




Introduction au logiciel

Hervé CARDOT



IMB, Université de Bourgogne
herve.cardot@u-bourgogne.fr




Master 1
2011-2012


Plan

- ▶ Quelques éléments historiques.
- ▶ Installer et lancer .
- ▶ Les objets essentiels : vecteurs, matrices, listes et data.frame
- ▶ Statistique élémentaire et génération de nombres aléatoires
- ▶ Les graphiques sous .
- ▶ Créer ses propres fonctions.
- ▶ Comment gagner du temps en évitant les boucles.
- ▶ Entrées-Sorties : sauvegarde et importation de données.
- ▶ Quelques exemples d'analyses avec  :
 - ▶ Éléments de statistique descriptive
 - ▶ Estimation de π à l'aide d'une méthode (probabiliste) de Monte Carlo.
 - ▶ Simulations du mouvement Brownien.


Un peu d'histoire

- ▶  est un logiciel libre (et donc **gratuit**) distribué par 'GNU Public Licence' spécialisé dans l'analyse statistique et la représentation graphique de données.
 - C'est au départ un clone du logiciel S+ (logiciel payant) qui a petit à petit acquis son autonomie ( existe depuis une dizaine d'années) et est devenu une référence dans le monde de la statistique de part son caractère libre qui en fait un outil **très dynamique**.
 - Il est maintenant disponible pour la plupart des plateformes (Linux, Windows, Mac OSX) sur le site :
<http://cran.r-project.org/>
- ▶ On peut développer ses propres **librairies** et les partager avec le monde scientifique via internet.
 - Il existe maintenant plus de 1000 librairies ainsi disponibles, spécialisées pour la plupart sur des sujets pointus de statistique et probabilités (MCMC, traitement d'images, génétique statistique, algorithmes stochastiques, ...).

- ▶ Le langage de programmation  est, comme Matlab ou Scilab, un langage *évolué* (interprété) basé sur le **calcul matriciel** (\neq C, C++, Fortran qui sont des langages compilés) et la manipulation simple d'objets complexes (listes, data.frame).
 - Sa simplicité d'utilisation permet de programmer rapidement des algorithmes évolués.
 - Initialement dédié à la statistique, le langage \mathbb{R} est maintenant suffisamment puissant (et précis) pour le calcul scientifique et l'ingénierie mathématique (domaine de prédilection de Matlab).
- ▶ Bibliographie.
 - De nombreuses doc. sur internet et à la bibliothèque.
 - ▶ "An Introduction to 
(<http://cran.r-project.org/doc/manuals/R-intro.pdf>)
 - ▶ " pour les débutants" , Emmanuel Paradis
http://cran.r-project.org/doc/contrib/Paradis-rdebuts_fr.pdf
 - ▶ [http://fr.wikipedia.org/wiki/R_\(logiciel\)](http://fr.wikipedia.org/wiki/R_(logiciel))
 - ▶ ...

L'aide générale au format HTML est disponible avec la commande en ligne sous  : `>help.start()`

Installation de

On télécharge gratuitement  sur le site <http://cran.r-project.org/> et on sélectionne l'OS voulu (Linux, windows, ...)



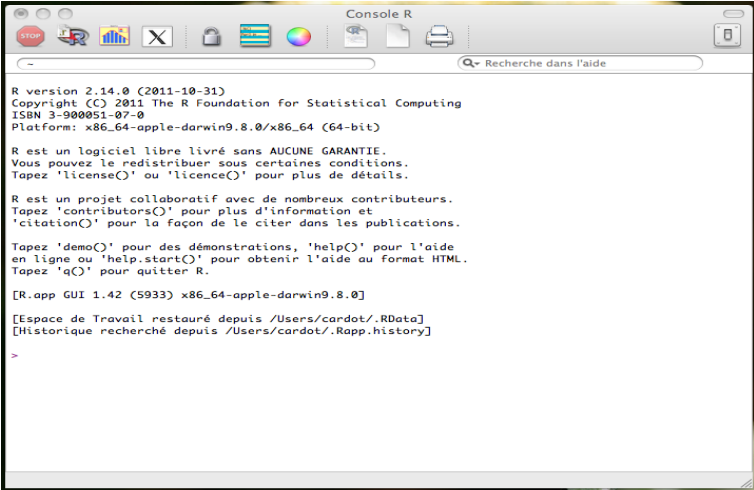
The screenshot shows a web browser window with the address bar displaying "cran.r-project.org". The page title is "The Comprehensive R Archive Network". On the left side, there is a navigation menu with links for "CRAN", "Mirrors", "What's new?", "Task Views", "Search", "About R", "R Homepage", "The R Journal", "Software", "R Sources", "R Binaries", "Packages", "Other", "Documentation", "Manuals", "FAQs", and "Contributed". The main content area is titled "Download and Install R" and contains the following text: "Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:" followed by a list of three links: "Download R for Linux", "Download R for MacOS X", and "Download R for Windows". Below this, there is a section titled "Source Code for all Platforms" with the text: "Windows and Mac users most likely want to download the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!" followed by a list of four items: "The latest release (2011-10-31): [R-2.14.0.tar.gz](#) (read [what's new](#) in the latest version).", "Sources of [R alpha and beta releases](#) (daily snapshots, created only in time periods before a planned release).", "Daily snapshots of current patched and development versions are [available here](#). Please read about [new features and bug fixes](#) before filing corresponding feature requests or bug reports.", and "Source code of older versions of R is [available here](#)." The final item in the list is "Contributed extension [packages](#)".

Les manuels et bibliothèques additionnelles (packages) sont également disponibles sur ce site.

Démarrage de

Pour lancer , on clique sur l'icône  et la fenêtre de commandes apparaît.

Je dispose de la version 2.14 qui date d'octobre 2011.



```
R version 2.14.0 (2011-10-31)
Copyright (C) 2011 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: x86_64-apple-darwin9.8.0/x86_64 (64-bit)

R est un logiciel libre livré sans AUCUNE GARANTIE.
Vous pouvez le redistribuer sous certaines conditions.
Tapez 'license()' ou 'licence()' pour plus de détails.

R est un projet collaboratif avec de nombreux contributeurs.
Tapez 'contributors()' pour plus d'information et
'citation()' pour la façon de le citer dans les publications.

Tapez 'demo()' pour des démonstrations, 'help()' pour l'aide
en ligne ou 'help.start()' pour obtenir l'aide au format HTML.
Tapez 'q()' pour quitter R.

[R.app GUI 1.42 (5933) x86_64-apple-darwin9.8.0]

[Espace de Travail restauré depuis /Users/cardot/.RData]
[Historique recherché depuis /Users/cardot/.Rapp.history]

>
```

Le prompteur `>` indique que  est prêt à recevoir les commandes (en ligne).

Pour quitter la session, il faut taper `q()`

Premiers objets : vecteurs et matrices

Les vecteurs et les matrices sont les objets de base dans .

- On peut *créer manuellement* un vecteur à l'aide de la commande `c(elt1,elt2, ...)`.

Les composantes `elt1,elt2, ...` du vecteur peuvent être numériques (réelles ou complexes), logiques (TRUE, FALSE) ou alphanumériques (chaines de caractères).

```
> a=c(1,-2,1.35)
> a
[1] 1.00 -2.00 1.35
> a[2]
[1] -2
> b=a[c(1,3)]
> log(a)
[1] 0.0000000      NaN 0.3001046
Warning message:
In log(a) : production de NaN
> log(b)
[1] 0.0000000 0.3001046
> cos(a)
[1] 0.5403023 -0.4161468 0.2190067
> exp(a)
[1] 2.7182818 0.1353353 3.8574255
> a-2*a
[1] -1.00 2.00 -1.35
> 1/a^2
[1] 1.0000000 0.2500000 0.5486968
```

On crée un vecteur `a` qui a 3 éléments.

On extrait le second élément, puis les 1er et 3ème. Ce nouveau vecteur est appelé `b`.

L'application de la fonction `log` au vecteur `a` (c-à-d à toutes ses composantes) renvoie un message d'avertissement !!

Les fonctions mathématiques classiques (`abs`, `sqrt`, `cos`, `sin`, `tan`, `exp`, `log`, `log10`, `asin`, `acos`, `^`, ...) sont présentes.

Manipulations de base sur les vecteurs

- On peut concaténer plusieurs vecteurs, extraire des éléments, faire des tests logiques simultanément sur toutes les composantes

```
> a/a
[1] 1 1 1
> a*a - a^2
[1] 0 0 0
> is.numeric(a)
[1] TRUE
> a>0
[1] TRUE FALSE TRUE
> a[a>0]
[1] 1.00 1.35
> ab = c(a,b)
> ab
[1] 1.00 -2.00 1.35 1.00 1.35
> length(ab)
[1] 5
> ab[-1]
[1] -2.00 1.35 1.00 1.35
> max(ab)
[1] 1.35
> which.max(ab)
[1] 3
> ab == max(ab)
[1] FALSE FALSE TRUE FALSE TRUE
> which(ab==max(ab))
[1] 3 5
> a<=1 & a>-1
[1] TRUE FALSE FALSE
```

Les opérations s'effectuent sur les vecteurs composantes par composantes.

Elles sont très rapides et ne nécessitent pas l'utilisation de boucles (for ... end comme en C par exemple).

Le vecteur a est numérique, sa 2nde composante n'est pas positive (FALSE).

La longueur du nouveau vecteur ab est de 5. Sa plus grande valeur est 1.35

Les fonctions logiques (>, ==, <, <=, !=) renvoient un vecteur de booléens (TRUE, FALSE).

Les opérateurs logiques sont & (et) et | (ou).

Quelques fonctions utiles sur les vecteurs

Les vecteurs créés a, A, B, \dots sont stockés dans la mémoire vive de l'ordinateur.

Le logiciel \mathbb{R} peut parfois nécessiter une mémoire vive importante (2 Gigas voire plus) lorsqu'on manipule de très gros objets.

Il faut noter que \mathbb{R} distingue les minuscules des majuscules.

```
> sort(a)
[1] -2.00  1.00  1.35
> order(a)
[1] 2 1 3
> a[order(a)]
[1] -2.00  1.00  1.35
> A = seq(-1,1,length=3)
> A
[1] -1  0  1
> A = seq(0,1,length=4)
> A
[1] 0.0000000 0.3333333 0.6666667 1.0000000
> A[c(1,3,4)]
[1] 0.0000000 0.6666667 1.0000000
> B = c(1:3,7)
> B
[1] 1 2 3 7
> A*B
[1] 0.0000000 0.6666667 2.0000000 7.0000000
> 3*A[2]
[1] 1
```

La fonction `sort` trie les éléments de `a` dans l'ordre croissant, la fonction `order` indique l'ordre des éléments.

La fonction `seq` crée une suite de 3 points équidistants entre -1 et 1, puis de 4 points entre 0 et 1.

On peut afficher certains éléments (1,3,4) du vecteur `A`.

On crée ensuite automatiquement un vecteur `B` et on effectue une multiplication *terme à terme* avec `A`.

Création de matrices

Les matrices sont également les objets de base pour \mathbb{R} . On peut effectuer sur les matrices de nombreuses manipulations de manière très simple.

```
> AO = matrix(c(1:6),ncol=2)
> AO
  [,1] [,2]
[1,]  1  4
[2,]  2  5
[3,]  3  6
> A = matrix(c(1:6),ncol=2,byrow=TRUE)
> A
  [,1] [,2]
[1,]  1  2
[2,]  3  4
[3,]  5  6
> A[1,2]
[1] 2
> A[1,]
[1] 1 2
> A[,1]
[1] 1 3 5
> dim(A)
[1] 3 2
> t(A)
  [,1] [,2] [,3]
[1,]  1  3  5
[2,]  2  4  6
> C = diag(c(1,2)) ### matrice diagonale
> C
  [,1] [,2]
[1,]  1  0
[2,]  0  2
```

La fonction `matrix` crée une matrice à 2 colonnes en "empilant" les éléments (1, 2, ..., 6) en "colonnes".

Pour la matrice `A`, l'option `byrow=TRUE` empile les éléments dans le sens des lignes.

On peut sélectionner (avec les conventions "usuelles") des éléments, des lignes ou des colonnes d'une matrice.

La fonction `dim` renvoie le vecteur qui contient le nombre de lignes et de colonnes de la matrice.

La transposée est obtenue avec `t(A)`.

Il est également possible de créer directement des matrices diagonales.

Manipulation de matrices

```
> B = t(A)%*%A ## Produit matriciel usuel
> B
      [,1] [,2]
[1,]   35   44
[2,]   44   56
> B*B ## produit terme a terme
      [,1] [,2]
[1,] 1225 1936
[2,] 1936 3136
> B[-1,]
[1] 44 56
> B[,-2]
[1] 35 44
>
> cbind(B,B) ## concaténation par colonne
      [,1] [,2] [,3] [,4]
[1,]   35   44   35   44
[2,]   44   56   44   56
> rbind(B,B) ## par ligne
      [,1] [,2]
[1,]   35   44
[2,]   44   56
[3,]   35   44
[4,]   44   56
>
> solve(B) ### inverse de B
      [,1] [,2]
[1,] 2.333333 -1.833333
[2,] -1.833333 1.458333
```

Le produit matriciel usuel s'effectue avec la commande `%*%` tandis que `*` effectue le produit terme à terme de deux matrices (de dimension compatible).

On peut supprimer une ou plusieurs lignes (ou colonnes) d'une matrice.

La concaténation de matrices (de dimension compatible), colonne par colonne s'effectue avec la commande `cbind`.

La commande `rbind` permet la concaténation ligne par ligne.

La fonction `solve` calcule l'inverse d'une matrice

Diagonalisation et résolution de systèmes linéaires

```
> eigen(B)
$values
[1] 90.7354949  0.2645051

$vectors
      [,1]      [,2]
[1,] 0.6196295 -0.7848945
[2,] 0.7848945  0.6196295

> y = c(1,2)

> solve(B)%*%y ## a éviter
      [,1]
[1,] -1.333333
[2,]  1.083333

> x= solve(B,y) ## a privilegier
> x
[1] -1.333333  1.083333

> B%*%x
      [,1]
[1,]    1
[2,]    2
```

La fonction `eigen` permet de diagonaliser une matrice carrée (symétrique ou non, valeurs propres peuvent être complexes).

Elle renvoie un *liste* constituée du vecteur des valeurs propres (`$values`) et de la matrice des vecteurs propres (`$vectors`) qui sont rangés en colonne.

La résolution du système linéaire $Bx = y$ peut s'effectuer en inversant la matrice B puis en multipliant par x .

Cette technique est moins précise numériquement et moins rapide que la seconde méthode `solve(B,y)` qui ne nécessite pas de calculer l'inverse de B .

Une nouvelle structure de données : les listes

Les listes sont des collections d'objets (vecteurs, matrices, ...) qui ne sont pas nécessairement du même type.


Elles sont utilisées par de nombreuses fonctions pour retourner les résultats et permettent en particulier de stocker et manipuler simplement des objets de longueurs différentes dans une même structure.

```
> eigen(B)
$values
[1] 90.7354949  0.2645051

$vectors
      [,1]      [,2]
[1,] 0.6196295 -0.7848945
[2,] 0.7848945  0.6196295

> Malist = list(nom=c("Truc", "Machin"),
               y=c(1,2,5), M=c(TRUE, FALSE))

> Malist$y
[1] 1 2 5
> Malist[[1]] ## = Malist$nom
[1] "Truc"  "Machin"
> Malist$M ## =Malist[[3]]
[1] TRUE FALSE
```

La fonction `eigen` est un exemple de fonction  dont les résultats sont fournis sous la forme d'une liste.

On crée une liste avec la fonction `list`.

Chaque élément de la liste peut être appelé soit en utilisant

- son nom précédé de `$`.
- Son indice entre double crochet `[[.]]`.

Une structure importante en statistique : les `data.frame`

Les `data.frame` se présentent sous la forme d'une matrice dont les colonnes peuvent être associées à des objets de différents modes (numeric, character, ...). Ils constituent une classe particulière de listes où chaque élément de la liste a la même longueur et est associé à une colonne.

Ce format est bien adapté au stockage de données statistiques :

- ▶ individus en lignes
- ▶ variables (quantitatives et qualitatives) en colonnes. Chaque colonne a un nom (liste).

Un petit exemple

```
> iris.f[c(1:5),]
  Sepale.long Sepale.larg Petale.long Petale.larg Espece
1          5.1          3.5          1.4          0.2 setosa
2          4.9          3.0          1.4          0.2 setosa
3          4.7          3.2          1.3          0.2 setosa
4          4.6          3.1          1.5          0.2 setosa
5          5.0          3.6          1.4          0.2 setosa
.....
> dim(iris.f)
[1] 150  5
> iris.f$Sepale.long[1:5]
[1] 5.1 4.9 4.7 4.6 5.0
> iris.f[[2]][1:5]
[1] 3.5 3.0 3.2 3.1 3.6
```

Le tableau `iris.f`, est de type `data.frame`, il contient des données sur 3 espèces d'iris.

On dispose de 150 fleurs sur lesquelles on a mesuré 5 caractéristiques (taille du `data.frame` : 150×5).

Data.frame et array

Les `data.frame` se manipulent à la fois comme des matrices (indices lignes et colonnes) et comme des listes (appel de colonnes par leur nom).

De nombreuses fonctions élémentaires et de statistique avancée sont configurées pour agir sur les structures de ce type.

► summary

```
> summary(iris.f)
  Sepale.long   Sepale.larg   Petale.long   Petale.larg   Espece
Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100   setosa   :50
1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300   versicolor:50
Median :5.800   Median :3.000   Median :4.350   Median :1.300   virginica :50
Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
```

- La plupart des fonctions liées à la modélisation statistique : modèle linéaire (`lm`), analyse en composantes principales (`princomp`), modèle linéaire généralisé (`glm`), analyse discriminante (`lda`), modèles additifs (`gam`), ... Ces fonctions spécifiques seront étudiées dans vos cours de statistique (M1 et M2).

Pour finir, on peut créer des tableaux (de type matrice) à plus de deux dimensions (évolution d'images dans le temps). La structure s'appelle `array` (voir aide de `R` : `>help(array)`). Ces objets se manipulent "comme" des matrices (sélections de lignes, colonnes, individus).

Statistique élémentaire

Le calcul statistique et la simulation de phénomènes aléatoires sont les points forts de \mathbb{R} . Il existe plusieurs centaines de bibliothèques additionnelles (et livres) dédiées à ces thèmes.

Voici quelques fonctions de base en statistique (on se reportera à l'aide de \mathbb{R} pour plus de détails sur la syntaxe) :

- ▶ `mean` : calcule la moyenne d'un vecteur numérique (`mean(X)`).
- ▶ `var` et `sd` : estimation de la variance et de l'écart-type (standard deviation)
Si `X` est une matrice, la fonction `var` renvoie la matrice de variance-covariance.
(Δ : le dénominateur est $n - 1$.)
- ▶ `quantile` : calcule les quantiles d'un vecteur numérique `X`.
Les quantiles de `X` à 10%, 50% (=médiane) et 90% :
`quantile(X, probs=c(0.1, 0.5, 0.9))`
- ▶ `summary` : fournit les statistiques de bases (moyenne, min, max, médiane, ...) et agit sur les données de type `data.frame`.
Cette fonction est générale et fournit un résumé des résultats de nombreuses fonctions statistiques plus évoluées (régression, ...)
- ▶ `table` : donne les fréquences et fréquences croisées d'une ou plusieurs variables qualitatives (type facteur)

Statistique élémentaire : quelques exemples

```
> mean(iris.f$Sepale.long)
[1] 5.843333
> quantile(iris.f$Sepale.long)
 0% 25% 50% 75% 100%
4.3 5.1 5.8 6.4 7.9
> quantile(iris.f$Sepale.long,probs=c(0.05,0.95))
 5% 95%
4.600 7.255

> cov(iris.f[,1:4]) ### = var(iris.f[,1:4])
      Sepale.long Sepale.larg Petale.long Petale.larg
Sepale.long 0.68569351 -0.04243400 1.2743154 0.5162707
Sepale.larg -0.04243400 0.18997942 -0.3296564 -0.1216394
Petale.long 1.27431544 -0.32965638 3.1162779 1.2956094
Petale.larg 0.51627069 -0.12163937 1.2956094 0.5810063

> sd(iris.f[,1:4])
Sepale.long Sepale.larg Petale.long Petale.larg
0.8280661 0.4358663 1.7652982 0.7622377

> cor(iris.f[,1:4])
      Sepale.long Sepale.larg Petale.long Petale.larg
Sepale.long 1.0000000 -0.1175698 0.8717538 0.8179411
Sepale.larg -0.1175698 1.0000000 -0.4284401 -0.3661259
Petale.long 0.8717538 -0.4284401 1.0000000 0.9628654
Petale.larg 0.8179411 -0.3661259 0.9628654 1.0000000

> table(iris.f$Espece)
  setosa versicolor virginica
    50         50         50
```

Par défaut la fonction `quantile` évalue les quantiles à 25% (premier quartile), 50% (médiane), 75% (3ème quartile) ainsi que le minimum et le maximum.

La fonction `cov` est équivalente à `var` lorsque l'entrée est une matrice (ou un `data.frame`). Elle calcule les variances des variables et les covariances.

La covariance entre `Sepale.long` et `Sepale.larg` est de -0.04.

La fonction `cor` calcule les coefficients de corrélations 2 à 2.

La fonction `table` évalue les effectifs, dans l'échantillon, de chacune des espèces.

Génération de nombres aléatoires

Il est possible de simuler simplement un grand nombre de loi de probabilités différentes (une vingtaine !!!). Voici les plus classiques (simulation de n réalisations indépendantes) :

Loi de probabilité	Commande \mathbb{R}
Poisson (λ)	<code>rpois(n, λ)</code>
Binomiale (n, k, p)	<code>rbinom(n, k, p)</code>
Uniforme [a, b]	<code>runif(n, a, b)</code> (par défaut [a, b] = [0, 1])
Normale $\mathcal{N}(\mu, \sigma^2)$	<code>rnorm(n, μ, σ)</code> (par défaut $\mu = 0$ et $\sigma^2 = 1$)
Student (q)	<code>rt(n, q)</code> où q est le nombre de ddl
Fisher (q_1, q_2)	<code>rf(n, q1, q2)</code> où q_1 et q_2 sont les ddl

```
### Simulation d'une loi Uniforme sur [0,1]
> X = runif(100) ## Simule un vecteur de taille 100, réalisations d'une U[0,1].
> length(X)
[1] 100
> X[1:6]
[1] 0.6413839 0.1981813 0.1463944 0.5828089 0.3969020 0.9056888
> mean(X) ## Calcul de la moyenne du vecteur X
[1] 0.4759544
```

```
### Simulation d'une loi normale
> Z = rnorm(20,1,2) ## Z contient 20 réalisations d'une loi N(1,2^2)
> Z[1:5]
[1] 0.4313072 1.5226537 4.7373083 4.3791279 5.1497121
> Z = rnorm(20,1,2)
> Z[1:5]
[1] 2.178782 2.585600 -3.583113 -1.150690 1.540848
```

Quantiles, densités et fonctions de répartition

On peut également évaluer les quantiles de ces distributions ainsi que la valeur de la densité et de la fonction de répartition.

Il faut pour cela remplacer le `r` (pour `random`) en préfixe de chaque fonction par :

d : évaluation de la densité (cas continu) ou de la probabilité (cas discret)



q : évaluation du quantile (cas continu).

p : évaluation de la fonction de répartition.

La syntaxe dépend des paramètres de la loi considérée (voir dans l'aide). Voici quelques exemples :

```
> dnorm(0) ## resultat : [1] 0.3989423 (=1/sqrt(2*pi))
> pnorm(1.96) ## resultat : [1] 0.9750021
> pnorm(-1.96) ## resultat : [1] 0.02499790
> qnorm(0.95) ## resultat : [1] 1.644854
> punif(0.5) ## resultat : [1] 0.5
> qunif(0.5) ## resultat : [1] 0.5
> qunif(0.5,0.2,1.2) ## resultat : [1] 0.7
```

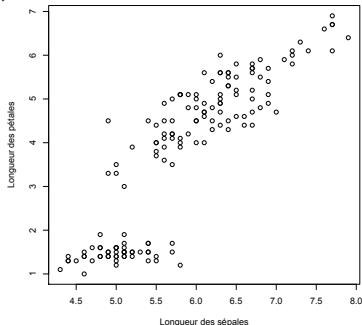
Introduction aux représentations graphiques

Les graphiques sont également un point fort de . Il offre de très nombreuses possibilités dont on peut avoir un premier aperçu avec la démonstration proposée par  : `>demo(graphics)`

Une fonction de base est la commande `plot` qui permet de tracer des nuages de points.

La syntaxe générale est la suivante pour représenter le vecteur `y` en fonction de `x`

```
> plot(x,y,xlab="Légende abscisses",ylab="Legende ordonnées")  
## Exemple, représentation de la longueur des pétales en fonction des sépales  
> plot(iris.f$Sepale.long,iris.f$Pétale.long,xlab="Longueur des sépales",  
ylab="Largeur des pétales")
```




Les options de la commande `plot` sont très nombreuses et on se reportera à l'aide (`help(plot)`) pour plus de détails :

- ▶ Ajout d'un titre avec `main` (Exemple : `plot(x,y,main="Y en fonction de X")`)
- ▶ Choix des couleurs avec `col` (Exemple : `plot(x,y,col="red")`)
- ▶ Choix de la taille des points avec `cex` (Exemple : `plot(x,y,cex=2)`)
- ▶ Choix de la forme de points avec `pch` (Exemple : `plot(x,y,pch=1)`)
- ▶ Possibilité de relier les points par des lignes avec `type` (Exemple : `plot(x,y,type="l")`)
- ▶ Préciser les limites des axes (valeur minimale et maximale) avec `xlim` et `ylim` (Exemple : `plot(x,y,xlim=c(0,2))`)

On peut également ajouter sur un graphique existant

- ▶ Des lignes (courbes) avec l'instruction `lines`
- ▶ Une légende avec l'instruction `legend`

Quelques autres fonctions graphiques utiles

Les diagrammes en barre (barplot) et les diagrammes en secteurs (camemberts, commande pie) sont faciles à tracer avec .

On peut également ajouter à un graphique des segments (segments), des flèches (arrows), du texte (text), ...

Ces formes s'ajoutent à un "plot" déjà créé.

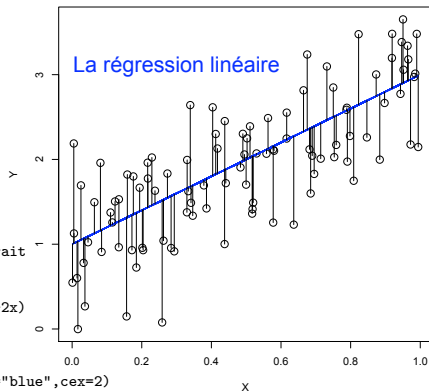
```
### Simulation des données
> X = runif(100)
> bruit = rnorm(100,sd=0.5)
> Y = 1+2*X+bruit

#### Représentations graphiques
### Nuage de points (X,Y)
> plot(X,Y,cex=1.5)

### Droite d'équation  $y = 1+2x$ 
### lwd permet d'augmenter l'épaisseur du trait
> lines(X,1+2*X,lwd=2,col="blue")

### Segments reliant les point (x,y) à (x,1+2x)
> segments(X,Y,X,1+2*X)

### Ajout d'un texte en (0.3,3.1)
> text(0.3,3.1,"La régression linéaire",col="blue",cex=2)
```

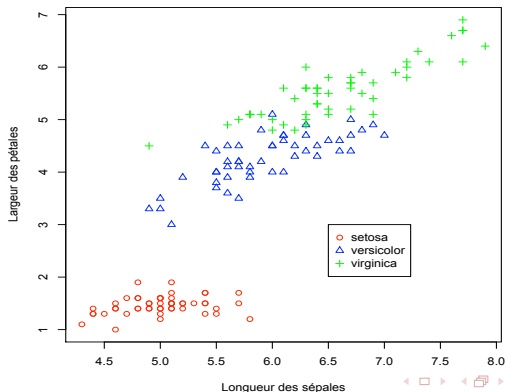


Un exemple plus sophistiqué

```
> X.dom = range(iris.f$Sepale.long) ## =c(min(iris.f$Sepale.long), max(iris.f$Sepale.long))
> Y.dom = range(iris.f$Pétale.long)

> plot(iris.f$Sepale.long[iris.f$Espece=="setosa"], iris.f$Pétale.long[iris.f$Espece=="setosa"],
xlim=X.dom,ylim=Y.dom,xlab="Longueur des sépales",ylab="Largeur des pétales",
main="Iris de Fisher",pch=1,col="red")

> points(iris.f$Sepale.long[iris.f$Espece=="versicolor"], iris.f$Pétale.long[iris.f$Espece=="versicolor"],
pch=2,col="blue")
> points(iris.f$Sepale.long[iris.f$Espece=="virginica"], iris.f$Pétale.long[iris.f$Espece=="virginica"],
pch=3,col="green")
> legend(6.5,3,c("setosa","versicolor","virginica"),col=c("red","blue","green"),pch=c(1,2,3))
```



Juxtaposition et sauvegarde de graphiques

Il est possible de juxtaposer plusieurs graphiques sur une même page en modifiant les options graphiques avec la commande `par` (qui offre de nombreuses possibilités : `> help(par)`)

```
par(mfrow=c(2,3)) ### 6 graphiques juxtaposés, 2 par ligne et 3 par colonne
```

On peut enfin sauvegarder de manière automatique les graphiques sous divers format (pdf, postscript, jpeg, ...):

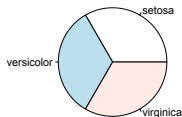
```
### Le fichier pdf "PremiereSauvegarde.pdf" est sauvé  
### dans le repertoire courant (de travail).  
### Il peut être incorporé dans un rapport  
### ou une présentation.
```

```
> pdf("PremiereSauvegarde.pdf")  
> par(mfrow=c(1,2)) ###  
> pie(table(iris.f$Espece), ## Diagramme en secteurs  
main="Fréquences des trois espèces")
```

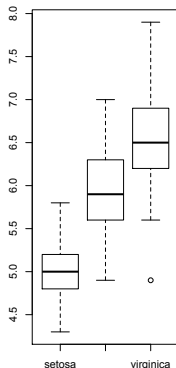
```
> with(iris.f, ## La commande with evite de repeter le nom du tableau  
boxplot(Sepale.long~Espece,main="Longueur des Sépales"))  
### La fonction boxplot (boite a moustache) représente les quartiles (25%,  
### mediane, 75%). Elle fournit une indication sur la répartition des observations.
```

```
### "Sepale.long~Espece" indique que l'on effectue l'analyse de Sepale.long en  
### fonction de la variable "Espece".  
> dev.off() ## indique la fin des instructions graphiques et effectue la sauvegarde.
```

Fréquences des trois espèces



Longueur des Sépales



Approximation du nombre π par une méthode de Monte Carlo

Nous présentons sur cet exemple les possibilités de calcul et graphiques de .

Petit problème : donner une estimation de π à l'aide de simulations.

Idée (parmi d'autres) : on simule n observations selon une loi uniforme sur le pavé $[0, 1] \times [0, 1]$ et on compte la proportion d'observations qui sont à l'intérieur du disque unité

Remarque : l'approximation d'intégrales par génération de nombres aléatoires s'appelle **méthode de Monte Carlo**.

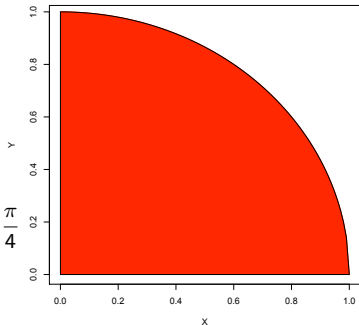
Si $X \sim [0, 1]$ et $Y \sim U[0, 1]$ indépendantes alors $(X, Y) \sim U([0, 1] \times [0, 1])$ et,

$$\Pr[(X, Y) \in [a, b] \times [c, d]] = \int_a^b \int_c^d dx dy$$

Par conséquent,

$$\Pr[X^2 + Y^2 < 1] = \iint_{\{x^2 + y^2 < 1, x \geq 0, y \geq 0\}} dx dy = \frac{\pi}{4}$$

Donc la probabilité de "tomber" dans la partie **rouge** = $\pi/4$.



Génération de n ($n=200$) réalisations
d'une loi uniforme sur $[0, 1] \times [0, 1]$.

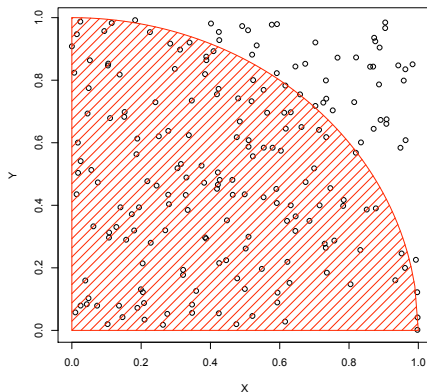
Calcul de la proportion d'observations
dans le quart de disque

Estimation de π

```
> n=200
> XY.mat = matrix(runif(2*n),ncol=2)
> est.prop = mean(XY.mat[,1]^2+XY.mat[,2]^2<1)
> est.pi = 4*est.prop
> est.pi
[1] 3.12
```

Représentation graphique de notre échantillon à l'aide de la fonction plot.

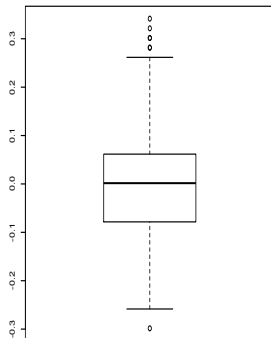
```
> plot(XY.mat[,1],XY.mat[,2],pch=1,xlab="X",ylab="Y")
> polygon(c(X.grid,0),c(sqrt(1-X.grid^2),0),density=10,col="red")
```



Quelle est la précision de la méthode ?

Nous allons répéter 500 fois la manipulation précédente. Introduisons pour cela la fonction `for` qui permet d'effectuer des boucles.

```
> n.sim=500 ## nombre de répétitions
> n=200
> diff.vecteur = rep(0,length=n.sim)
> for (i in 1:n.sim){ ## debut de la boucle
XY.mat = matrix(runif(2*n),ncol=2)
est.prop = mean(XY.mat[,1]^2+XY.mat[,2]^2<1)
est.pi = 4*est.prop
diff.vecteur[i] = pi - est.pi
} ## fin de la boucle
> summary(diff.vecteur)
##      Min.      1st Qu.      Median      Mean      3rd Qu.      Max.
## -0.298400 -0.078410  0.001593  0.001113  0.061590  0.341600
> boxplot(diff.vecteur) ## répartition des écarts à pi
```



La précision est plutôt bonne en moyenne (écart médian = 0.0015) et 50% des valeurs ont un écart dans l'intervalle $[-0.078, 0.062]$.

Exemple : simulations de mouvements browniens

Le mouvement Brownien est un processus stochastique (variable aléatoire qui évolue dans le temps) bien connu en théorie des probabilités. Il permet de modéliser

- ▶ Les trajectoires de particules soumises à des chocs aléatoires (Einstein, 1905)
- ▶ l'évolution temporelle d'actifs boursiers (Bachelier, 1900).

Voici une définition possible (on se limite à l'intervalle de temps $[0, 1]$).

*Un **mouvement Brownien** $B(t)$, pour $t \in [0, 1]$ est un processus stochastique qui vérifie*

- ▶ $B(0) = 0$, le point de départ est connu et déterministe (ici égal à 0)
- ▶ Les accroissements entre deux instants $t > s$ sont indépendants et sont distribués selon une loi normale centrée de variance $t - s$:

$$B(t) - B(s) \sim \mathcal{N}(0, t - s).$$

On peut montrer ensuite facilement que

$$\forall (s, t) \in [0, 1] \times [0, 1], \quad \mathbb{E}(B(t)) = 0 \quad \text{et} \quad \text{Cov}(B(t), B(s)) = \min(s, t)$$

Simulation d'un mouvement brownien avec

On commence par discrétiser l'intervalle de temps $[0, 1]$ en 501 points équidistant

On utilise ensuite la propriété des accroissements Gaussiens (loi normale) pour simuler chaque accroissement.

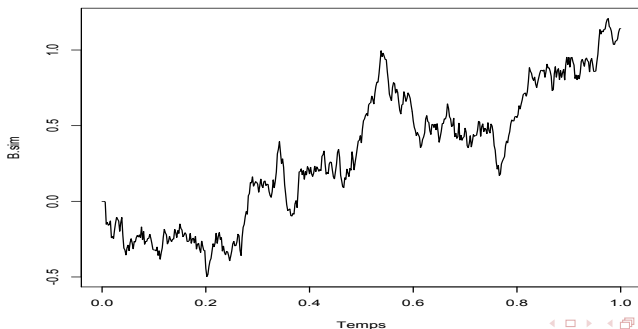
La fonction `cumsum` calcule les sommes cumulées et permet de générer une trajectoire.

```
### Simulation d'un mvt Brownien
### discretisation du temps
> temps = seq(0,1,length=501)
> pas.temps = 1/500

### Simulation des accroissements
> B.acc = rnorm(500,sd=sqrt(pas.temps))

### Simulation d'une trajectoire
> B.sim = c(0,cumsum(B.acc))

> plot(temps,B.sim,type="l",xlab="Temps")
## L'option type="l" relie les points entre eux.
```



Trajectoire moyenne des mouvements Browniens

Nous avons simulé maintenant 300 trajectoires indépendantes de mouvements Browniens et souhaitons vérifier que la trajectoire moyenne empirique est proche de 0.

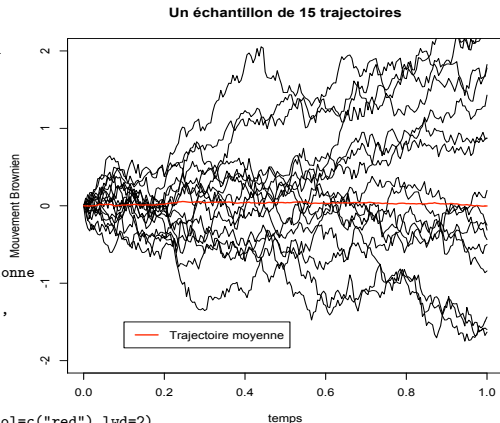
```
> n.sim=300 ### Nombre de trajectoires
> n.point = 201 ### Points de discretisation
> temps = seq(0,1,length=n.point)
> pas.temps = 1/(n.point-1)
> B.acc = matrix(rnorm((n.point-1)*n.sim,
sd=sqrt(pas.temps)),nrow=n.sim)
> B.sim = matrix(NA,ncol=n.point,nrow=n.sim)
> for (i in 1:n.sim)
{B.sim[i,] = c(0,cumsum(B.acc[i,]))}

> dim(B.sim) ## [1] 300 201

> B.mean = apply(B.sim,2,mean)
### apply calcule la moyenne pour chaque colonne

> plot(temps,B.sim[1,],xlab="temps",type="l",
ylab="Mouvement Brownien",ylim=c(-2,2))
> title("Un échantillon de 15 trajectoires")
> for (i in 1:15){lines(temps,B.sim[i+1,])}

> lines(temps,B.mean,lwd=2,col="red")
> legend(0.1,-1.5,c("Trajectoire moyenne"),col=c("red"),lwd=2)
```



Covariance empirique des mouvements Browniens

On peut aussi tracer la fonction de covariance empirique des trajectoires simulées.

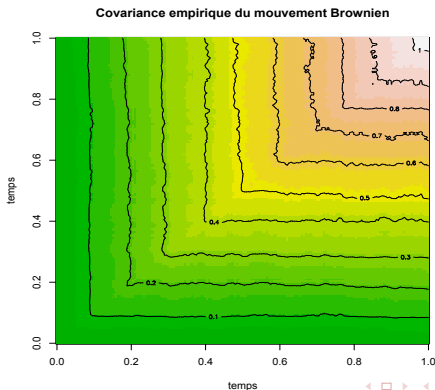
C'est une fonction de 2 variables qui doit être proche de $(s, t) \rightarrow \min(s, t)$.

On peut utiliser pour cela des fonctions graphiques 2D (image, persp, contour, ...) qui produisent de jolis graphiques mais possèdent de très nombreuses options (voir > demo(persp))


```
> B.cov.emp = cov(B.sim) ## estimation de la matrice de variance-covariance
```


```
> image(temps, temps, B.cov.emp,col=terrain.colors(20),xlab="temps",ylab="temps",  
main="Covariance empirique du mouvement Brownien")
```

```
> contour(temps, temps,B.cov.emp, add=TRUE)
```



Programmation de ma première fonction


On peut créer simplement ses propres fonctions sous .

Il faut travailler avec un **éditeur de texte** (Tinn-R par exemple : <http://sciviews.org/Tinn-R/>) plutôt que directement dans la console  :

- Sauvegarde et traces des instructions.
- Correction et modification plus facile.

```
moyvar = fonction(X){  
## Calcul de la moyenne et variance  
## Entree  
## X : vecteur de taille n  
## Sortie, liste a deux elements  
## moy = moyenne de X  
## var = variance de X
```


```
n = length(X)  
moyX = sum(X)/n  
varX = sum((X-moyX)^2)/n  
list(moy=moyX,var=varX)  
}
```

Il est quand même possible depuis la console  d'éditer et de modifier une fonction à l'aide de la commande `fix`.

```
> fix(moyvar)
```

La syntaxe est la suivante :

```
Mafonction = fonction(arg1,arg2,...)  
{  
## commentaires  
instructions  
} ## fin de la fonction
```

 renvoie le résultat de la dernière expression évaluée (dernière ligne de commande).

La fonction `moyvar` a pour entrée un vecteur (numérique) et pour sortie une liste composée de 2 éléments, `moy` et `var`.

```
> moyvar(iris.f$Sepale.long)  
$moy  
[1] 5.843333  
$var ### La variance est légèrement différente (denominateur 1/n)  
[1] 0.6811222
```


Boucles et exécutions conditionnelles

Comme dans la plupart des langages structurés, les commandes `for`, `if`, `while`, `repeat` permettent d'effectuer des boucles et des exécutions conditionnelles.

Syntaxe de `for` :

```
for (i in 1:100) {  
  suite d'instructions  
}
```

On peut remplacer `1:100` par une suite ordonnée ou non d'entiers stockés dans un vecteur.

La syntaxe de `while` et `if` sont assez similaires

```
while (expression logique est vraie) {  
  suite d'instruction à exécuter  
}  
## Exemple  
i=1  
while (i<=N){  
  if (X[i]==TRUE) {NombreP = NombreP+1}  
  i = i+1}
```

On peut fixer également la valeur de certains paramètres par défaut. Ici `N=1000` pour `res1`.

Pour effectuer le calcul sur les 10000 premiers termes (`res2`), il faut préciser `N=10000`.

```
SommeN.v1 = function(N){  
## Calcul de 1/1+1/2+...+1/N  
resultat=0  
for (i in 1:N){  
  resultat = resultat+1/i  
}  
resultat ## Valeur retournée par la fonction  
} ## Fin des instructions de la fonction
```

```
NombrePos.v1 = function(X,N=1000){  
## Calcul du nombre d'elements de X  
## qui sont 'TRUE' parmi les N premiers  
## (par défaut N=1000)  
NombreP = 0 ## initialisation du compteur  
for (i in 1:N){  
  if (X[i]==TRUE) {NombreP = NombreP+1}  
}  
NombreP  
}
```

```
> res1 = NombrePos.v1(X)  
> res2 = NombrePos.v1(X,N=10000)
```

Eviter les boucles et exécutions conditionnelles

Ⓡ n'étant pas un langage compilé (mais interprété ligne à ligne) les instructions de ce type sont exécutées lentement et on préférera, lorsque c'est possible, utiliser les outils de calcul sur les vecteurs et les matrices ("optimisés" sous Ⓡ).

```
SommeN.v2 = fonction(N){  
## calcul de 1/1+1/2+...+1/N  
vecteurN = c(1:N)  
resultat = sum(1/vecteurN)  
resultat}
```

```
> system.time(SommeN.v1(100000))  
utilisateur      système      écoulé  
   0.217         0.001         0.218  
> system.time(SommeN.v2(100000))  
utilisateur      système      écoulé  
   0.002         0.002         0.004
```

```
NombrePos.v2 = fonction(X,N){  
## Calcul du nombre d'elements de X  
## qui sont 'TRUE' parmi les N premiers  
NombreP = sum(X[1:N])  
NombreP}
```

```
> X=runif(100000) ##  
> system.time(NombrePos.v1(X>0.5,N=50000))  
utilisateur      système      écoulé  
   0.168         0.002         0.168  
> system.time(NombrePos.v2(X>0.5,50000))  
utilisateur      système      écoulé  
   0.002         0.001         0.003
```

La nouvelle version de SommeN remplace la boucle par la fonction sum.

Dans NombrePos.v2 la boucle for et le test if sont remplacés par le comptage (sum) du nombre de TRUE.

La fonction system.time permet de mesurer le temps d'exécution d'une fonction et de comparer la vitesse d'exécution de plusieurs programmes.

La fonction runif(N) simule N réalisations indépendantes d'une loi uniforme sur [0, 1]. On cherche donc le nombre de réalisations supérieures à 0.5 parmi les 50000 premières.

Les nouvelles versions (sans boucle) sont au moins 50 fois plus rapides!!!!

Retour sur la simulation des mouvements Browniens

Nous allons pouvoir maintenant créer nos propres fonctions de manière performante.

```
Simul.B = fonction(p){  
## Simulation d'une trajectoire d'un mvt Brownien sur [0,1] discretise en p points  
pas.temps = 1/(p-1)  
B.acc = rnorm((p-1),sd=sqrt(pas.temps))  
B.sim = c(0,cumsum(B.acc))  
B.sim  
}
```

L'instruction `sapply` permet ensuite d'appliquer la fonction `Simul.B` à un vecteur d'entrées et fournit en sortie la matrice des trajectoires simulées correspondant à chaque valeur du vecteur d'entrée.

```
Brown.mat = fonction(n,p){  
## Simulation de n trajectoires Browniennes sur [0,1] discretisées en p points  
## Sortie : matrice de taille n x p  
res = sapply(rep(p,length=n),Simul.B)  
t(res)  
}  
  
## Simulation de 300 trajectoires, avec 200 points de mesure  
> essai = Brown.mat(300,200)
```

Cette démarche produit des fonctions en général très rapides. On ne fait ici appel à aucune boucle.


Sauvegarde et lecture de données

La sauvegarde et la lecture des données permet de récupérer les résultats obtenus dans une session précédente, importer un fichier de données (statistiques par exemple) ou exporter des données après transformations. La fonction `ls()` donne la liste des objets présents en mémoire vive

```
>ls()
[1] "Brown.mat"      "diff.vecteur" "est.pi"        "est.prop"      "iris.f"
[8] "Simul.B"        "XY.mat"
```


Pour effacer de l'espace de travail les objets "res" et "XY.mat", il suffit de taper

```
> rm(res,XY.mat)
```


La sauvegarde de ces objets (fonctions, matrices, ...), dans un format binaire propre à  est effectuée dans le répertoire courant avec les commandes `save` et `save.image`. Pour charger en mémoire des objets sauvés à l'aide de `save` ou `save.image`, il faut utiliser l'instruction `load`

```
> save.image()
> save.image("Mesdonnees.Rdata") ## Pour sauver dans le fichier appelé "Mesdonnees.Rdata"
> save(n.sim,Simul.B,Brown.mat,file="3fns.Rdata") ## pour ne sauver qu'une selection d'objets

> load(".Rdata") ## charge en mémoire ce qui a été sauvé par save.image() par défaut
> load("Mesdonnees.Rdata") ## charge en mémoire les objets contenus dans "Mesdonnees.Rdata"
```

Lorsqu'on quitte une session avec la commande `q()`,  propose systématiquement de sauver l'espace de travail dans un fichier de type ".Rdata", et affiche le message

```
> q()
> Save workspace image? [y/n/c]:
```

Les fichiers de type .Rdata (load et save) sont d'un format qui est propre à  et ne permettent pas de communiquer avec d'autres logiciels.

Les commandes `read.table`, `read.csv`, ... permettent de lire des fichiers textes de formats reconnus (.csv, ...) par de nombreux logiciels (Excel, SAS, ...) et de sauver le résultat dans un objet de type `data.frame`.

L'exportation d'objets `data.frame` s'effectue à l'aide des commandes `write.table`, `write.csv` en précisant le nom du fichier de sortie.

```
> write.csv(iris.f, file="DonneesIris.csv") ## Sauvegarde de "iris.f" dans le fichier "DonneesIris.csv"

### Il est également possible d'importer en mémoire ces données sauvées au format csv
> Donnees.iris = read.csv("DonneesIris.csv")
### Donnees.iris est un objet de type data.frame. Les noms des variables sont toujours présents

> Donnees.iris[1,]
```

Les possibilités sont nombreuses (gestion des tabulations, des virgules, des noms variables, ...) et on se reportera à l'aide (parfois obscure malheureusement) pour plus de précisions.