Introduction à R

Programmation & Logiciels Statistiques

TD 2

Salim Lardjane - Université de Bretagne-Sud

Expressions

- Une *expression* est une suite de caractères qui a un sens pour R. Toutes les commandes valides de R sont des expressions. Lorsqu'une commande est saisie au clavier, elle est ensuite *évaluée* par R afin d'être exécutée si elle est valide. Une chaîne de caractère peut être convertie en expression à l'aide de la fonction parse.

Expressions

```
> x<-5
> expr<-parse(text="x<-7")
> expr
```

- Une expression peut être évaluée à l'aide de la fonction eval.

```
> eval(expr)
> x
```

Expressions

- On peut également définir des expressions de façon directe.

```
> expr1<-expression(x/(y+exp(z)))
> expr1
> x<-1;y<-2;z<-3
> eval(expr1)
> expr1
```

Expressions

- Il est parfois utile de construire des expressions sans les évaluer. Par exemple pour le calcul des dérivées partielles d'une fonction.

```
> D(expr1,"x")
> D(expr1,"y")
> D(expr1,"z")
```

Simulations avec R
Générateurs (pseudo)-aléatoires

- R dispose de ne nombreux générateurs de nombres pseudo-aléatoires. Il s'agit de fonctions appelées runif, rnorm, ...etc. On les assimile généralement à des générateurs de nombres aléatoires en faisant les hypothèses suivantes.

Simulation avec R

- Hypothèse 1 : Tout *appel* à un générateur de nombres pseudo-aléatoire de R associé à une loi donnée est une variable aléatoire de loi correspondante
- Hypothèse 2 : Les *appels* successifs à des générateurs de nombres pseudo-aléatoires de R sont des variables aléatoires indépendantes
- Une discussion rigoureuse de ces hypothèse sort du cadre de cette introduction.

Simulations avec R

- Sous les hypothèses précédentes, pour obtenir des réalisations de vecteurs de dimension n dont les éléments sont i.i.d. de loi *normale* centrée réduite, on peut utiliser la commande rnorm(n,mean=0,sd=1) ou, plus succinctement, rnorm(n). Pour une distribution *uniforme* sur [0,1], on utilisera la commande runif(n,min=0,max=1) ou, plus succinctement, runif(n).

Simulations avec R

```
> n <-1000
> A <- rnorm(n)
> B <- runif(n)
```

- *Tracé de l'histogramme de la série obtenue échantillon N(0,1)*

```
> nclasses<-50
> hist(A,breaks=nclasses,main="Histogramme -
        Echantillon N(0,1)")
```

Simulations avec R

- Si on désire obtenir une distribution normale de moyenne $\mu = 5$ et de variance $\sigma^2 = 4$, on peut opérer la transformation

$$A2 = A * \sigma + \mu$$

ou utiliser directement la commande

rnorm(n,mean=5,sd=2)

```
>  A2<-rnorm(1000,mean=5,sd=2);  nclasses<-50
> hist(A2,breaks=nclasses,main="Histogramme -
        Echantillon N(5,4)")
```

Simulations avec R

- *Tracé de l'histogramme de la série obtenue - échantillon U(0,1)*

```
> nclasses<-50
> hist(B,breaks=nclasses,main="Histogramme -
        Echantillon U(0,1)")
```

Simulations avec R

- Si on désire obtenir une distribution uniforme sur l'intervalle [a,b] avec a=-2 et b=3, on peut opérer la transformation

$$B2 = B * (b - a) + a$$

ou utiliser directement la commande

runif(n,min=-2,max=3)

```
> B2<-runif(1000,min=-2,max=3)
> hist(B2,breaks=50,main="Histogramme - Echantillon U(-
        2,3)")
```

Exercice

- Générer une matrice aléatoire *x* de dimensions 20*5, dont les composantes sont indépendantes de loi *N*(2,9).
- Générer une matrice *y* de taille 20*5 dont toutes les composantes sont égales à 30.
- Additionner les matrices *x* et *y*. On appellera *z* la matrice résultante.
- Calculer les moyennes, les écart-types et les variances des colonnes de *z.*
- Calculer les sommes des carrés des composantes des colonnes de *z.*

Générateurs aléatoires de R

Loi		Générateur pseudo-aléatoire R
Beta		rbeta(n,shape1,shape2)
Binomiale		rbinom(n,size,prob)
Binomiale Négative		rnbinom(n,size,prob)
Cauchy		rcauchy(n,location,scale)
Chi-Deux		rchisq(n,df)
Exponentielle		rexp(n,rate)
Fisher-Snedecor		rf(n,df1,df2)
Gamma		rgamma(n,shape,scale)
Gaussian		rnorm(n,mean,sd)
Géometrique		rgeom(n,prob)
Hypergéometrique		rhyper(nn,m,n,k)
Logistique		rlogis(n,location,scale)
Lognormale		rlnorm(n,meanlog,sdlog)
Multinomiale		rmultinom(n,size,prob)
Poisson		rpoiss(n,lambda)
Student		rt(n,df)
Uniforme		runif(n,min,max)
Weibull		rweibull(n,shape,scale)
Wilcoxon		rwilcox(nn,m,n)

Graphiques 2D

- La commande plot permet de tracer des graphiques en 2D. Avec plot(x,y), on trace y en fonction de x. x et y sont des vecteurs de *même dimension*.

Tracé de courbe

Définition des valeurs de x et calcul des valeurs de y=f(x)

```
> x<-seq(-pi,pi,0.1)
> y<-sin(x)
```

Graphiques 2D

Tracé de la fonction

```
> plot(x,y,type="l")
```

- L'option type="l" permet de spécifier que le graphique doit être une courbe. D'autres possibilités sont type="p" (points), type="h" (traits verticaux), type="s" (escaliers) et type="n" (ne rien tracer). Les essayer.

Documentation du graphique

```
> plot(x,y,type="l")
> grid()
> title(main="y = sin(x)",xlab="x",ylab="y")
> text(0,0,"(0,0)")
```

Impression d'un graphique

- Afin d'imprimer un graphique dans un fichier ou sur papier, on peut utiliser les commandes pdf, postscript, jpeg,...etc.

```
> jpeg(file="sin.jpg") #ouvre un fichier jpeg
> plot(x,y,type="l")
> grid()
> title(main="y = sin(x)",xlab="x",ylab="y")
> text(0,0,"(0,0)")
> dev.off() #ferme le fichier jpeg
```

Impression d’un graphique

- Lire l'image à l'aide de Mozilla Firefox, la redimensionner à l'aide de GIMP ou PAINT.

Fenêtres graphiques

- Par défaut plot ouvre une fenêtre graphique de type windows.

```
> windows() # ouvre une nouvelle fenêtre windows
> plot(x,y,type="l")
> grid()
> title(main="y = sin(x)",xlab="x",ylab="y")
> text(0,0,"(0,0)")
> dev.off() # ferme la fenêtre windows
```

Fenêtres graphiques

- *Options de windows()*
  Les dimensions de la fenêtre graphiques peuvent être spécifiés (en pixels) à l'aide des options height (hauteur) et width (largeur) .

```
> windows()
> windows(height=500,width=400)
> windows(height=200,width=100)
```

Autres options de plot

- col (couleur) peut prendre les valeurs "green", "blue", "red", "black", ...etc.
- La liste des couleurs ainsi disponibles peut être obtenues en faisant

```
> colors()
```

Autres options de plot

- Pour les tracés continus (type="l"), lty (type de courbe) peut prendre les valeurs "solid" (trait plein), "dashed" (tirets), "dotted" (points), "dotdash" (point-tiret), "longdash" (longs tirets), "twodash" (tiret long - tiret court)

Autres options de plot

- Pour les tracés discontinus (type="p"), pch (symbole de point) peut prendre les valeurs "+", "o", ".", "*",...etc, ainsi que les valeurs 0 à 25 (formes géométriques).

Options de plot

```
> plot(x,y,type="l",col="green",lty="solid")
> windows()
> plot(x,y,type="p",col="blue",pch="+")
> windows()
> plot(x,y,type="l",col="red",lty="dotdash")
```

Fermer les fenêtres graphiques

- On utilise dev.off() pour fermer la fenêtre graphique ou le fichier graphique en cours d'utilisation. Pour fermer toutes les fenêtres graphiques ou fichiers graphiques, on utilise graphics.off().

```
> graphics.off()
```

Nuage de points

- Le tracé d'un nuage de points se fait à l'aide de la commande plot(...,type="p").

Exemple : nuage de points aléatoires

```
> N<-100
> x<-runif(N)
> y<-runif(N)
> plot(x,y,type="p",pch="o")
> grid()
> title(main="Nuage de points aléatoires")
```

Exercice

- Répéter l'exemple avec N = 10 000
- Répéter l'exemple avec rnorm au lieu de runif

Diagramme en barres

- La commande barplot(y) dessine un graphique en barres des valeurs de y.

```
> x<-seq(-2*pi,2*pi,pi/10)
> y<-cos(x)
> barplot(y)
> grid()
> title(''Graphique en barres des valeurs d'une fonction'')
```

- Pour plus d'information sur barplot (options) faire

```
> ?barplot
```

Diagramme circulaire

- pie(x) : Tacé du diagramme circulaire en 2D correspondant au vecteur ligne (distribution de fréquences). Les éléments du vecteur sont normalisés par x/sum(x).

```
> x<-c(0.25,0.40,0.35)
> pie(x)
> windows()
> pie(x,labels=c("x1","x2","x3"))
> title(main="Diagramme circulaire")
```

- Pour plus d'information sur pie (options) faire

```
> ?pie
```

Histogramme

- hist(y,breaks=N) : trace l'histogramme de la série des éléments de y pour N classes de même amplitude

```
> y<-rnorm(1000)
> hist(y,breaks=10,main="Histogramme d'une série normale (10 classes)")
> grid()
```

Histogramme

- Au lieu de spécifier directement le nombre de classes, on peut passer en argument un vecteur qui spécifie les centres des classes.

```
> x<-seq(-5,5,0.2)
> hist(y,breaks=x,main="Histogramme à intervalles et espacements fixés")
> grid()
```

Histogramme

- Pour un histogramme en fréquences *relatives*, utiliser l'option freq=FALSE.

```
> windows()
> hist(y,breaks=x,main="Histogramme à intervalle et espacements fixés",freq=FALSE)
> grid()
```

Histogramme

- On peut surimposer la courbe de la densité normale à l'aide de la fonction curve.

```
> curve(exp(-x^2/2)/sqrt(2*pi),from=-5, to =5,add=TRUE)
```

- La fonction curve permet plus généralement de tracer une courbe de fonction en spécifiant celle-ci soit par une expression en x, soit par son nom.

```
> windows()
> curve(sin,from=-2*pi, to =2*pi)
```

Nuages de points croisés

- pairs(x) : Tracé de nuages de points entre les colonnes de la matrice .

```
> x<-matrix(rnorm(150),nrow=50)
> y<-matrix(runif(150),nrow=50)
> pairs(x)
> x[,2]<-3*x[,1]+y[,1]
> pairs(x)
```

- Pour plus d'information sur pairs (options) faire

```
> ?pairs
```

Graphiques 2D

- Autres commandes graphiques 2D : boxplot, dotchart, qqplot, cdplot, stem, stars...etc.

Graphiques 3D

- On souhaite représenter graphiquement la surface d'équation

$$z = \frac{\sin(x^2+y^2)}{x^2+y^2}$$

pour x et y variant de -pi à pi avec un pas de pi/10 .

Graphiques 3D

- *Définition de la fonction à représenter*

```
> f<-function(x,y)
{
        ans<- sin(x^2+y^2)/(x^2+y^2)
        return(ans)
}
```

Graphiques 3D

- Les vecteurs x et y définissent le domaine de calcul de z.
- *Evaluation des valeurs de x, y, z*

```
> x<- seq(-pi,pi,pi/10)
> y<-x
> z<-outer(x,y,f)
> z[is.nan(z)]<-1  #(traite le cas x=y=0)
```

Graphiques 3D

- *Tracé de la fonction*

```
> persp(x,y,z,theta=30,phi=30)
```

Documentation du graphique

```
> persp(x,y,z,theta=30,phi=30,main="Tracé d'une surface",
xlab="x",ylab="y",zlab="z")
```

- *Avec davantage d'options*

```
> persp(x,y,z,theta=30,phi=30,expand=0.5,col="lightblue",
        ltheta=120,shade=0.75,ticktype="detailed",
        main="Tracé d'une surface",
        xlab="x",ylab="y",zlab="z")
```

- Pour plus d'information sur persp (options) faire

```
> ?persp
```

Graphiques 3D

- Autres commandes graphiques en 3D : image, contour, ...
- Pour avoir un aperçu des autres possibilités graphiques de R, saisir les commandes

```
> demo(persp)
> demo(image)
> demo(graphics)
```

Partitionnement d'une fenêtre graphique

- Il arrive fréquemment qu'on souhaite afficher plusieurs graphiques dans une même fenêtre. Ceci peut être fait de plusieurs façons avec R. On peut en effet utiliser les fonctions split.screen, layout ou par.

par

- Un partitionnement rectangulaire de la fenêtre graphique peut facilement être réalisé à l'aide des options mfrow et mfcol de la fonction par.

```
> x<-rnorm(150)
> y<-runif(150)
> windows()
> par(mfrow=c(2,2))
> plot(x,y)
> hist(x)
> hist(y)
> plot(y,x)
```

par

```
> windows()
> par(mfcol=c(2,2))
> plot(x,y)
> hist(x)
> hist(y)
> plot(y,x)
```

split.screen

- Pour des arrangements plus complexes que des arrangements rectangulaires, on peut utiliser split.screen. Cette fonction n'est toutefois pas compatible avec toutes les fonctions graphiques de R.

```
> x<-rnorm(150)
> y<-runif(150)
> windows()
> split.screen(c(1,2))
> screen(1)
> hist(x)
> close.screen(1)
```

split.screen

```
> screen(2)
> hist(y)
> close.screen(2)
> windows()
> split.screen(c(2,1))
> screen(1)
> hist(x)
> close.screen(1)
> screen(2)
> hist(y)
> close.screen(2)
```

split.screen

- Une partie obtenue par split.screen peut également être divisée à l'aide de split.screen.

```
> windows()
> split.screen(c(2,1))
> screen(1)
> plot(x,y)
> close.screen(1)
> screen(2)
> split.screen(c(1,2))
> close.screen(2)
```

split.screen

```
> screen(3)
> hist(x)
> close.screen(3)
> screen(4)
> hist(y)
> close.screen(4)
```

- Il est recommandé, lorsqu'on utilise split.screen de s'occuper de chaque graphique de façon complète avant de passer au suivant.

Layout

- La fonction layout partitionne la fenêtre graphique en plusieurs sous-fenêtres sur lesquelles sont affichés les graphiques successivement. Elle permet d'obtenir des arrangements complexes.

```
> windows()
> m<-matrix(1:4,nrow=2,ncol=2)
> m
> layout(m)
```

Layout

- Un arrangement spécifié dans layout peut être visualisé à l'aide de la fonction layout.show qui prend en argument le nombre de sous-fenêtres.

```
> layout.show(n=4)
> plot(x,y)
> hist(x)
> hist(y)
> plot(y,x)
```

Layout

- Les exemples qui suivent illustrent quelques uns des arrangements susceptibles d'être obtenus à l'aide de layout.

```
> m<-matrix(1:6,nrow=3,ncol=2)
> layout(m)
> layout.show(n=6)
> m<-matrix(1:6,nrow=2,ncol=3)
> layout(m)
> layout.show(n=6)
```

Layout

```
> m<-matrix(c(1:3,3),nrow=2,ncol=2)
> layout(m)
> layout.show(n=3)
> m<-matrix(c(1:3,3),nrow=2,ncol=2,byrow=TRUE)
> layout(m)
> layout.show(n=3)
> m<-
matrix(c(2,1,4,3),nrow=2,ncol=2,byrow=TRUE)
> layout(m)
> layout.show(n=4)
```

Layout

```
> m<-matrix(scan(n=25),nrow=5,ncol=5)
 1:0
 2:0
 3:3
 4:3
 5:3
 6:1
 7:1
 8:3
 9:3
 10:3
 11:0
 12:0
 13:3
 14:3
```

Layout

```
 15:3
 16:0
 17:2
 18:2
 19:0
 20:5
 21:4
 22:2
 23:2
 24:0
 25:5
> layout(m)
> layout.show(n=5)
```

Layout

- Par défaut, layout partitionne la fenêtre graphiques de façon proportionnelle. On peut spécifier des dimensions relatives pour les différentes sous-fenêtres à l'aide des options widths et heights.

```
> m<-matrix(1:4,nrow=2,ncol=2)
> layout(m,widths=c(1,3),heights=c(3,1))
> layout.show(n=4)
> m<-matrix(c(1,1,2,1),nrow=2,ncol=2)
> layout(m,widths=c(2,1),heights=c(1,2))
> layout.show(n=2)
```

Layout

```
> m<-matrix(0:3,nrow=2,ncol=2)
> layout(m,widths=c(1,3),heights=c(1,3))
> layout.show(n=3)
```

Objets

- Les vecteurs, matrices, fonctions et expressions vues jusqu'ici sont traitées par R commes des *objets*. Un objet R est caractérisé par son nom et son contenu mais également par ses *attributs*. On distingue des *attributs intrinsèques* : *type*, *mode* et *length* (longueur), liés à la façon dont R stocke l'objet sur machine, et des éventuels *attributs extrinsèques* : *class, comment, dim, dimnames, names, row.names, rownames, colnames, tsp* et *levels* qui vont déterminer la façon dont les fonctions agissent sur l'objet.

Vecteurs

- Le *type* d'un objet détermine la façon dont R le stocke sur machine. Le *mode* d'un objet est fonction de son type et en donne une version moins détaillée. Ainsi, pour un vecteur, les types integer et double seront tous deux résumés par le mode numeric.

```
> x
> typeof(x)
> mode(x)
> x<-x+0.5
> x
> typeof(x)
> mode(x)
```

Vecteurs

- Un vecteur logique est de type et de mode logical. Il existe également un type et mode complex, mais il ne sera pas discuté ici.

```
> t<-(x<5)
> typeof(t)
> mode(t)
```

Vecteurs

- L'attribut *class* d'un objet détermine la façon dont il est traité par différentes fonction de R. La fonction class permet d'obtenir la classe d'un objet. Pour un vecteur, la classe est implicitement identique au mode mais peut être modifiée par l'utilisateur.

```
> x
> class(x)
> class(x)<-"a"
> x
> class(x)<-"numeric"
> x
```

Vecteurs

- L'attribut *length* peut être utilisé pour obtenir ou modifier la longueur d'un vecteur.

```
> x
> length(x)
> length(x)<-3
> x
> length(x)<-5
> x
```

Vecteurs

- L'attribut *names* peut être utilisé pour affecter des labels ("noms") aux différentes composantes d'un vecteur.

```
> names(x)
> names(x)<-c("A","B","C","D","E")
> x
> names(x)
```

Extraction d'un élément d'un vecteur à l'aide de son label

```
> x["B"]
> x[c("A","B")]
> x[c("F","A","B")]
```

Vecteurs

- L'attribut *comment* peut être utilisé pour associer un commentaire à un vecteur.

```
> comment(x)<-c("Un exemple simple", "24.09.2011")
> comment(x)
> x
```

Matrices

```
> y <- matrix(1:9,nrow=3,ncol=3,byrow=TRUE)
> y
> typeof(y)
> mode(y)
> length(y)
> class(y)
```

Matrices

```
> z <- matrix(seq(from=1,to=2,length=9),nrow=3,ncol=3,byrow=TRUE)
> z
> typeof(z)
> mode(z)
> length(z)
> class(z)
```

Matrices

```
> m <- (z<1.5)
> m
> typeof(m)
> mode(m)
> length(m)
> class(m)
```

Matrices

- Ainsi qu'on l'a déjà vu, l'attribut dim correspond aux dimensions d'une matrice. Il peut être obtenu ou modifié à l'aide de la fonction dim. La longueur d'une matrice correspond à la somme de ses dimensions.

```
> y <- matrix(1:6,nrow=2,ncol=3,byrow=TRUE)
> y
> dim(y)
> dim(y)<-c(3,2)
> y
```

Matrices

- L'attribut rownames permet d'affecter des labels ("noms") aux lignes d'une matrice. La fonction rownames correspondante permet d'obtenir la valeur de cet attribut ou d'en modifier la valeur.

```
> rownames(y)
> rownames(y)<-c("a","b","c")
> y
```

Matrices

- *Extraction d'une ligne d'une matrice à l'aide de son label*

```
> y["a",]
```

Matrices

- L'attribut colnames permet d'affecter des labels ("noms") aux colonnes d'une matrice. La fonction colnames correspondante permet d'obtenir la valeur de cet attribut ou d'en modifier la valeur.

```
> colnames(y)
> colnames(y)<-c("x1","x2")
> y
```

Matrices

- *Extraction d'une colonne d'une matrice à l'aide de son label*

```
> y[,"x1"]
```

- *Extraction d'un élément d'une matrice à l'aide des labels de ligne et de colonne*

```
> y["a","x1"]
```

Matrices

- De même que pour les vecteurs, l'attribut *comment* peut être utilisé pour associer un commentaire à une matrice.

```
> comment(y)<-c("Un exemple simple de matrice","24.09.2011")
> comment(y)
> y
```

Fonctions

- Les fonctions R sont de mode function. Celui-ci correspond à trois types : special, builtin et closure, les deux premiers types correspondant aux fonctions et opérateurs de base disponibles sous R.

```
> f <- function(x,y) {return(x+y-2)}
> f
> typeof(f)
> mode(f)
```

Fonctions

- La longueur ("length") d'une fonction est toujours égale à 1. Sa classe est par défaut identique à son mode. Il est possible d'associer un commentaire à une fonction à l'aide de l'attribut comment.

```
> length(f)
> class(f)
> comment(f)<- c("Un exemple simple de fonction", "29.09.2011")
> comment(f)
> typeof(sin)
> mode(sin)
> length(sin)
> class(sin)
```

Expressions

- Les expressions R sont de type, mode et par défaut de classe expression. Leur longueur est fixée à 1. De même que les vecteurs, matrices et fonction, il est possible d'associer un commentaire à une expression à l'aide de l'attribut comment.

```
> expr <- expression(x/(y+exp(z)))
> expr
> typeof(expr)
> mode(expr)
> length(expr)
> class(expr)
> comment(expr)<- c("Un exemple d'expression R", "29.11.2011")
> comment(expr)
```

Sauvegarde d'objet

```
> x<-c(1,2,3)
> save(x,file="fichier.Rdata")
```

- Le fichier obtenu aura pour extension .Rdata, ce qui permettra de la reconnaître comme fichier de données R, et sera sauvegardé dans le répertoire de travail sous le nom fichier.Rdata.

Sauvegarde d'objet

- On peut effacer toutes les objets de la mémoire :

```
> rm(list=ls())
> x
```

- Si l'on charge le fichier fichier.Rdata, l'objet est de nouveau présent dans l'espace de travail :

```
> load("fichier.Rdata")
> x
```

Sauvegarde d'objet

- Pour afficher (presque) *tous* les objets en mémoire, on peut utiliser les instructions ls(), objects()

```
> x<-c(1,2,3)
> y<-1
> z<-2
> ls()
> objects()
```

Sauvegarde d'objet

- Pour effacer les variables x et y :

```
> rm(x,y)
> ls()
```

Contenu d'un objet

- Pour afficher un résumé succinct du contenu d'un objet R : str

```
> x<-c(1,2,3)
> y<-1
> f <- function(x,y) {return(x+y-2)}
> expr <- expression(x/(y+exp(z)))
> str(x)
> str(y)
> str(f)
> str(expr)
```

Contenu d’un objet

- Pour afficher un résumé succinct de tous les objets présents dans l'espace de travail :

```
> ls.str()
```

Fonctions génériques

- Certaines fonctions R agissent différemment sur un objet en fonction de sa classe. De telles fonctions sont dites *génériques*. Elles ne font rien en elles-mêmes si ce n'est appeler la fonction (appelée *méthode*) adaptée à la classe de l'objet en question parmi un ensemble de fonctions associées.

Exemple : summary

```
> x<-c(1,2,3)
> y<-matrix(1:6,nrow=2)
> expr <- expression(x/(y+exp(z)))
> summary(x)
> summary(y)
> summary(expr)
```

Fonctions génériques

- Pour afficher l'ensemble des méthodes associées à une fonction générique : methods

```
> methods("summary")
```

- On voit ainsi les différentes classes des objets qu'on peut résumer à l'aide de summary.

- Deux autres exemples de fonctions génériques : print et plot. Afficher les méthodes correspondantes.

Fichiers et programmation

- R peut exécuter des séquences d'instructions stockées dans des fichier. Ces fichiers sont appelés scripts. Leur extension est .R.
- Un script est une séquence d'instructions R.
- Les variables d'un fichier script sont généralement globales (à moins qu'elles apparaîssent dans une définition de fonction).
- Les valeurs des variables de l'espace de travail peuvent être modifiées par un fichier script.

Fichiers et programmation

- Une première utilisation des fichiers scripts est la *lecture et mise en forme de données*.
- Editeur de scripts : menu Files/New Script.
-
- Sauvegarder le script

```
A<-matrix(1:6,nrow=2)
```

sous le nom donnees.R puis l'executer par Ctrl a/Ctrl r ou en utilisant le menu Edit/Run All.

Fichiers et programmation

- Examiner les variables présentes dans l'environnement de travail.

```
> ls()
```

- Au lieu d'éxécuter le script en le sélectionnant, on peut utiliser la fonction source.

```
> rm(A)
> source("donnees.R")
> ls()
```

Fichiers et programmation

- *Script courbe1.R*

```
x<-seq(-5,5,0.1)
y<-x^2+5
plot(x,y,type="l")
grid()
title(main="Tracé de y=x^2
        +5",xlab="x",ylab="y")

> source("courbe1.R")
```

Fichiers et programmation

- Pour insérer une ligne de commentaire dans un programme, utiliser le symbole #.

- Les scripts peuvent être utilisés pour définir des fonctions R.

- *Exemple*

Nous allons écrire une fonction générant un tableau de nombres aléatoires entiers compris entre 0 et une valeur maximale contenue dans une variable notée max.

Fichiers et programmation

- *Fichier de fonction randint.R*

```
randint <- function(n,max)
# res : vecteur de n entiers compris entre 0 et max
# runif : génère un nombre aléatoire entre 0 et 1
# floor : renvoie la partie entière d'un nombre
{       temp<-runif(n)
        res<-floor((max+1)*temp)
        return(res)
}
```

Fichiers et programmation

- Il est préférable de donner à ce type de fichier un nom identique à celui de la fonction. Ainsi, l'exemple sera sauvegardé sous le nom randint.R.
- La première ligne déclare le nom de la fonction et les arguments d'entrée.
- Pour appeler une fonction, il suffit d'exécuter le script correspondant puis de procéder selon la syntaxe suivante :

```
resultat <- nom_fonction(liste des arguments
        d'appel)
```

Fichiers et programmation

- *Exemple*

- > nb_alea<-randint(10,50)
- > nb_alea

- On peut également appeler la fonction sous la forme

> randint(n=10,max=50)

Fichiers et programmation

- Il est souhaitable de faire suivre la première ligne d'un fichier de fonction par des lignes de commentaire dans lesquels on décrit son but et ses arguments.
- La commande source permet également à un fichier de fonction ou un script quelconque de charger une fonction R.

Fichiers et programmation

- *Exemple*

```
> rm(list=ls())
> randint(10,50)
> source("randint.R")
> randint(10,50)
```

Instructions de contrôle

- R dispose des instructions de contrôle : for, while, if, else, repeat, break, switch

if, else

- *Exemple : Script flog.R*

```
 flog <- function(x)
{
 # Une fonction simple
 if (x<0) return(x) else return(log(x))
 }
> flog(2)
> flog(-2)
```

for

- *Exemple : Script ncarres.R*

```
 ncarres <- function()
{
 # tableau des carrés des 10 premiers entiers naturels
 n <- 10
 x <- NULL # vecteur vide
 for (i in 1:n) {x<-c(x,i^2)}
 cat(x," \n") # Affiche le résultat obtenu
 }
```

Ctrl a/Ctrl r

```
> ncarres()
```

for

- Des boucles for peuvent notamment être utilisées lors de la création d'objets R. Dans ce cas l'objet doit être créé en dehors de la boucle.

```
> x<-NULL
> for (i in 1:10) x[i]<-i
> x
> y<-matrix(0,nrow=3,ncol=4)
> for (i in 3) { for (j in 1:4) { y[i,j]<-i+j}}
> y
```