```

L'explication du mauvais comportement de la deuxième solution est dans la gestion de la mémoire par R. À chaque concaténation, R crée un nouveau vecteur et y recopie l'ancien... et il faut, de temps en temps, qu'il fasse le ménage dans la mémoire pour y supprimer les vieilles copies (opération apellée *garbage collecting*. Voyez la commande `gc`).

5.4 Fonctions un peu moins simples

5.4.1 Une fonction mystère

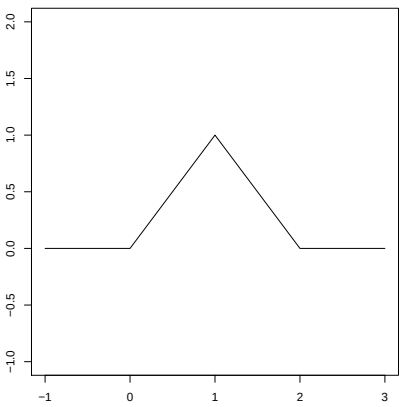
Dessiner le graphe de la fonction suivante. Essayer de répondre sans utiliser R. Ne taper la fonction qu'après avoir sérieusement essayé de comprendre ce qu'elle fait!

```
F <- function(x) {
  if(x<0) {
    return(-x);
  }
  if(x<10) {
    return(0);
  }
  return(x-10);
}
```

Écrire une fonction équivalente en utilisant `ifelse()`.

5.4.2 Fonction tente

Écrire une fonction dont le graphe est :



5.4.3 Décrypter, comparer

L'opérateur arithmétique %% renvoie le reste d'une division entière : par exemple si `i` est entier, `i%%2` vaut 0 ou 1 ; il vaut 1 si `i` est impair. Comparer les trois fonctions suivantes (n'utiliser R que le plus tard possible).

```
f1 <- function(a) {
  resultat <- 0;
  for(i in a) {
    if(i%%2 == 1) {
      resultat <- i;
      return(resultat);
    }
  }
}
```

```
f2 <- function(a) {
  resultat <- 0;
  for(i in a) {
    if(i%%2 == 1) {
      resultat <- i;
    }
    return(resultat);
  }
}
```

```
f3 <- function(a) {
  resultat <- 0;
  for(i in a) {
    if(i%%2 == 1) {
      resultat <- i;
    }
  }
  return(resultat);
}
```

5.4.4 Décrypter, comparer (bis)

Même question avec ces fonctions. Quels sont les résultats de ces fonctions appliquées à `1:6`, `c(1,3,5)`, `c(2,4,6)` ?

```
g1 <- function(a) {
  for(i in a) {
    if(i%%2 == 1) {
      return(FALSE)
    }
  }
  return(TRUE);
}
```

```
g2 <- function(a) {
  for(i in a) {
    if(i%%2 != 1) {
      return(TRUE)
    }
  }
  return(FALSE);
}
```

5.4.5 Une fonction à décortiquer

Que fait cette fonction ? Pouvez-vous remplacer la boucle `while` par une boucle `for` ? Pouvez-vous éviter les concaténations de vecteur à répétition ? Pouvez-vous, enfin, vous passer de boucle ?

```
which.zero <- function(x) {
  res <- c();
  i <- 0;
  while(i < length(x)) {
    i <- i+1;
    if(x[i] == 0) {
      res <- c(res,i);
    }
  }
  return(res);
}
```

5.4.6 La conjecture de Collatz

Ce qui suit est connu sous les noms de « problème $3n+1$ », ou « problème de Syracuse », et encore : « conjecture de Collatz ». On construit une suite d'entiers $u_0, u_1, u_2, \ldots$ à partir d'un entier u_0 arbitraire comme suit : à chaque étape,

– si $u_n = 1$, on arrête ;
– si u_n est pair, l'élément suivant est $u_{n+1} = \frac{1}{2} u_n$;
– si u_n est impair, l'élément suivant est $u_{n+1} = 3u_n + 1$;
– on recommence...

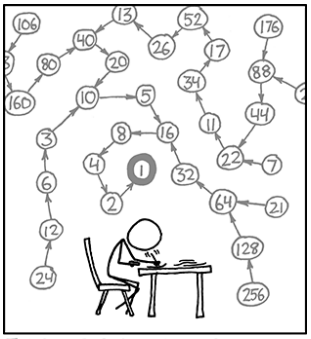
Par exemple, si on part de 7 qui est impair, l'élément suivant est $3 \times 7 + 1 = 22$; 22 étant pair, on a ensuite 11, puis 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1.

Remarque : si on s'arrête en rencontrant 1, c'est parce qu'ensuite on a un motif répété : 1, 4, 2, 1, etc.

Écrire un programme qui calcule la suite partant de u_0 ; est-il certain que ça s'arrête ? Faire des essais.

Peut-on suivre les recommandations de la section 5.3.4 ?

Essayer des variantes (par exemple $5n+1$, etc).



THE COLLATZ CONJECTURE STATES THAT IF YOU PICK A NUMBER, AND IF IT'S EVEN DIVIDE IT BY TWO AND IF IT'S ODD MULTIPLY IT BY THREE AND ADD ONE, AND YOU REPEAT THIS PROCEDURE LONG ENOUGH, EVENTUALLY YOUR FRIENDS WILL STOP CALLING TO SEE IF YOU WANT TO HANG OUT.

http://xkcd.com/710

5.5 Les fonctions sont des objets comme les autres

Les fonctions en R sont des objets, au même titre que les nombres ou les vecteurs. Ceci permet en particulier de passer le nom d'une fonction comme paramètre.

5.5.1 Passer le nom d'une fonction comme paramètre

La fonction `sapply()` prend deux paramètres : `X` et `FUN` (dans cet ordre). En général, `X` est un vecteur, et `FUN` est une fonction à une variable ; le résultat est un vecteur de même longueur que `X`, obtenu en appliquant `FUN` à tous les éléments de `X` un par un.

Bien sûr certaines fonctions peuvent prendre un vecteur comme paramètre, rendant inutile l'utilisation de `sapply()`. Ainsi, par exemple, `sapply(X = -5:5, FUN = sin)` doit produire le même résultat que `sin(-5:5)`.

Par contre, d'autres fonctions en sont incapables. Prenons la fonction définie par l'utilisateur.

```
F <- function(x, alpha=1) {
  if(x<0) {
    return(exp(alpha*x));
  }
  return(1+alpha*sin(x));
}
```

Comparer `F(-1:1)` et `sapply(-1:1, F)`.

Est-ce que l'utilisation de `plot(F, xlim=c(-5,5))` produit un résultat satisfaisant ? Comparer à la séquence de commandes suivante.

```
> x <- seq(-10,10, by = 0.05)
> y <- sapply(a,F)
> plot(x,y,type="l")
```

Pour tracer le graphe de `F` pour `\alpha = 2`, il suffit de passer à `sapply` les paramètres supplémentaires :

```
> y2 <- sapply(a,F, alpha = 2)
> lines(x,y2, col="red")
```

5.5.2 Utiliser cette possibilité dans ses fonctions

Il est très facile d'utiliser cette possibilité dans des fonctions définies par l'utilisateur. La fonction suivante renvoie la valeur que son paramètre `FUN` prend en 0 :

```
valeur_en_0 <- function(FUN) {
  return(FUN(0));
}
```

Le résultat est le suivant :

```
> valeur_en_0(FUN=sin)
[1] 0
> valeur_en_0(FUN=cos)
[1] 1
```

On peut raisonner comme ceci : R remplace `FUN` par ce qu'on lui a passé ; dans le premier cas, `FUN` vaut `sin` et donc `FUN(0)` est interprété comme `sin(0)`.
Remarque : Par contre, un appel comme `valeur_en_0(x+sin(x))` n'a aucun sens : `FUN(0)` est interprété comme `x+sin(x)(0)`, qui ne veut rien dire du tout.

5.5.3 Les paramètres supplémentaires

Pour créer une fonction qui prend des paramètres supplémentaires, on utilise l'opérateur `...`. L'exemple suivant, une nouvelle version de `valeur_en_0`, devrait être parlant :

```
valeur_en_0 <- function(FUN, ...) {
  return(FUN(0, ...));
}
```

Ainsi, si on définit la fonction `H` suivante :

```
H <- function(x, alpha = 1) x**2 - alpha
```

On obtient les résultats suivants :

```
> valeur_en_0(H)
[1] -1
> valeur_en_0(H, alpha = 5)
[1] -5
```

Lors du premier appel, on ne fournit pas de paramètres supplémentaires ; `FUN(0, ...)` est interprété comme `H(0)` et `alpha` prend sa valeur par défaut.
Lors du second appel, on a fournit `alpha = 5` ; c'est cette valeur que prend `...` ; et `FUN(0, ...)` est interprété comme `H(0,alpha = 5)`.

5.5.4 Encore plus fort... !

La responsabilité de l'auteur de ce cours ne saurait être engagée en cas de migraine.
Une fonction étant un objet, on peut renvoyer une fonction avec `return()`. Une fonction peut donc renvoyer comme résultat une autre fonction. Par exemple :

```
Z <- function(alpha) {
  A <- function(x) x**2 - alpha
  return(A);
}
```

À l'intérieur de la fonction `Z`, on crée une fonction qu'on renvoie à l'utilisateur :

```
> Z(5) -> essai
> essai(1)
[1] -4
> essai(2)
[1] -1
> essai(3)
[1] 4
```

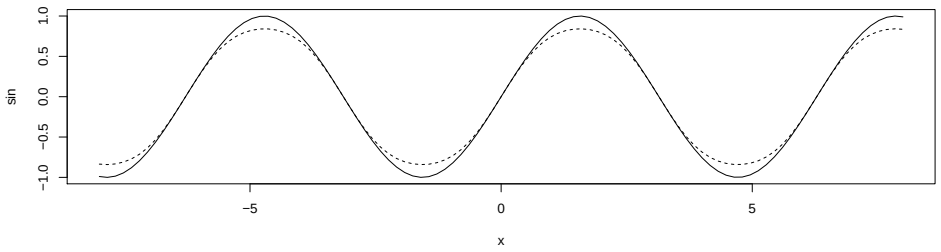
On peut utiliser `Z` pour créer d'autres fonctions, sans altérer le comportement de la fonction `essai` créée ci-desus :

```
> Z(0) -> une_autre
> une_autre(1)
[1] 1
> essai(1)
[1] -4
```

Et bien sûr, on peut prendre en entrée une fonction et en renvoyer une autre :

```
Itere <- function(F) {
  G <- function(x) F(F(x))
  return(G)
}
```

```
> sinsin <- Itere(sin)
> plot(sin, xlim=c(-8,8))
> t <- seq(-8,8,length=101)
> lines(t, sinsin(t), lty=2);
```



Exercice : Que fait la fonction suivante ?

```
Shift <- function(F, shift = 1) {
  G <- function(x) F(x + shift)
  return(G)
}
```

6 Programmation vectorisée

Après avoir appris à faire des boucles, nous allons apprendre à ne pas en faire. Cela sera la plupart du temps beaucoup plus efficace – et, une fois que le coup est pris, cela sera également plus simple.

6.1 Fonctions d'un vecteur ; `sapply()` et `vapply()`

6.1.1 Appliquer une fonction à un vecteur

Nous avons remarqué que des fonctions comme celle qui suit se comportent mal quand on leur fournit un vecteur comme paramètre :

```
F <- function(x) {
  if(x > 0) {
    return(1);
  }
  return(0);
}
```

Pour mémoire, la meilleure solution est de créer une fonction `F_vec` qui sait traiter un vecteur :

```
F_vec <- function(x) ifelse(x>0,1,0)
```

Cependant, il n'est pas toujours aussi simple de créer une telle fonction. Mais on peut toujours faire une boucle, comme ceci :

```
F_for <- function(x) {
  n <- length(x)
  res <- numeric(n)
  for(i in 1:n) res[i] <- F(x[i])
  return(res)
}
```

La fonction `sapply()` permet de faire (grosso modo) la même chose, sans avoir à écrire la boucle. Elle prend en entrée deux arguments : un vecteur `X` et une fonction `FUN`, et renvoie le vecteur qui contient `FUN(X[1])`, `FUN(X[2])`, etc (jusqu'à épuisement des valeurs de `X`).

```
> F_for(-5:5)
 [1] 0 0 0 0 0 0 1 1 1 1 1
> F_vec(-5:5)
 [1] 0 0 0 0 0 0 1 1 1 1 1
> sapply(-5:5, F)
 [1] 0 0 0 0 0 0 1 1 1 1 1
```

6.1.2 Vectorisation des fonctions qui renvoient un vecteur

Considérons la fonction `G` suivante :

```
G <- function(x) c(x, x**2)
```

Elle peut traiter des vecteurs, mais pas forcément de la façon dont on le souhaiterait. `sapply()` permet d'obtenir une matrice :

```
> G(-5:5)
 [1] -5 -4 -3 -2 -1  0  1  2  3  4  5 25 16  9  4  1  0  1  4  9 16 25
> sapply(-5:5, G)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
[1,]   -5   -4   -3   -2   -1    0    1    2    3     4     5
[2,]   25   16    9    4    1    0    1    4    9    16    25
```

Note : la fonction `rbind()` permet de créer une fonction vectorisée qui se comporte comme `sapply(x, G)`.

6.1.3 Arguments optionnels

Considérons la fonction `F` suivante :

```
F <- function(x, a = 1) {
  if(x > 0) {
    return(a);
  }
  return(0);
}
```

Supposons qu'on veuille calculer `F(x, 2)` pour un vecteur `x`. Une solution est de définir une fonction auxiliaire, qu'on peut passer à `sapply()` sans même avoir à lui donner un nom :

```
> sapply(-5:5, function(x) F(x,2))
 [1] 0 0 0 0 0 0 2 2 2 2 2
```

Mais on peut, et c'est plus simple, passer des arguments optionnels à `sapply()` ;

```
> sapply(-5:5, F, a = 2)
 [1] 0 0 0 0 0 0 2 2 2 2 2
```

6.1.4 Plus efficace : `vapply()`

Quand on travaille dans la console R sur des exemples de petite taille, `sapply()` fait bien son travail. Quand on réalise un script qui va traiter un gros volume de données, on a intérêt à utiliser `vapply()` qui va aller plus vite. Le prix à payer est qu'il faut à `vapply()` un paramètre supplémentaire `FUN.VALUE` qui dit à la fonction de quelle classe et de quelle longueur est le résultat :

```
> vapply(-5:5, F, numeric(1))
 [1] 0 0 0 0 0 0 1 1 1 1 1
```

Ici `numeric(1)` précise que `F` produit un vecteur de type `double` de longueur 1. Pour utiliser la fonction `G` vue plus haut (qui renvoie `c(x,x**2)`), il faudrait donner `numeric(2)` :

```
> vapply(-5:5, G,  numeric(2))
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
[1,]   -5   -4   -3   -2   -1    0    1    2    3     4     5
[2,]   25   16    9    4    1    0    1    4    9    16    25
```

Comparons la rapidité d'exécution des différentes solutions :

```
> x <- runif(1e5, -1,1)
> system.time( F_for(x) )
   user  system elapsed
  0.124   0.000   0.121
> system.time( sapply(x, F) )
```

```
   user  system elapsed
  0.068   0.000   0.071
> system.time( vapply(x, F, numeric(1) ) )
   user  system elapsed
  0.064   0.004   0.066
> system.time( F_vec(x) )
   user  system elapsed
  0.028   0.000   0.029
```

La fonction vectorielle `F_vec` est bien sûr la plus rapide. Les performances de `vapply()` sont suffisament supérieures à celles de `sapply()` pour qu'on prenne la peine de fournir le paramètre `FUN.VALUE`.

6.2 La fonction `lapply()`

La fonction `lapply()` ne simplifie pas son résultat, ce qui permet d'agréger des résultats de taille variable. Ici on applique la fonction « carré » à une liste.

```
> X <- list( 1:5, 1:4 )
> X2 <- lapply(X, function(x) x**2)
> X2
[[1]]
[1]  1  4  9 16 25

[[2]]
[1]  1  4  9 16
```

La fonction `sapply()` utilise en fait `lapply()`.

6.3 Répéter une commande : la fonction `replicate()`

Une autre utilisation possible des boucles est le remplissage d'un vecteur ou d'une matrice par répétition d'une commande qui ne dépend pas de l'indice de la boucle, par exemple comme ceci :

```
> set.seed(123)
> X <- matrix(nrow=3, ncol = 5)
> for(i in 1:5) { X[,i] <- rpois(3, lambda=4) }
> X
     [,1] [,2] [,3] [,4] [,5]
[1,]    3    6    4    4    5
[2,]    6    7    7    8    4
[3,]    3    1    4    4    2
```

La fonction `replicate()` fait la même chose :

```
> set.seed(123)
> replicate(5, { rpois(3, lambda=4) } )
     [,1] [,2] [,3] [,4] [,5]
[1,]    3    6    4    4    5
[2,]    6    7    7    8    4
[3,]    3    1    4    4    2
```

Autre exemple : on peut simuler des tirages dans une loi $\chi^2(3)$ comme ceci, en tirant profit de la définition d'un χ^2 comme somme de carrés de loi normale :

```
> simus_chi2 <- function(n, df) {
   res <- numeric(n)
   for(i in 1:n) {
     res[i] <- sum( rnorm(df)**2 )
   }
   return(res)
 }
> x <- simus_chi2(100000, df = 3)
```

Notez que `rnorm()` permet de faire des tirages dans une loi normale standard.
La version qui utilise `replicate()` est nettement plus compacte :

```
> x <- replicate(100000, sum(rnorm(3)**2) )
```

Bien entendu la bonne solution est d'utiliser directement la fonction `rchisq()`...

6.4 Argument multiples : `mapply()`

Quand on a plusieurs arguments à vectoriser, par exemple une fonction comme celle-ci :

```
maximum <- function(x,y) {
  if(x < y) return(y)
  return(x)
}
```

La solution est d'utiliser `mapply()`, ainsi :

```
> x <- c(1,0,8)
> y <- c(9,2,1)
> mapply(maximum, x, y)
[1] 9 2 8
```

6.5 Travailler sur les lignes et colonnes d'une matrice

Considérons la matrice suivante :

```
> set.seed(2)
> X <- matrix(rpois(12, 3), nrow=3)
> X
     [,1] [,2] [,3] [,4]
[1,]    1    1    1    3
[2,]    4    6    5    3
[3,]    3    6    3    2
```

Si on veut calculer la somme des éléments de `X` colonne par colonne, ou ligne par ligne, on dispose des fonctions `colSums()` et `rowSums()` :

```
> colSums(X)
[1]  8 13  9  8
> rowSums(X)
[1]  6 18 14
```

Il existe aussi des fonctions `colMeans` et `rowMeans` pour la moyenne.
Et si on veut l'écart-type de chaque colonne, ou de chaque ligne? La fonction `apply()` est là pour ça :

```
> apply(X, 2, sd)
[1] 1.5275252 2.8867513 2.0000000 0.5773503
```

```
> apply(X, 1, sd)
[1] 1.000000 1.290994 1.732051
```

Le premier argument est la matrice sur laquelle on veut travailler. Le second (nommé MARGIN) vaut 1 pour les lignes, 2 pour les colonnes ; enfin, le troisième (nommé FUN) est la fonction à appliquer.

Reprenons l'exemple des tirages dans un $\chi^2(3)$. En utilisant colSums comme ceci, on ira encore plus vite qu'avec replicate :

```
> x <- colSums(matrix(rnorm(3*100000), nrow=3)**2)
```

Bien sûr le plus efficace c'est d'utiliser la fonction rchisq(), qui est dédiée à cette tâche.

6.6 Application à un vecteur « morceau par morceau » : tapply()

Le cas qui se présente le plus naturellement est celui d'un calcul « stratifié sur un facteur ». Considérons par exemple ce data frame :

```
> set.seed(124)
> T <- data.frame( sexe = sample(c("M", "F"), 10, replace = TRUE),
                   x = rnorm(10, mean = 2, sd = 0.5))
> T
   sexe        x
1     M 2.372240
2     M 2.350115
3     F 1.885323
4     M 2.098547
5     M 2.603577
6     M 2.159168
7     F 1.288101
8     M 1.797455
9     F 2.497693
10    M 2.479409
```

On veut la moyenne de la variable x chez les hommes et chez les femmes. La solution qu'on a utilisé jusqu'à présent est d'extraire de T$x les valeurs qui correspondent à chacun des niveaux du facteur T$sexe, et d'y appliquer la fonction mean(). Avec tapply(), c'est bien plus simple – et, si les données sont volumineuses, bien plus rapide :

```
> tapply(T$x, T$sexe, mean)
       F        M
1.890372 2.265787
```

Pour bien comprendre comment fonctionne tapply() on peut en faire une version qui utilise une boucle for :

```
mon_tapply <- function(x, f, FUN) {
  le <- levels(f)
  n <- length(le)
  res <- numeric(n)
  for( i in 1:n ) {
    res[i] <- FUN( x[f == le[i]] )
  }
  names(res) <- le
  return(res)
}
```

Notez l'usage de names() pour nommer les éléments du vecteurs par les niveaux du facteur. La boucle qui est dans cette fonction ressemble à une boucle qu'on sait remplacer par un sapply() :

```
mon_tapply2 <- function(x, f, FUN)
    sapply(levels(f), function(l) FUN(x[f == l]))
```

7 Introduction à la régression

7.1 Régression linéaire

Nous allons considérer à titre d'exemple un jeu de données inclu dans le package R `alr3`. Pour installer ce package, taper : `install.packages("alr3")`.

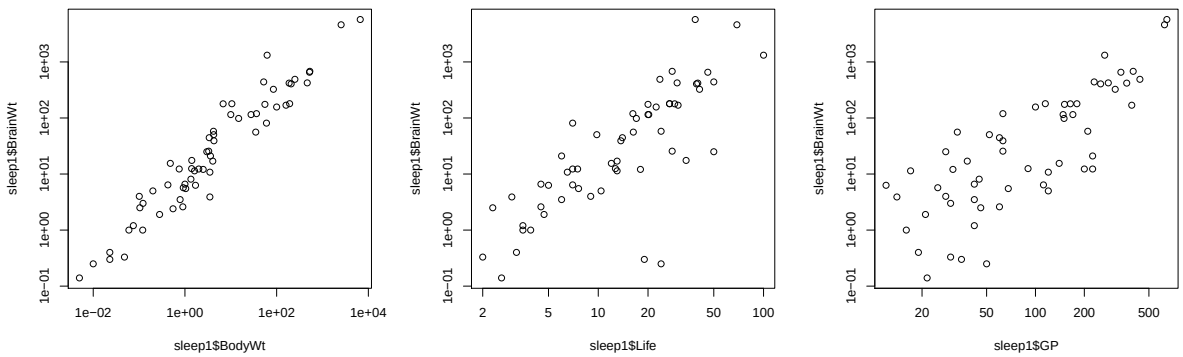
Nous pouvons ensuite le charger avec `library`, puis charger les données avec `data` :

```
> library("alr3")
> data(sleep1)
> head(sleep1, 10)
                          SWS  PS   TS   BodyWt BrainWt Life  GP P SE D
African_elephant           NA  NA  3.3 6654.000  5712.0 38.6 645 3  5 3
African_giant_pouched_rat  6.3 2.0  8.3    1.000     6.6  4.5  42 3  1 3
Arctic_Fox                 NA  NA 12.5    3.385    44.5 14.0  60 1  1 1
Arctic_ground_squirrel     NA  NA 16.5    0.920     5.7   NA  25 5  2 3
Asian_elephant             2.1 1.8  3.9 2547.000  4603.0 69.0 624 3  5 4
Baboon                     9.1 0.7  9.8   10.550   179.5 27.0 180 4  4 4
Big_brown_bat             15.8 3.9 19.7    0.023     0.3 19.0  35 1  1 1
Brazilian_tapir            5.2 1.0  6.2  160.000   169.0 30.4 392 4  5 4
Cat                       10.9 3.6 14.5    3.300    25.6 28.0  63 1  2 1
Chimpanzee                 8.3 1.4  9.7   52.160   440.0 50.0 230 1  1 1
```

Dans ce jeu de données, on trouve des mesures moyennes pour 62 espèces de mammifères. Nous allons nous intéresser à la variable `BrainWt` : le poids du cerveau, à prédire par `BodyWt`, le poids du corps, `Life`, la durée de vie maximale (en années), `GP` le temps de gestation (en jours).

Tout d'abord quelques graphes ; l'échelle logarithmique s'avère de rigueur :

```
> par(mfrow=c(1,3), cex=1.2)
> plot( sleep1$BodyWt, sleep1$BrainWt,  log = "xy" )
> plot( sleep1$Life, sleep1$BrainWt, log = "xy"  )
> plot( sleep1$GP, sleep1$BrainWt, log = "xy"  )
```



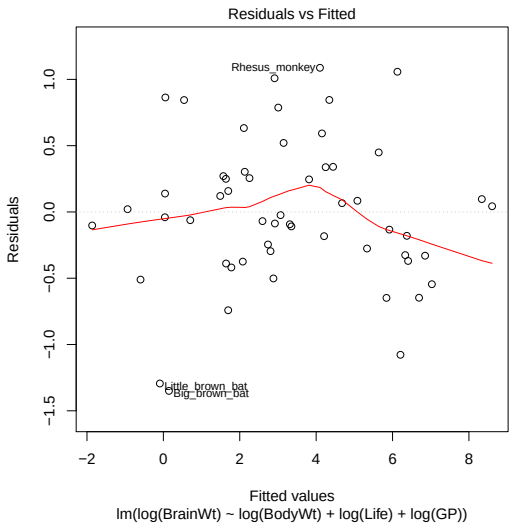
La régression se fait par la commande `lm`.

```
> reg <- lm( log(BrainWt) ~ log(BodyWt) + log(Life) + log(GP), data = sleep1 )
> reg
```

```
Call:
lm(formula = log(BrainWt) ~ log(BodyWt) + log(Life) + log(GP),
    data = sleep1)

Coefficients:
(Intercept)  log(BodyWt)    log(Life)      log(GP)
    -0.3573       0.5674       0.4929       0.3352
> summary(reg)
Call:
lm(formula = log(BrainWt) ~ log(BodyWt) + log(Life) + log(GP),
    data = sleep1)

Residuals:
     Min       1Q   Median       3Q      Max
-1.34961 -0.32775 -0.04003  0.28666  1.08746

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -0.35725    0.46991  -0.760 0.450600
log(BodyWt)  0.56741    0.04008  14.158  < 2e-16 ***
log(Life)    0.49295    0.12387   3.980 0.000219 ***
log(GP)      0.33522    0.10925   3.068 0.003442 **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.5606 on 51 degrees of freedom
  (7 observations deleted due to missingness)
Multiple R-squared:  0.9528,        Adjusted R-squared:   0.95
F-statistic: 343.3 on 3 and 51 DF,  p-value: < 2.2e-16
```

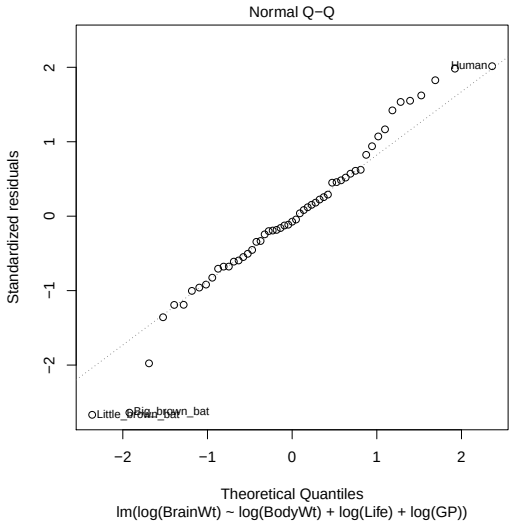
Plusieurs graphes de diagnostic sont utiles. Les résidus en fonctions de la valeur prédite peuvent permettre de détecter certaines formes classiques d'hétéroscédasticité. Par exemple, un nuage de points « en trompette » indiquerait que la variance augmente avec la valeur prédite.

```
> plot(reg, which = 1)
```



La normalité des résidus est intéressante également.

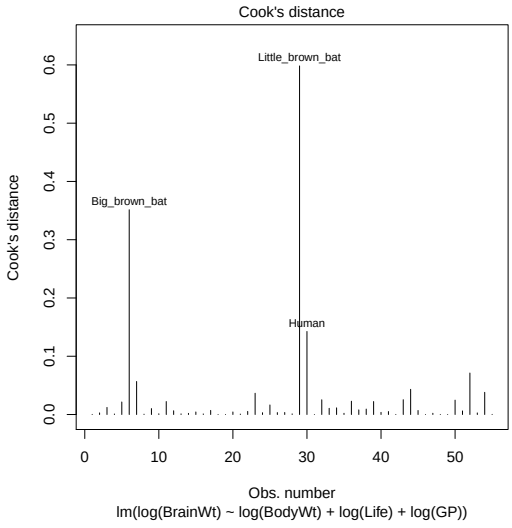
```
> plot(reg, which = 2)
```



On voit ici que l'humain est la plus grande déviation à droite, sans pour autant « sortir du modèle ». Les deux chauves-souris brunes ont par contre un cerveau beaucoup trop petit !

La distance de Cook mesure l'influence d'une observation sur le modèle. Plus elle est élevée, plus cette observation influence les valeurs prédites pour les autres points.

```
> plot(reg, which = 4)
```



7.1.1 Sélection de variables avec AIC et BIC

On peut se demander si le temps de gestation et la durée de la vie améliorent vraiment le modèle, ou si le poids du corps suffit à bien prédire le poids du cerveau (au vu des significations marginales, il y a peu de doutes).

Les critères AIC et BIC, définis respectivement comme $2k - 2\log \mathrm{L}$ et $\log \mathrm{N} \times k - 2\log \mathrm{L}$, où k est le nombre de paramètres, N le nombre d'observations, et L la vraisemblance, sont des critères de sélection de modèles. On choisit le modèle pour lequel AIC ou BIC prennent la valeur la plus petite.

```
> reg1 <- lm(log(BrainWt) ~ log(BodyWt), data = sleep1)
> AIC(reg)
[1] 98.27753
> AIC(reg1)
```

```
[1] 134.6729
> BIC(reg)
[1] 108.3142
> BIC(reg1)
[1] 141.0543
```

7.2 Régression logistique

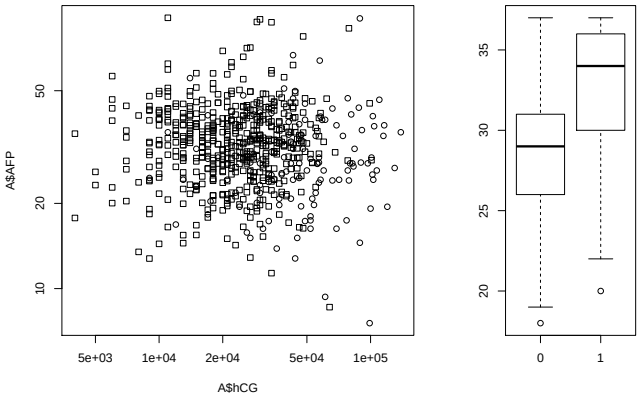
On va cette fois considérer le jeu de données contenu dans le fichier `T21.txt` dont voici les 15 premières lignes (ce fichier est dans http://www.g2s.u-psud.fr/data/).

Il s'agit d'un jeu de données cas-témoins pour la trisomie 21, avec 524 grossesses normales et 125 grossesses avec trisomies (colonne `T21`). Ces données ont été rassemblées à Versailles par Françoise Jauzein et sont disponibles sur `http://acces.ens-lyon.fr/`

Les colonnes `AFP` et `hCG` correspondent à la concentrention en alpha-fœtoprotéine et en hormone gonadotrophone chorionique, exprimées en unités internationales, et mesurées entre la 15^{e} et la 17^{e} semaine de grossesse (colonne `age.gest`). La colonne `age` contient l'âge de la mère.

Commençons par charger les données et les regarder un peu.

```
> A <- read.table("T21.txt", header=TRUE)
> layout(t(1:2), width=c(2,1))
> plot( A$hCG, A$AFP, log = "xy", pch=A$T21)
> boxplot( A$age~A$T21 )
```



On peut faire une régression logistique...

```
> reg <- glm( T21 ~ age + age.gest + log(AFP) + log(hCG), data = A, family = binomial )
> reg
Call:  glm(formula = T21 ~ age + age.gest + log(AFP) + log(hCG), family = binomial,
    data = A)

Coefficients:
(Intercept)          age     age.gest     log(AFP)     log(hCG)
   -38.1844       0.2530       0.4225      -1.8309       2.7527

Degrees of Freedom: 648 Total (i.e. Null);  644 Residual
Null Deviance:            636
Residual Deviance: 380.5         AIC: 390.5
> summary(reg)
```

```
Call:
glm(formula = T21 ~ age + age.gest + log(AFP) + log(hCG), family = binomial,
    data = A)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.9588  -0.4717  -0.2384  -0.0853   3.3996

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept) -38.18436    4.79980  -7.955 1.79e-15 ***
age           0.25296    0.03616   6.996 2.64e-12 ***
age.gest      0.42253    0.19016   2.222   0.0263 *
log(AFP)     -1.83086    0.40819  -4.485 7.28e-06 ***
log(hCG)      2.75265    0.29652   9.283  < 2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 635.99  on 648  degrees of freedom
Residual deviance: 380.45  on 644  degrees of freedom
AIC: 390.45

Number of Fisher Scoring iterations: 6
```

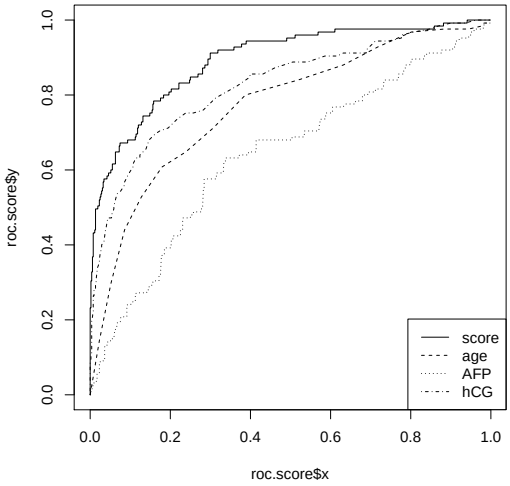
7.2.1 Tracer une courbe ROC

Le pouvoir discriminant d'un score est souvent mesuré par l'aire sous la courbe ROC. Il y a des packages qui font ça mais on peut simplement utiliser une petite fonction comme celle-ci :

```
roc <- function( score, status ) {
  sc <- sort(unique(score), decreasing=TRUE)
  s0 <- score[status == 0]
  s1 <- score[status == 1]
  n0 <- length(s0)
  n1 <- length(s1)
  x <- sapply(sc, function(s) sum(s0 > s))/n0
  y <- sapply(sc, function(s) sum(s1 > s))/n1
  auc <- sum( y[-1]*diff(x))
  list(x = x, y = y, auc = auc)
}
```

Traçons la courbe ROC pour le score produit par la régression linéaire, ainsi que celles qui correspondent aux variables isolées.

```
> roc.score <- roc(score = predict(reg, A), status = A$T21)
> roc.score$auc
[1] 0.8911145
> plot( roc.score, type="l")
> lines( roc(A$age, A$T21), lty = 2)
> lines( roc(-A$AFP, A$T21), lty = 3)
> lines( roc(A$hCG, A$T21), lty = 4)
> legend( "bottomright", lty = 1:4, legend = c("score", "age", "AFP", "hCG"))
```



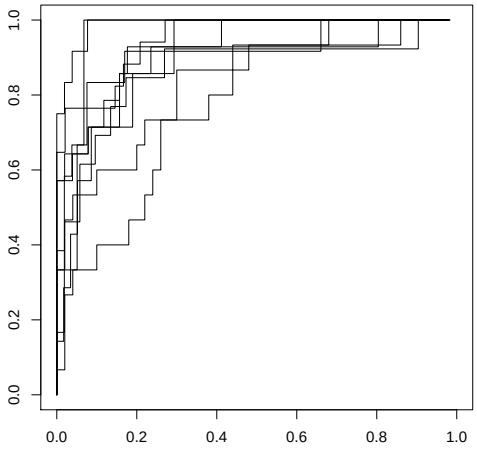
Il resterait à calibrer le score avant de pouvoir l'utiliser. Il faudrait pour cela des données de cohorte et non des données cas-témoins.

7.2.2 Validation croisée

Nous avons sur-estimé le pouvoir discriminant de notre score en l'évaluant sur les données qui ont servi à déterminer les poids accordés aux variables. La validation croisée (K-fold cross-validation) consiste à découper le jeu de données en K parties. On détermine les poids des variables avec $K-1$ des parties regroupées en un ensemble d'apprentissage (training set), et on estime son pouvoir discriminant sur la partie qui reste (test set).

Cette procédure produit K courbes ROC, et K valeurs pour l'aire sous la courbe. On prendra ici simplement leur moyenne.

```
K <- 10
A$k <- sample(rep_len(1:K, nrow(A)), nrow(A))
plot(0,0, type="n", xlab="", ylab="", xlim=c(0,1), ylim=c(0,1))
auc <- numeric(K)
for(k in 1:K) {
  B <- A[ A$k != k, ]
  C <- A[ A$k == k, ]
  reg <- glm( T21 ~ age + as.factor(age.gest) + log(AFP) + log(hCG), data = B, family = binomial )
  score <- predict(reg, C)
  rr <- roc(score, C$T21)
  lines(rr, lwd=0.2)
  auc[k] <- rr$auc
}
```




```
> auc
 [1] 0.9103641 0.8866995 0.8883648 0.9301471 0.8767507 0.8491124 0.7453333
 [8] 0.8253333 0.9435028 0.9695513
> mean(auc)
[1] 0.8825159
```

8 Approche de Monte-Carlo

Les méthodes de Monte-Carlo sont utilisées pour calculer un résultat numérique de façon approchée, au moyen de valeurs aléatoires simulées.

8.1 Premiers exemples

8.1.1 Un coup de dés

Par exemple, parmi les questions qui ont été à l'origine du développement de la théorie des probabilités, on trouve les suivantes (questions du Chevalier de Méré) :

— On lance 4 fois un dé à 6 faces. Quelle est la probabilité de faire au moins une fois un six?

— On lance 24 fois deux dés. Quelle est la probabilité de faire au moins une fois un double-six?

Le chevalier avait obtenu un estimation de la réponse à ces questions en jouant aux dés.

Répondons à la première. On peut simuler B lancers de 4 dés ainsi, à l'aide de la fonction `sample` :

```
> B <- 1e5
> X <- matrix( sample(1:6, 4*B, replace = TRUE), ncol=4)
> head(X)
     [,1] [,2] [,3] [,4]
[1,]    2    5    2    5
[2,]    5    2    6    6
[3,]    4    3    2    5
[4,]    2    3    4    3
[5,]    6    5    4    3
[6,]    6    2    5    3
```

On peut tester l'égalité à six :

```
> head(X == 6)
      [,1]  [,2]  [,3]  [,4]
[1,] FALSE FALSE FALSE FALSE
[2,] FALSE FALSE  TRUE  TRUE
[3,] FALSE FALSE FALSE FALSE
[4,] FALSE FALSE FALSE FALSE
[5,]  TRUE FALSE FALSE FALSE
[6,]  TRUE FALSE FALSE FALSE
```

Pour compter les lignes où il y a un six, on applique à chaque ligne de la matrice `X = 6` la fonction `any()` :

```
> head( apply(X == 6, 1, any) )
[1] FALSE  TRUE FALSE FALSE  TRUE  TRUE
```

Pour finir, on peut estimer la probabilité de gagner par la proportion de parties gagnantes parmi les B parties simulées :

```
> p <- sum( apply(X == 6, 1, any) )/B
```

Et on peut calculer un intervalle de confiance sur p, avec la formule habituelle pour les proportions :

```
> p + c(-1,1)*1.96*sqrt(p*(1-p)/B)
[1] 0.5159832 0.5221768
```

Plus B est grand, plus l'estimation sera précise. En pratique, sur un ordinateur « du modèle courant », prendre $B = 10^5$ ou 10^6 est généralement possible, mais $B = 10^7$ ne l'est plus (temps de calcul, encombrement mémoire...).

Exercice : Répondre à la seconde question posée ci-dessus.

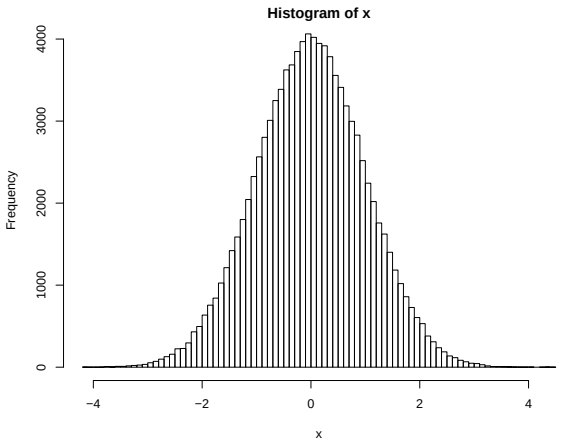
8.1.2 Simulations de variables aléatoires

La table suivante donne la liste des lois pour lesquelles des commandes existent par défaut en R. D'autres lois sont disponibles dans des packages (par exemple la loi normale multivariée est disponible dans le package MASS).

Commande	Loi	Commande	Loi
`rbeta`	Beta	`rlogis`	Logistique
`rbinom`	Binomiale	`rmultinom`	Multinomiale
`rchauchy`	Cauchy	`rnbinom`	Binomiale négative
`rchisq`	Chi-carré	`rnorm`	Normale
`rexp`	Exponentielle	`rpois`	Poisson
`rgamma`	Gamma	`runif`	Uniforme
`rgeom`	Géométrique	`rweibull`	Weibull
`rhyper`	Hypergéométrique	`rwilcox`	Statistique de Wilcoxon
`rnorm`	Normale	`rWishart`	Wishart

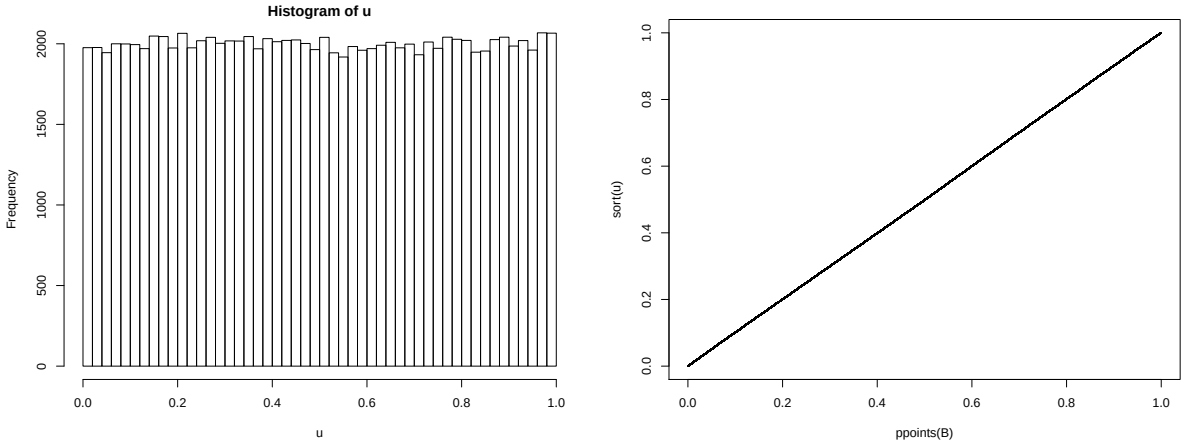
On peut par exemple simuler des valeurs selon une loi normale standard, et en faire l'histogramme :

```
> B <- 1e5
> x <- rnorm(B)
> hist(x, breaks=100)
```



Faisons un histogramme et un diagramme quantile-quantile (qq-plot) pour la loi uniforme :

```
> u <- runif(B)
> par(mfrow=c(1,2))
> hist(u, breaks=50)
> plot(ppoints(B), sort(u), pch=".")
```



De façon générale les diagrammes quantile-quantile permettent une vérification plus fine de l'adéquation à une loi que les histogrammes.

8.1.3 Vérifier qu'un test n'est pas biaisé

La propriété première d'un test est que si l'hypothèse nulle est vraie, la p-valeur suit une loi uniforme.
Reprenons l'exemple du jeu de données `T21` :

```
> A <- read.table("T21.txt", header=TRUE)
> summary(glm( A$T21 ~ A$age, family = binomial ))
Call:
glm(formula = A$T21 ~ A$age, family = binomial)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.2933  -0.5893  -0.4553  -0.2668   2.9777

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) -9.93865    0.98938 -10.045   <2e-16 ***
A$age        0.27587    0.03086   8.941   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 635.99  on 648  degrees of freedom
Residual deviance: 536.19  on 647  degrees of freedom
AIC: 540.19

Number of Fisher Scoring iterations: 5
```

On peut être pris d'un doute : est-ce que ce test est correct? Est-il susceptible de produire un tel résultat alors qu'on est sous H_0 ?

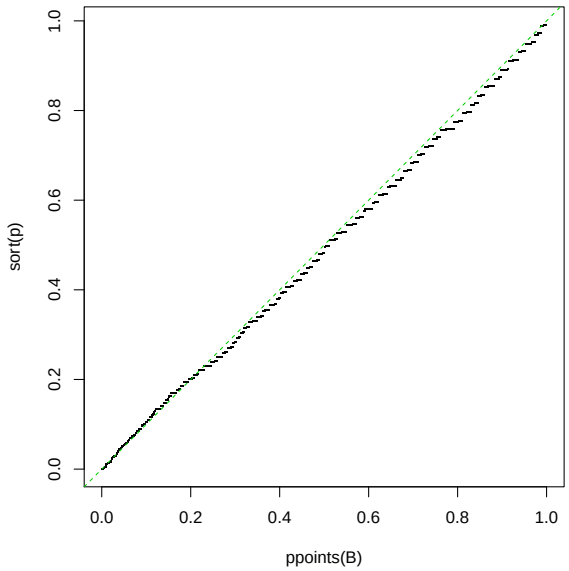
Une façon simple d'obtenir des données sous H_0 est de permuter les valeurs de y :

```
> y <- sample(A$T21);
> summary(glm(y ~ A$age))
Call:
glm(formula = y ~ A$age)

Deviance Residuals:
```

```
    Min       1Q   Median       3Q      Max
-0.2339  -0.2005  -0.1837  -0.1615   0.8497

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 0.027731   0.114988   0.241    0.810
A$age       0.005572   0.003851   1.447    0.148

(Dispersion parameter for gaussian family taken to be 0.1554852)

    Null deviance: 100.92  on 648  degrees of freedom
Residual deviance: 100.60  on 647  degrees of freedom
AIC: 637.86

Number of Fisher Scoring iterations: 2
```

Ceci commence à nous donner confiance. Pour être vraiment certains que le test se comporte de façon correcte, on répétera un grand nombre de fois, et on vérifiera que les p-valeurs sont distribuées de façon uniforme :

```
> B <- 1e3
> p <- replicate( B, { y <- sample(A$T21); x <- summary(glm(y ~ A$age));
                       x$coefficients[2,4] } )
> plot(ppoints(B), sort(p), pch=".")
> abline(0,1,lty=2,col=3)
```



8.1.4 Puissance d'un test

On considère ici l'exemple de la régression logistique $\text{logit}\,\mathbb{P}(Y=1) = \beta_0 + \beta_1 X$, où X prend des entières 0, 1 ou 2 avec les probabités p^2, $2pq$, q^2.

On va s'intéresser à sa puissance qui dépend de la taille d'échantillon n, de la valeur de p et de celle de β.

La fonction suivante réalise B fois la régression logistique et renvoie une matrice contenant les estimations $\hat{\beta}_1$ et $\hat{\sigma}$ de β_1 et de l'écart-type de cette estimation, le z-score $\hat{\beta}_1/\hat{\sigma}$ et la p-valeur correspondant.

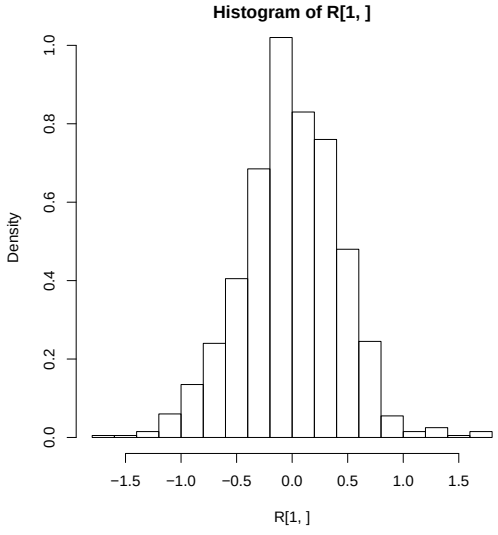
```
regression <- function(n, beta1, p = 0.5, beta0 = 0.4, B = 1000) {
  f <- function() {
```

```
    X <- sample(0:2, n, replace = TRUE, prob = c(p**2, 2*p*(1-p), (1-p)**2))
    S <- beta0 + beta1*X
    pr <- 1/(1 + exp(-S))
    Y <- runif(n) < pr;
    a <- glm(Y ~ X, family=binomial())
    b <- summary(a)
    b$coefficients[2,]
  }
  replicate(B, f() )
}
```

Remarque : On va garder $\beta_0 = 0{,}4$. Pour bien faire, et si on s'intéresse au cas d'une étude cas/témoins, il faudrait se poser la question de la simulation d'un panel cas/témoin dans un ratio fixé... On fera à titre d'exercice varier p en TD.

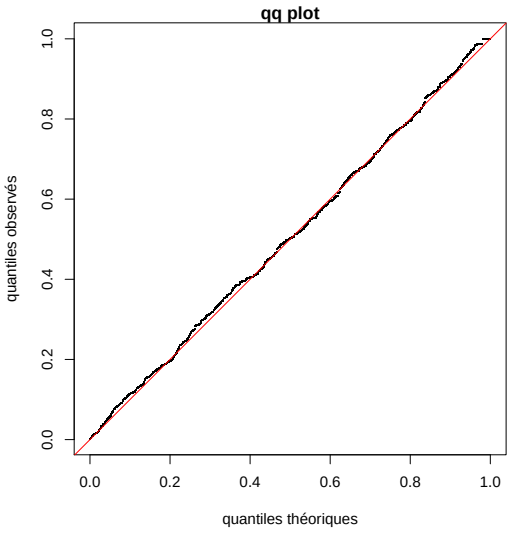
On peut commencer par s'intéresser aux propriétés de la procédure pour $\beta_1 = 0$:

```
> R <- regression(50, 0)
> R[,1:10]
               [,1]      [,2]        [,3]       [,4]        [,5]       [,6]
Estimate   0.5511424 0.1464328 -0.06938821 -0.1512644 -0.05118283 -0.1201808
Std. Error 0.4418333 0.4287396  0.41710790  0.4109866  0.40009030  0.3980172
z value    1.2473990 0.3415426 -0.16635554 -0.3680520 -0.12792820 -0.3019488
Pr(>|z|)   0.2122512 0.7326952  0.86787716  0.7128345  0.89820580  0.7626911
                [,7]       [,8]        [,9]     [,10]
Estimate   -0.07728143 -0.2796281 -1.28297073 0.4569095
Std. Error  0.41928578  0.3764119  0.53940339 0.4078066
z value    -0.18431685 -0.7428779 -2.37849957 1.1204072
Pr(>|z|)    0.85376490  0.4575556  0.01738326 0.2625403
> hist(R[1,], breaks=20, freq=FALSE)
> sum(R[4,] < 0.05) / 1000
[1] 0.043
```



Un qq-plot (diagramme quantile-quantile) s'impose pour comparer la distribution de la p-valeur avec sa distribution attendue :

```
> plot( ppoints(1000), sort(R[4,]), main='qq plot', xlab='quantiles théoriques',
       ylab = 'quantiles observés', pch=".")
> abline(0,1,col="red")
```

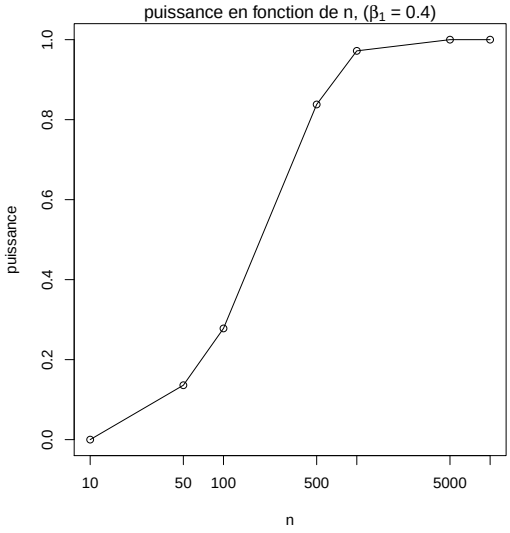


On crée une fonction pour l'estimation de la puissance (pour un test à un seuil α donné) :

```
puissance <- function(n, beta1, alpha = 0.05, B = 1000) {
  R <- regression(n, beta1, B = B)
  sum(R[4,] < alpha)/B
}
```

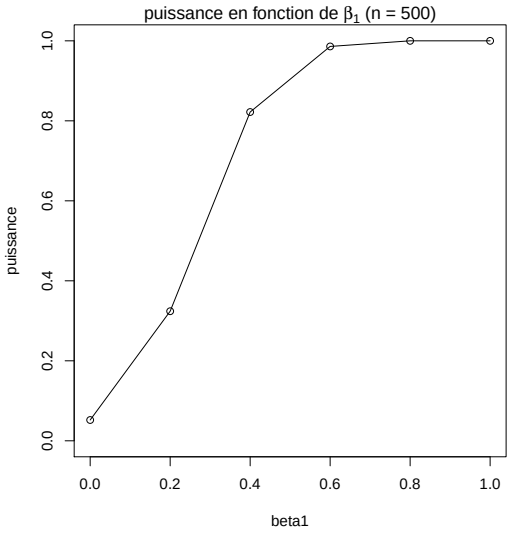
On peut s'intéresser à la puissance en fonction de n :

```
> n <- c(10,50,100,500,1000,5000,10000)
> pow <- sapply(n, puissance, beta1 = 0.4, B = 500)
> plot(n, pow, type='o', ylim=c(0,1), ylab="puissance", log="x",
      main = expression(paste( "puissance en fonction de n, (", beta[1], " = 0.4)")))
```



Ou en fonction de β :

```
> beta1 <- seq(0, 1, by = 0.2)
> pow <- sapply(beta1, puissance, n = 500, B = 500)
> plot(beta1, pow, type='o', ylim=c(0,1), ylab="puissance",
      main = expression(paste("puissance en fonction de ", beta[1], " (n = ", 500, ")")) )
```



8.1.5 Biais et la variance d'un estimateur

On considère un échantillon de n valeurs prises dans une loi de Poisson de paramètre inconnu λ. Il est naturel d'estimer λ par la moyenne empirique $\hat{\mu}$; cependant la variance empirique $\widehat{\sigma^2}$ fournit un autre estimateur. Quel est le plus efficace des deux ? On peut aborder la question d'un point de vue théorique... ou par la méthode de Monte-Carlo.

Essayons avec $n = 20$ et $\lambda = 2$. On fait B = 5000 tirages des deux estimateurs et on compare leur variance.

```
> n <- 20
> B <- 5000
> set.seed(1414)
> X <- matrix( rpois(n*B, lambda = 2), ncol = n )
> mu <- rowMeans(X)
> s2 <- apply(X,1,var)
> mean(mu)
[1] 1.99623
> mean(s2)
[1] 1.995945
> var(mu)
[1] 0.1014686
> var(s2)
[1] 0.5122463
```

8.2 Comment ça marche ? !

Il est utile de savoir un peu ce qu'il y a sous le capot. Comment l'ordinateur fait-il pour générer des valeurs aléatoires ? En fait, ces valeurs ne sont pas aléatoires. Elles sont calculées au fur et à mesure des besoins. On parle de suite pseudo-aléatoire.

Ceci a une conséquence intéressante : en prenant le même point de départ, on produira toujours la même séquence. C'est utile en particulier quand on veut pouvoir reproduire un résultat (par exemple pour traquer une erreur dans un programme). Ceci peut se faire en R en utilisant la fonction `set.seed()` :

```
> set.seed(784)
> runif(5)
[1] 0.0934020 0.5332140 0.6561710 0.8381604 0.9410595
> runif(5)
```

```
[1] 0.09626372 0.49695259 0.35868296 0.12150020 0.65276974
> set.seed(784)
> runif(5)
[1] 0.0934020 0.5332140 0.6561710 0.8381604 0.9410595
```

Dans la suite, nous allons montrer brièvement le principe de ces générateurs.

8.2.1 Générateurs congruentiels linéaires

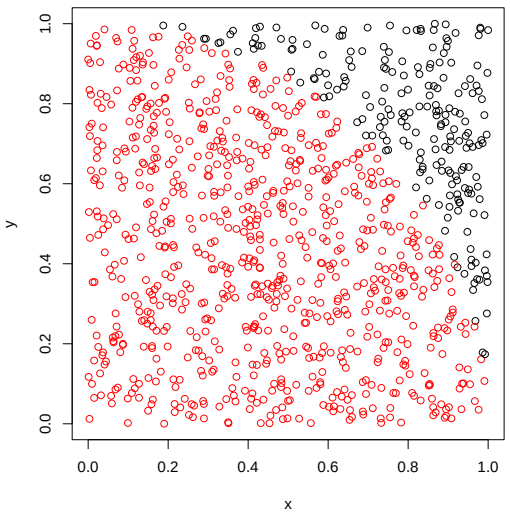
Les générateurs congruentiels linéaires ont été (et restent) populaires. Le principe est le suivant : on choisit un *module* m (assez grand) et un nombre a (sans diviseur commun avec m). On a une « graine » `seed` qui est un nombre entre 1 et $m-1$; quand l'utilisateur veut un nombre uniforme dans $[0,1]$, on remplace `seed` par le reste de la divition de $a \times$ `seed` par m (opération modulo, réalisée en R avec l'opérateur `%%`), et on fournit à l'utilisateur la valeur $\frac{1}{m}$ `seed`.

Voici une implémentation en R du générateur RANDU, qui fut distribué par IBM dans une librairie scientifique dans les années 60 et 70 – et fut beaucoup utilisé. Il prend $m = 2^{31}$ et $a = 65539$. Notez l'usage de l'opérateur `<<-` qui permet de modifier une variable globale.

```
> seed <- 1
> RANDU <- function() { seed <<- (65539 * seed) %% (2**31); seed/(2**31) }
> c(RANDU(), RANDU(), RANDU())
[1] 3.051898e-05 1.831097e-04 8.239872e-04
> seed
[1] 1769499
> seed <- 1
> c(RANDU(), RANDU(), RANDU())
[1] 3.051898e-05 1.831097e-04 8.239872e-04
```

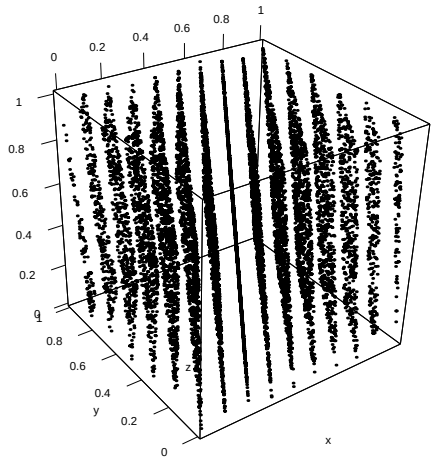
On peut par exemple estimer π avec ce petit générateur (pourquoi est-ce que ça marche ?)...

```
> B <- 1e3
> x <- y <- numeric(B);
> for(i in 1:B) { x[i] <- RANDU(); y[i] <- RANDU() }
> plot(x,y, col=ifelse(x**2+y**2<1, "red", "black"))
> 4*sum(x**2+y**2<1)/B
[1] 3.164
```



8.2.2 Le défaut des générateurs congruentiels linéaires

Le gros défaut de ces générateurs c'est que les tirages successifs ne sont pas indépendants. C'est particulièrement dramatique pour RANDU : quand on prend des triplets de coordonnées tirées les unes après les autres, on obtient des points (x, y, z) qui tombent dans une quinzaine de plans différents.



10000 triplets générés par RANDU

Une autre façon d'exprimer le même résultat est : la combinaison de trois valeurs successives $9u_n - 6u_{n+1} + u_{n+2}$ prend toujours des valeurs entières.

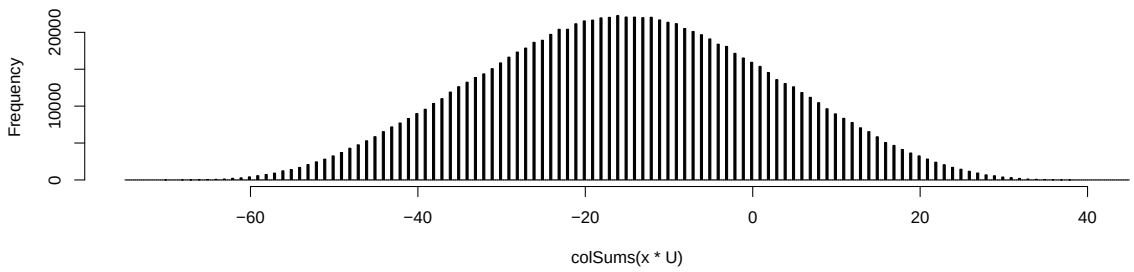
```
> 9*RANDU() - 6*RANDU() + RANDU()
[1] 4
> 9*RANDU() - 6*RANDU() + RANDU()
[1] 7
> 9*RANDU() - 6*RANDU() + RANDU()
[1] 8
```

Tous les générateurs congruentiels linéaires souffrent de ce défaut – même si tous ne sont pas aussi mauvais que RANDU. Parmi les générateurs proposés en R, il y a le générateur de Wichman-Hill, qui est un générateur congruentiel linéaire :

```
> RNGkind(kind="Wichman")
> x <- c(-10, -22, 38, -3, 1, 4, -38)
> sum( x*runif(7) )
[1] -25
> sum( x*runif(7) )
[1] 10
```

Toutes ces valeurs sont entières. Du coup, supposons qu'on veuille (justement !) étudier la distribution de $X = -10U_1 - 22U_2 + 38U_3 - 3U_4 + U_5 + 4U_6 - 38U_7$ avec les U_i indépendentes et uniformes sur $[0,1]$. Avec ce générateur ça ne marche pas très bien :

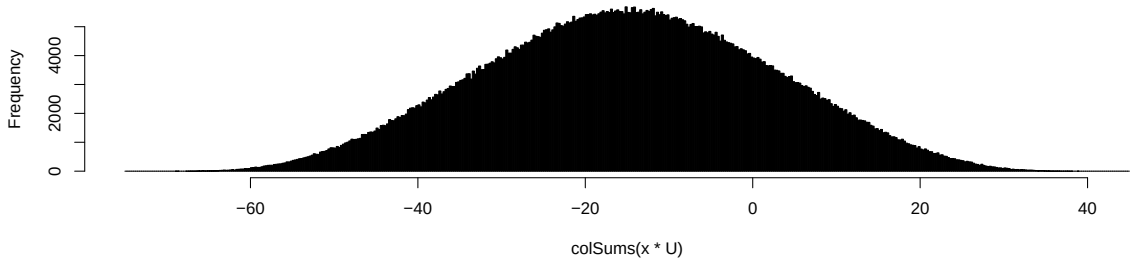
```
> RNGkind(kind="Wichman")
> x <- c(-10, -22, 38, -3, 1, 4, -38)
> U <- matrix( runif(7*1e6), nrow= 7 )
> hist( colSums(x * U), breaks = seq(-75,45,by=0.25), col="black",main="" )
```



8.2.3 Autres générateurs

R propose d'autres générateurs. Le choix par défaut est le Mersenne Twister, dont les tirages successifs peuvent être considérés comme indépendants jusqu'en dimension 623. Reprenons l'exemple précédent à titre de comparaison.

```
> RNGkind(kind="Mersenne")
> x <- c(-10, -22, 38, -3, 1, 4, -38)
> U <- matrix( runif(7*1e6), nrow= 7 )
> hist( colSums(x * U), breaks = seq(-75,45,by=0.25), col="black", main="" )
```



Voyez l'aide de RNG (en tapant `?RNG`) pour obtenir la liste des générateurs disponibles. Nous n'entrerons pas plus dans les détails ici : la morale de l'histoire est sans doute d'utiliser le générateur par défaut (Mersenne twister) qui est de bonne qualité, et de penser à utiliser `set.seed()` quand on veut des résultats reproductibles. Quand vous utilisez un autre programme que R, vérifiez que le générateur utilisé est de qualité.

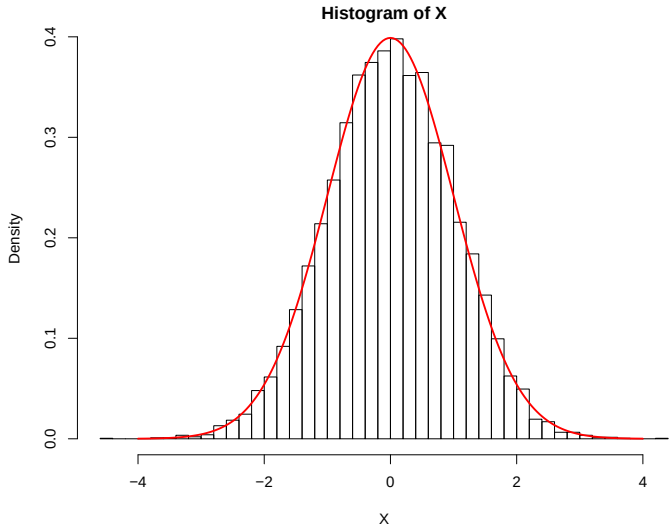
8.2.4 D'une variable uniforme à une variable aléatoire quelconque

Nous avons vu rapidement comment étaient générées les variables uniformes. Mais comment sont générées les autres variables ? Il y a une multitude de méthodes plus ou moins efficaces et plus ou moins spécifiques.

Une façon très générale de générer des variables aléatoires de loi prescrite est la méthode dite « de la transformée inverse » : si F est la fonction de répartition de la loi désirée, si U est uniforme sur $[0,1]$, alors $F^{-1}(U)$ a pour fonction de répartition F. Notez que F^{-1} est la « fonction quantile » : pour tout α entre 0 et 1, $F^{-1}(\alpha)$ est le quantile de niveau α de la loi considérée.

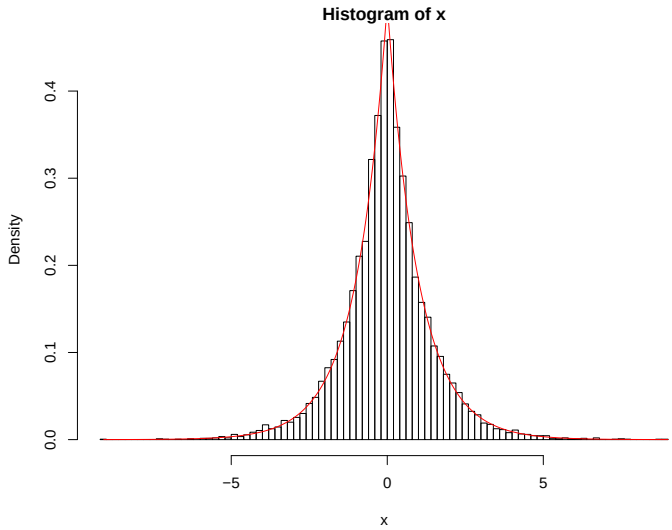
Ainsi, par exemple, pour faire un tirage dans une loi normale, on peut tirer des valeurs dans une loi uniforme et leur appliquer la fonction quantile `qnorm()` :

```
> X <- qnorm( runif(1e4) )
> hist(X, breaks = 50, freq = FALSE)
> x <- seq(-4,4,length=201)
> lines(x, dnorm(x), col="red", lwd=2)
```



Pour tirer des variables aléatoires dans la loi de Laplace, dont la densité est $f(x) = \frac{1}{2}e^{-|x|}$, on pourra utiliser la fonction suivante :

```
> rlaplace <- function(n) {u <- runif(n); ifelse(u < 0.5, log(2*u), -log(2*(1-u))) }
> x <- rlaplace(1e4)
> mean(x)
[1] -0.01362784
> var(x)
[1] 2.018548
> hist(x, breaks=100, freq=FALSE)
> xx <- seq(min(x), max(x), length=501)
> lines(xx, exp(-abs(xx))/2, col="red")
```



9 Introduction aux graphiques

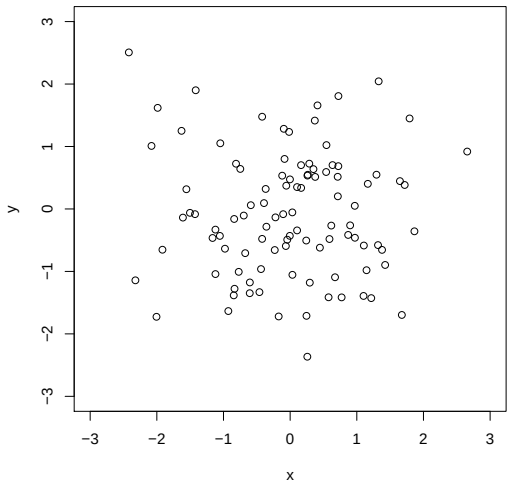
Il y a des livres entiers consacrés à la question : les fonctions graphiques de R sont très nombreuses ; et il faut y ajouter les nombreuses librairies de fonctions graphiques destinées à des usages particuliers... Nous ne faisons ici qu'effleurer la question.

9.1 Premiers graphiques

9.1.1 Nuages de points

La fonction `plot` prend en entrée deux deux vecteurs de même longueur, qui seront interprétés comme des coordonnées de points à tracer. Voici une centaine de points tirés au hasard selon un loi normale :

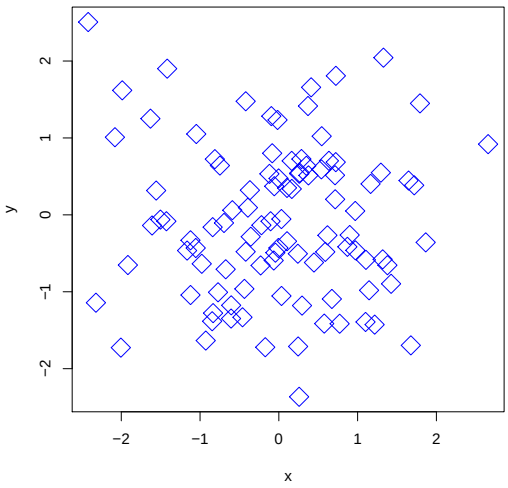
```
x <- rnorm(100)
y <- rnorm(100)
plot(x,y,xlim=c(-3,3),ylim=c(-3,3))
```



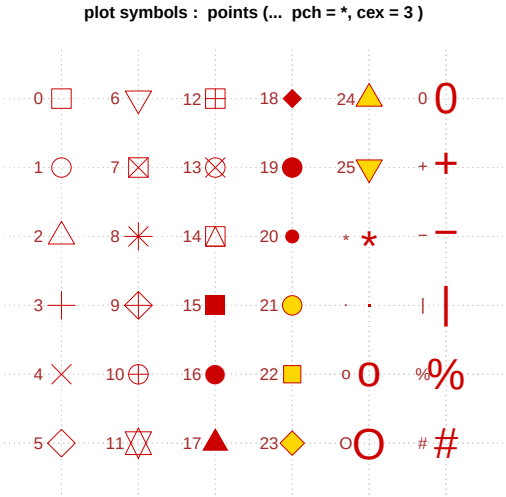
L'argument `xlim` donne les limites de l'axe des x et `ylim` celles de l'axe des y. Si on ne donne pas ces arguments R trouvera tout seul des valeurs correctes.

Par défaut, les points sont des petits cercles. L'argument `pch` permet de changer la forme des points, `col` leur couleur, et `cex` leur taille :

```
plot(x,y,pch=5,cex=2,col="blue")
```



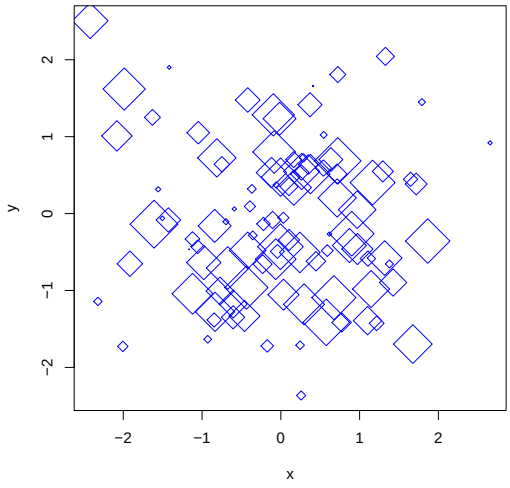
Les formes disponibles sont les suivantes.



Cette grille a été obtenue avec la fonction `pchShow()` donnée en exemple dans l'aide de `points()`.

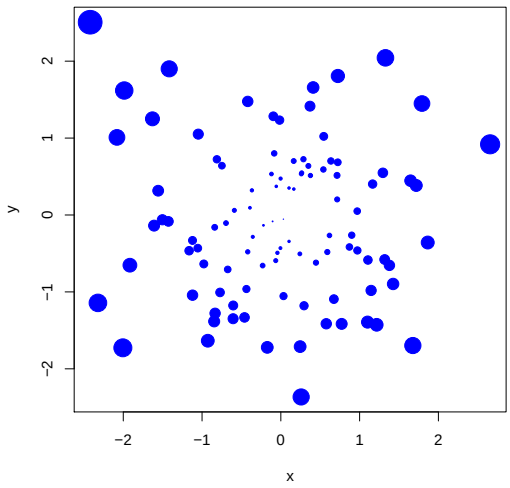
On peut également fournir à ces arguments de forme, de couleur, de taille, un vecteur de même longueur qui sera utilisé pour chacun des points, dans l'ordre des deux premiers arguments. Dans l'exemple qui suit on utilise `runif()` pour générer un vecteur d'une centaine de valeurs au hasard pour `cex`.

```
plot(x,y,pch=5,cex=5*runif(100),col="blue")
```



Dans cet exemple, on fait varier la taille comme la distance entre le point tracé et l'origine du repère :

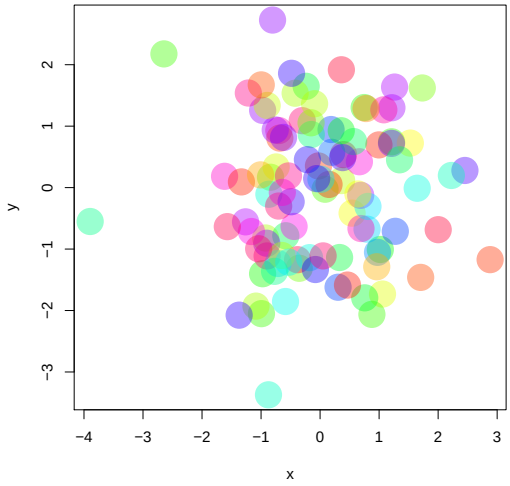
```
r <- sqrt(x^2+y^2)
plot(x,y,pch=19,cex=r,col="blue")
```



9.1.2 De toutes les couleurs

Pour spécifier la valeur de la couleur, vous pouvez utiliser des entiers, qui produiront les couleurs de base ; des noms (utilisez la fonction `colors()` pour en obtenir la liste) ; et enfin, vous pouvez définir vos propres couleurs avec `rgb()` ou `hsv()`. La fonction `rgb()` permet de définir des couleurs en dosant la quantité de rouge, de vert, de bleu (red, green, blue), et `hsv()` en définissant la nuance (hue), la saturation, et la valeur. Ces deux fonctions peuvent également utiliser un paramètre `alpha`, qui définit la transparence. Les fonctions `palette()`, `hcl()`, `gray()` et `rainbow()` peuvent aussi être utiles.

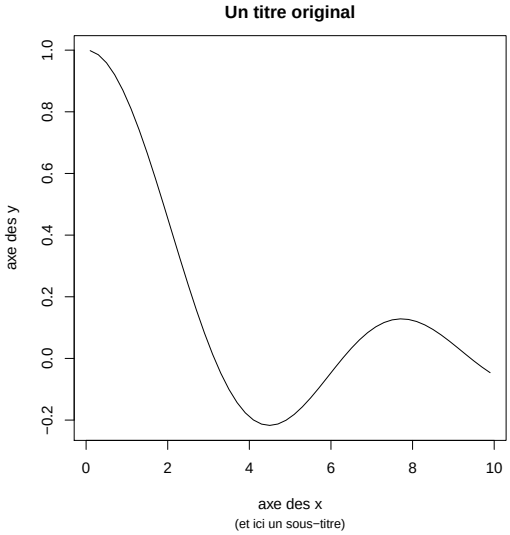
```
x <- rnorm(100)
y <- rnorm(100)
plot(x,y, pch=16, cex=4, col=hsv(runif(100), 1, 1, alpha = 0.4))
```



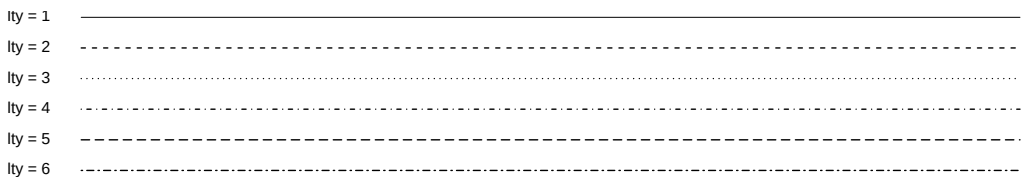
9.1.3 Lignes, etc

En passant à la fonction plot un paramètre `type="l"`, les points successifs sont reliés par des lignes. Les différentes légendes qui apparaissent sur la figure peuvent être modifiées avec `xlab`, `ylab`, etc.

```
t <- seq(0.1, 10, by = 0.2)
plot( t, sin(t)/t, type="l", xlab="axe des x",ylab="axe des y",
  main="Un titre original",
  sub="(et ici un sous-titre)",cex.sub=0.8)
```



On peut ajouter un paramètre `lty` qui change le type de ligne tracée.

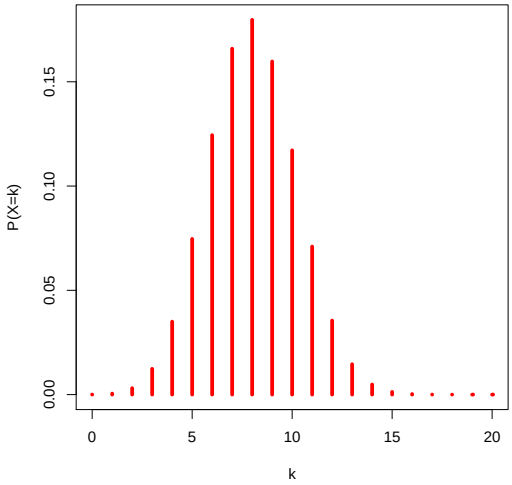


Le paramètre `type="p"` correspond au nuage de points. On essaiera également `type="o"` et `type="b"`. Le paramètre `cex.sub` qui apparaît dans l'exemple permet d'ajuster la taille du sous-titre. Il existe également des paramètres `cex.lab` et `cex.main` pour la légende des axes et le titre principal.

9.1.4 Barres

Le paramètre `type="h"` (comme « histogramme ») est bien adapté au tracé de fonctions de masse.

```
k <- 0:20
plot( k, dbinom(k, size=20, prob=0.4), col='red', lwd=4, type="h", ylab="P(X=k)")
```

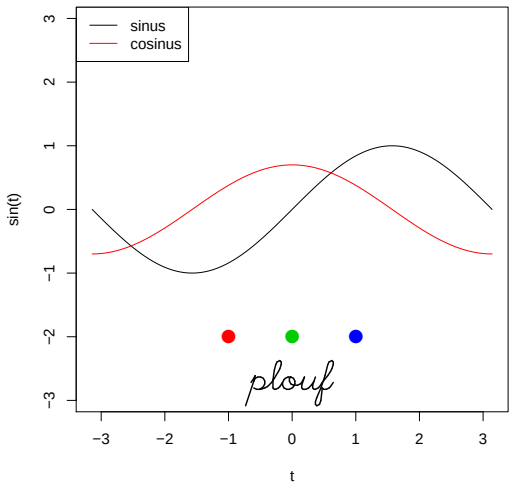


9.2 Ajouter des éléments à un graphe

9.2.1 Lignes, points, texte

Les fonctions `points()` et `lines()` permettent de superposer à un graphique des tracés similaires à ceux obtenus avec `type="p"` et `type="l"`; la fonction `legend()` permet d'ajouter un cadre de légende dans le graphe (cette fonction est très souple : cf son aide). On peut ajouter du texte avec `text()`.

```
t <- seq(-pi,pi,length=100)
plot(t,sin(t),type="l", asp=1)
lines(t,0.7*cos(t),col="red")
points(-1:1, rep(-2,3), pch=16, col=2:4, cex=2)
legend("topleft", c("sinus","cosinus"), col=c("black","red"), lwd=1)
text(0, -2.6, "plouf", cex=3, vfont=c("script","plain"))
```



Le paramètre `asp` est le « rapport d'aspect » : le fixer à 1 permet d'avoir les mêmes unités sur les deux axes.

9.3 Exporter des graphiques

Pour agrémenter vos articles à paraître dans Nature...

9.3.1 Créer un fichier pdf

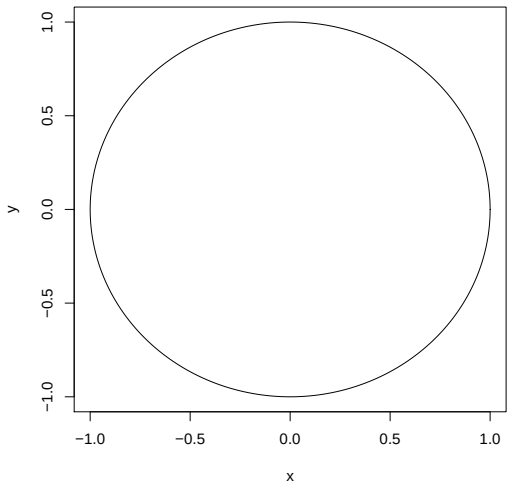
Les fichiers pdf sont des fichiers qui contiennent un dessin dans un format *vectoriel :* les coordonnées de tous les points, toutes les lignes, sont enregistrées, et le logiciel de rendu (acrobat reader par exemple) refait le dessin à chaque fois. Ceci permet de zoomer sur le dessin autant que l'on veut, sans que la qualité se dégrade.

Faire une capture d'écran (un copier/coller...) est une solution absolument proscrite !

Vous pouvez copier votre graphique dans un fichier pdf avec la commande `dev.copy2pdf()`. Il faut fournir un nom de fichier ; par exemple `dev.copy2pdf(file="graphe.pdf")` créera le fichier `graphe.pdf`. Attention à bien avoir modifié le répertoire de travail.

Le résultat peut être un peu différent de ce qu'on avait à l'écran (en particulier pour le rendu du texte). La solution est de créer son graphe dans un fichier pdf dès le départ, en créant un petit fichier de script comme celui-ci, qu'on pourra modifier pour afiner le résultat :

```
pdf(width=6, height=6, file="mon_dessin.pdf");
t <- seq(0, 2*pi, length=500)
x <- cos(t)
y <- sin(t)
plot(x,y,type="l")
dev.off()
```



Notez la commande `dev.off()` qui sert à refermer (et finaliser) le fichier créé par la commande `pdf()`. **Si vous l'oubliez votre fichier sera vide!** Et si il y a une erreur dans votre script, qui empêche R de lire cette commande alors qu'il a déjà ouvert le fichier, il est impératif que vous l'exécutiez à la main pour refermer le fichier avant de faire de nouveaux essais.

Une autre possibilité est d'utiliser la librairie Cairo (ou cairoDevice). Une fois cette librairie installée, vous avez à votre disposition une fonction `Cairo()`, qui ouvre un périphérique graphique dont les copies vers un pdf seront parfaitement fidèles.

```
library(cairoDevice)
Cairo(width=3,height=3)
```

9.3.2 Créer un fichier png

Les fichiers png sont des fichiers au format *bitmap* : au lieu de contenir les coordonnées des points et des lignes à tracer, il contient une matrice de points où l'image est dessinée (comme pour une photo numérique). On ne peut pas zoomer indéfiniment sur une telle image sans que la qualité se dégrade.

On peut copier un graphique dans un fichier png grâce à `dev2bitmap(file="fichier.png")`.

De même que pour les fichiers pdf, on peut créer d'emblée un fichier bitmap grâce aux fonctions `png()` ou `bitmap()`.

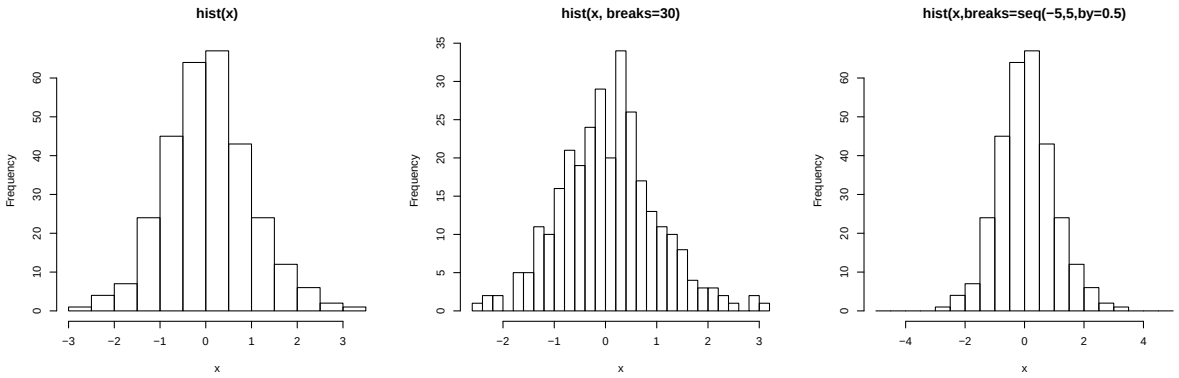
```
png(width=6, height=6, file="mon_dessin.png");
x <- cos(t)
y <- sin(t)
plot(x,y,type="l")
dev.off()
```

9.4 Quelques graphiques utilisés en statistiques

9.4.1 Histogrammes et densités

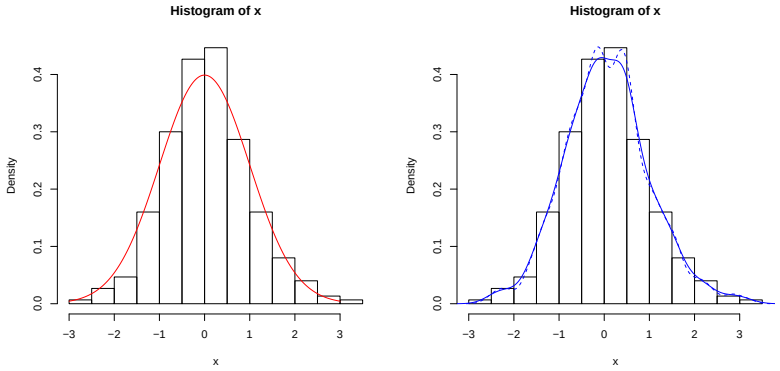
On génère un vecteur de 300 valeurs tirées au hasard suivant une loi normale standard, et on trace l'histogramme de ces valeurs : voir ci-dessous, trois histogrammes, pour illustrer le paramètre `breaks`, et du même coup l'utilisation de `par(mfrow = ...)` pour placer plusieurs graphes côte-à-côte.

```
x <- rnorm(300);
dev.new(width=12, height=4);
par(mfrow=c(1,3));
hist(x);
hist(x,breaks=30);
hist(x, breaks=seq(-5,5,by=0.5));
```



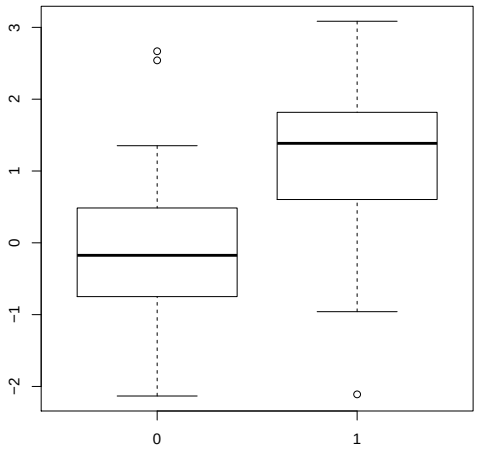
Avec le paramètre `freq=FALSE`, l'échelle de l'axe des y est modifiée de façon à ce que l'histogramme approche une densité. Dans l'exemple qui suit, nous y superposons d'abord la densité de la loi normale (la »vraie« densité qui a servi à générer le vecteur `x`), puis deux densités estimées à partir de `x` avec la fonction `density()` (qui utilise un estimateur à noyau).

```
dev.new(width=12, height=6);
par(mfrow=c(1,2));
hist(x, freq=FALSE);
t <- seq(-3,3,length=200)
lines(t,dnorm(t),col="red")
hist(x, freq=FALSE);
d <- density(x)
lines(d$x, d$y, col="blue")
d <- density(x, bw=0.18)
lines(d$x, d$y, col="blue", lty=2)
```



9.4.2 Boîtes à moustaches

```
x <- sample(c(0,1), 100, replace=TRUE)
y <- x+rnorm(100)
boxplot(y~x)
```



On obtient la même chose avec plot si on transforme au préalable x en facteur :

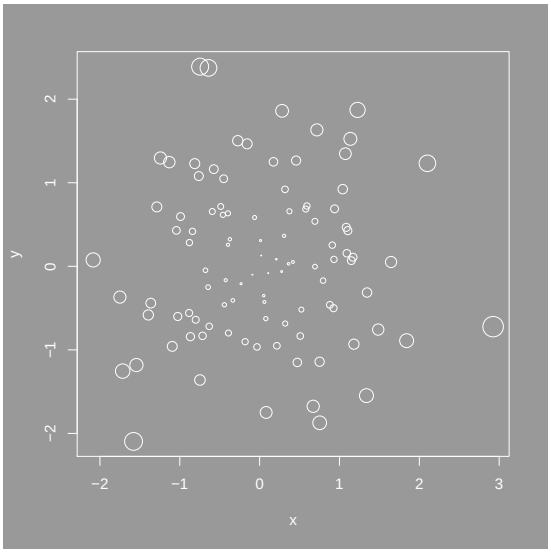
```
x <- as.factor(x)
plot(y~x)
```

9.5 Encore plus de contrôle

Nous ne donnerons ici que quelques pistes de lecture (le sujet est vaste) !

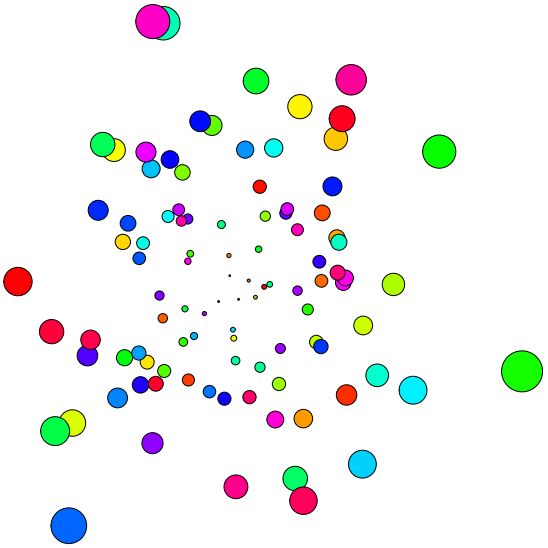
– Lire le manuel de la fonction `par` est fastidieux mais indispensable.

```
par(bg = gray(.6), fg = "white", col.axis = "white", col.lab = "white")
x <- rnorm(100); y <- rnorm(100); r <- sqrt(x**2 + y**2)
plot(x, y, cex = r)
```



– Le manuel de `plot` est moins informatif que celui de `plot.default`, qui vous apprendra par exemple à faire des choses comme ça :

```
par(mar=c(0,0,0,0))
plot(x, y, cex = 2*r, ann = FALSE, axes = FALSE, bg = rainbow(100), pch = 21)
```



– `layout` permet de diviser la fenêtre en sous-graphes, de façon plus souple que l'utilisation de `mfrow` ou `mfcol` rencontrée plus haut (histogrammes).

```
par(mar=c(3,3,0.2,0.2))
layout( matrix(c(1,2,3,4), nrow = 2), width = c(2,3), height = c(2,3))
plot(0,0,xlim=c(-1,1), ylim=c(-1,1), type="n", ann = FALSE, axes = FALSE)
text(0, 0, "Hop!", cex=4, family = "HersheyGothicGerman")
plot(dnorm, xlim=c(-3,3))
hist(x)
plot(x, y, cex = r)
```