Timothée Poisot

Introduction à R

Par la pratique

UQAR — Mai 2012

Table des matières

1 Introduction

1.1 Objectifs du cours

1.2 Environnement de travail

Installation de R

`http://www.r-project.org/`

Installation de RStudio

RStudio est un environnement de travail pour R grauit, multi-plateforme, disponible en ligne à `http://rstudio.org/`. La première partie de la séance sera consacrée à l'utilisation de RStudio.

1.3 Types de données

Types de valeurs

Numériques

Chaînes

Booléens

Collections de valeurs

Vecteurs et vectorisation

Le vecteur est, avec la matrice, l'objet le plus important de R. R est un langage dit *vectorisé*, c'est-à-dire qui peut traîter plusieurs valeurs regroupées dans un objet unique. Si on utilise une commande très simple, comme

```
> 2
[1] 2
```

on remarque que la sortie est `[1] 2`. L'indicateur `[1]` indique que la valeur retournée est le premier élément d'un vecteur. La puissance de la notation vectorielle est qu'on peut accéder à une partie du vecteur, avec un *indice*. Si on prend l'exemple suivant,

```
> a = 2
> a[1]
[1] 2
> a[2]
[1] NA
```

, accéder à la position 1 *via* l'*indice* `[1]`, on récupère la première valeur du vecteur `a`. Voilà une des particularités de R : tout objet est un vecteur ! Accéder à la position `[2]` retourne `NA`, parce que le vecteur `a` ne possède pas de deuxième position.

On peut créer des vecteurs dans R en utilisant la commande `c`.

```
> vecteur = c(1,2,3,4,5)
```

R propose différents raccourcis pour créer rapidement des vecteurs. Par exemple, examinez le comportement des commandes suivantes :

```
> seq(from=0,to=5,by=1)
[1] 0 1 2 3 4 5
> seq(from=0,to=10,length=3)
[1]  0  5 10
> c(0:5)
[1] 0 1 2 3 4 5
```

L'avantage de la vectorisation est que R va automatiser une grande partie des opérations sur les vecteurs. Par exemple, examinez l'effet des commandes suivantes :

```
> a = c(1:10)
> a/2
 [1] 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
> log(a,10)
 [1] 0.0000000 0.3010300 0.4771213 0.6020600 0.6989700
 [6] 0.7781513 0.8450980 0.9030900 0.9542425 1.0000000
> a*a
 [1]   1   4   9  16  25  36  49  64  81 100
```

Avec les vecteurs vient le concept important de *recyclage*. Le recyclage consiste a répéter un vecteur autant de fois que nécessaire pour le rendre compatible avec un autre vecteur dans le cadre d'une opération. Par exemple, les commandes

```
> c(1,2,3,4,5) + c(1,2)
[1] 2 4 4 6 6
```

```
> c(1,2,3,4,5) + c(1,2,1,2,1)
[1] 2 4 4 6 6
```

sont équivalentes.

Matrices

Listes

2 Lecture et écriture des données

3 Opérations sur les tables de données

3.1 Travail sur les lignes et colonnes

Dans une grande variété de situations, il peut être avantageux de répéter une opération sur toutes les lignes, ou toutes les colonnes. R propose une fonction pour automatiser ce traitement, *via* la fonction `apply`.

```
> dat = matrix(rnorm(100),nrow=10)
> apply(dat,1,mean)
 [1]  0.24792467  0.21601310  0.65499007  0.11655810
 [5]  0.01708206 -0.32615953  0.43052493  0.39561040
 [9]  0.18597926 -0.04952184
> apply(dat,2,var)
 [1] 0.5646966 0.5820893 0.3217383 0.8751605 0.5774498
 [6] 0.8093642 3.3105245 1.3007209 0.7901021 0.7952149
```

3.2 Traitement des données

3.3 Division et traitement par niveau

4 *Introduction à la programmation*

4.1 Programmer, pourquoi ?

Dans les séances précédentes, nous avons utilisé des fichiers `.R` pour sauvegarder des listes d'instructions. Nous avons aussi chargé et manipulé des jeux de données. Il est souvent nécessaire d'automatiser tout ou partie de ce processus, ce qui implique de faire appel à de la programmation.

L'objectif de cette séance est de

4.2 Boucles

Les boucles permettent parcourir une liste, ou de répéter une série d'instructions, dans des conditions bien définies ; c'est une des structures de base de l'algorithmique. R propose deux types de boucles, les boucles `for` et les boucles `while`. En français, on peut les résumer par «pour chaque» et «tant que».

Boucles de type *for*

Une boucle `for` permet de répéter un bloc d'instructions un nombre prédéfini de fois, ou d'éxécuter des commandes pour chaque élément d'un tableau de données. La syntaxe de base est la suivante :

```
> for (step in c(1:10)) cat(step)
12345678910
```

En clair, pour chaque valeur entre 1 et 10, qu'on nomme `step` (mais qu'on aurait pû nommer n'importe comment), on affiche (`cat`) la valeur de step. On peut bien sûr spécifier plusieurs instructions qui doivent être éxécutées à chaque *itération* (étapes de la boucle) en utilisant les accolades :

```
> for (step in c(1:3))
  {
          cat(step)
          print(summary(rnorm(100,mean=step)))
  }
```

```
1   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-1.5820  0.4398  1.0350  1.1320  1.8640  3.7730
2   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 -1.330   1.436   2.143   2.080   2.858   4.072
3    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-0.02333  2.46800  2.82900  2.96500  3.60300  6.20700
```

Cette commande affiche le numéro de l'itération en cours (`cat(step)`), puis affiche les information de base (`summary`) sur une distribution normale (`rnorm`) centrée sur `step`. Les boucles `for` peuvent contenir des instructions aussi longues que souhaité.

Une autre application des boucles `for` est de parcourir un objet. Par exemple, on peut souhaiter, pour chaque élément d'un objet, afficher sa valeur. R permet de réaliser ce genre d'opérations, avec la syntaxe suivante :

```
> vect =  c(a,b,c,d)
> for (val in vect) cat(val)
abcd
```

Pour chaque élément du vecteur `vect`, que l'on nomme `val` pour pouvoir y accéder pendant les itérations, R va afficher la valeur que l'élément contient.

Les boucles `for` sont donc une façon explicite de faire la même chose que ce qui a été traité dans la séance précédente, avec un inconvénient majeur : elles sont extrèmement demandantes en temps de calcul. Dans la majorité des cas, il est préférable d'avoir recours autant que possible aux fonctions `*apply`, qui sont bien plus optimisées que des boucles.

Boucles de type *while*

Les boucles de type `while`, littéralement *pendant*, permettent de répéter une série d'instructions tant qu'une condition n'a pas été atteinte. Pour cette raison, il faut bien prendre en compte le fait que mal utilisées, ces boucles peuvent ne jamais stopper. Il faut donc faire particulièrement attention à la condition qui est évaluée à chaque itération.

Un exemple simple d'utilisation d'une boucle `while` est le calcul d'une factorielle. On veut calculer $n!$, ce qui se fait simplement en multipliant l'ensemble des $1 \leq k \leq n$.

```
> n = 5
> k = n
> while(k > 1){
          k = k - 1
          n = n*k
  }
> print(n)
```

```
[1] 120
```

On remarquera que dans la parenthèse après `while` se trouve un test logique ; les tests sont abordés dans la partie suivante.

Sortir d'une boucle et sauter des étapes

Lors de l'éxécution d'une boucle, on peut ne pas vouloir éxécuter toutes les instructions. R possède des structures de contrôle pour effectuer ce type d'opérations. Les principales qu'il faut connaître sont `break` et `next`. L'instruction break permet de stopper l'éxécution de la boucle, c'est-à-dire de sortir de la boucle comme si la condition de sortie était remplie, ou le nombre maximal d'itérations atteint. La boucle suivante va afficher les valeurs de `i` tant qu'elles sont inférieures à 3, et afficher `bye!` puis stopper la boucle sinon.

```
> for(i in c(1:5)){
          if(i < 3){
                  cat(i)
          } else {
                  cat( bye!)
                  break
          }
  }
12 bye!
```

Une autre structure de contrôle intéréssant est next, qui permet de sauter une itération si la valeur de l'itérateur ne nous plaît pas. Cette structure est particulièrement utile quand on réalise un nombre important d'opérations à chaque itération, et qu'on ne veut pas perdre de temps a traiter des valeurs qui ne nous intéréssent pas. On peut utiliser `next` pour avoir, par exemple, une liste de tous les nombres pairs entre 1 et 10 :

```
> is.even <- function(x) x %% 2 == 0
> for(i in c(1:10)){
          if(is.even(i)){
                  cat(i)
                  cat( is even\n)
          } else {
                  next
          }
  }
2 is even
4 is even
```

```
6 is even
8 is even
10 is even
```

4.3 Tests

Cette partie est consacrée à l'utilisation des tests. On a vu dans les séances précédentes l'existence de variables de type booléen, qui prennent les valeurs TRUE ou FALSE. Des variables de ce type sont utilisées dans le cadre d'expressions conditionelles, c'est-à-dire quand on souhaite effectuer différentes instructions en fonction de la valeur d'une condition.

Expressions conditionnelles

La structure de base d'une expression conditionnelle est la suivante :

```
> if (condition){
          instruction(1)
  } else {
          instruction(2)
  }
```

Une notation comme

```
> if (condition) instruction
```

est aussi acceptable, de même que

```
> val = ifelse(condition,valeur_if,valeur_else)
```

Cette dernière notation permet de gagner du temps quand on veut que la valeur d'une variable dépende d'une condition. Par exemple, $(n \geq 2) \vee (n = 0)$ s'écrit `(n <= 2) or (n == 0)`, ou encore `(n <= 2) | (n == 0)`. Les différents opérateurs logiques sont regroupés dans le tableau 4.1.

L'argument `condition` prend la forme d'un test logique, qui peut être d'une complexité aussi grande que l'on veut. Pour certains des opérateurs, il existe deux variantes (| et ||, & et &&). On peut comprendre pourquoi avec les exemples suivants :

```
> a = c(1,2,3,4,5,6)
> b = c(1,2,3,4,4,6)
> d = c(3,2,3,4,5,4)
> (a == b)
[1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE
```

Opérateur	Signification	Version 1	Version 2
$\vee$	ou (au moins une des deux)	`or`	`|`, `||`
$\wedge$	et (les deux)	`and`	`&`, `&&`
$\neg$	non	`not`	`!`
$\oplus$	ou conditionnel (seulement une des deux)	`xor(a, b)`	
$\in$	est compris dans	`%in%`	
$=$	égalité		`==`
$\leq$	inférieur ou égal		`<=`
$\geq$	supérieur ou égal		`>=`
$>$	supérieur		`>`
$<$	inférieur		`<`

TABLE 4.1: Différents opérateurs logiques disponibbles dans R.

```
> (a == d)
[1] FALSE  TRUE  TRUE  TRUE  TRUE FALSE
> (b == d)
[1] FALSE  TRUE  TRUE  TRUE FALSE FALSE
> (a == d) & (a == b)
[1] FALSE  TRUE  TRUE  TRUE FALSE FALSE
> (a == d) && (a == b)
[1] FALSE
> (a == d) | (a == b)
[1] TRUE TRUE TRUE TRUE TRUE TRUE
> (a == d) || (a == b)
[1] TRUE
```

L'opérateur répété une seule fois est *elemt-wise* : les éléments des vecteurs sont comparés deux à deux (et on applique du recyclage). L'opérateur répété deux fois

Manipulation des valeurs booléennes

addition, multiplication, ...

4.4 Fonctions

Généralités

L'utilisation des fonctions va permettre de gagner du temps dans la programmation. Comprendre le principe des fonctions dépasse de beaucoup le cadre de R, et mérite qu'on s'y arrête. Qu'est-ce qu'une fonction ? Une série d'instructions qui vont, à partir d'arguments, renvoyer un résultat. En quoi est-ce différent des scripts que nous avons utilisé jusqu'ici ? Écrire une fonction revient en quelque sorte a 'expliquer' le code une fois, et R se charge ensuite de redonner la bonne valeur aux arguments.

Le parallèle le plus évident est celui des fonctions mathématiques : si $f(x) = x^2$, on peut calculer $f(x)$ pour tout x, parce qu'on sait quoi faire. Ça devient donc très avantageux si on doit calculer $f(x)$ un grand nombre de fois. En écrivant uniquement des scripts, pour calculer la valeur de beaucoup de x^2, on aurait écrit :

```
> 1^2
> 2^2
> 3^2
> 4^2
> 5^2
```

ou encore

```
> for(i in c(1:5)) i^2
```

Si il faut revenir sur ce code plus tard, et transformer tous les `^2` en `^3`, la première solution implique de tout corriger manuellement. En utilisant une fonction, la logique est différente :

```
> f = function(x) x^2
> f(2)
[1] 4
```

Peut importe le nombre de fois ou on devra effectuer l'opération contenue dans `f`, si on veut la modifier, elle sera toujours stockée au même endroit. Avec cet avantage en tête, quelques généralités sur les fonctions en R.

Tout est fonction

Dans la pratique, la majorité des instructions utilisées jusau'à présent sont des fonctions. Par exemple, `print`, `read.table`, et `apply` sont des fonctions rendues disponibles par R.

Le *scope*

Les fonctions existent comme un espace «à part» dans R : ce qui se déroule dans une fonction, reste dans une fonction. Prennons le cas du code suivant :

```
> a = 2
> b = 3
> f = function(x){
          y = x+a
          z = y
          return(z)
  }
> f(b)
[1] 5
```

La fonction `f` peut aller chercher la valeur de `a` dans l'environnement global, mais tout ce qui est défini au sein de `f` est innaccessible. D'ailleurs, tout ce qui est créé dans la fonction est détruit – retiré de la mémoire – une fois que la dernière instruction est éxécutée. Voyez la section sur la fonction `return` pour plus de détails. Cette notion de quel objet est accessible est extrèmement importante à maîtriser.

Dans R, les objets existent dans deux «mondes», l'environnement global, et l'intérieur de chaque fonction. L'intérieur de chaque fonction peut avoir accès aux objets et aux variables de l'environnement global, même si cette pratique est à éviter pour différentes raisons (une variable globale peut être modifiée entre deux appels à la fonction, notamment). En revanche, l'environnement global n'a pas accés aux objets crées ou modifiés à l'intérieur d'une fonction. La communication avec les fonctions se fait ...

Le fait qu'on puisse passer des objets d'un environnement à l'autre peut entraîner un comportement assez surprenant. Dans l'exemple suivant :

```
> a = 2
> print(a)
[1] 2
> f = function(a)
  {
          a = a+1
          return(a)
  }
> print(f(a))
[1] 3
> print(a)
[1] 2
```

l'appel à la fonction `f` devrait modifier `a`, puisque la seule instruction de cette fonction est a = a+1. Or, quand on appelle cette fonction, puis qu'on affiche la valeur de `a`, elle n'a pas changé. Ce comportement vient du fait que l'objet `a` qui existe dans la fonction n'est pas celui qui existe dans l'environnement global. Par conséquent, le `a` de la fonction peut être modifiè, sans que cela n'affecte le `a` de l'environnement de travail.

Cet example illustre aussi pourquoi le nom des variables est important. On sait que R est capable d'aller chercher, depuis une fonction, des variables de l'environnement global. Quelle version de `a` faut-il aller chercher ? Pour éviter les erreurs liées au fait que plusieurs variables aient le même nom, on essaie de donner un nom unique à toutes les variables. C'est sans doute plus long à écrire – même si cet argument n'est pas valable avec un éditeur qui auto-complète le code –, mais ça évite surtout les erreurs à l'éxécution.

Déclarer une fonction

Comme illustré dans les exemples précédents, la déclaration d'une fonction se fait par

```
> nom_de_la_fonction = function(argument1,argument2)
  {
          instructions
          return(sortie)
  }
```

Les éléments les plus importants sont les arguments et l'instruction `return`.

La commande *return*

Arguments

Les arguments sont des variables

4.5 Mise en application

Test par permutation

Lorsque les données devient de la normalité, on peut préférer réaliser un test paramétrétique avec des permutations plutôt qu'un test non paramétrique. La plupart des programmes de statistique n'offrent pas cette possibilité qui demande pourtant très peu d'efforts pour être implémentée dans R. Dans cette mise en application, on veut effectuer un test t, pour comparer deux distributions, disponibles dans un fichier `s4-data.txt`.

Le principe d'un test par permutations est simple. La première étape est d'effectuer le test sur l'échantillon non permuté, pour obtenir la valeur de la statistique (T). Dans le cas du test t, R propose la fonction `t.test`, et un rapide survol de `?t.test` vous donnera les arguments nécéssaires et la

manière de récupérer la statistique. Commencent ensuite les permutations a proprement parler. Pour un nombre n d'itérations choisies (en général 9999), on mélange l'ensemble des valeurs des deux distributions. On reconstruit ensuite, en tirant au hasard dans le *pool* de valeurs ainsi formées, deux distributions de taille égale. Cette étape peut, par example, prendre la forme d'une fonction `resample`, qui prendrait une `data.frame` avec deux colonnes (la valeur, et le groupe d'origine) en argument. R propose la fonction `sample`, qui permettra de mélanger la colonne correspondant au groupe (ce qui recréera automatiquement les deux distributions – économisons nous !). Une fois les deux distributions reconstruites, on calcule la nouvelle statistique T'. Si la valeur de T' est inférieure ou égale à la valeur de T, on incrémente une variable N de 1. Sinon, la valeur de N reste la même.

Le calcul de la *p-value* se fait de la manière suivante :

$$p = \frac{N}{n+1} \tag{4.1}$$

À partir de ces informations, et des informations données dans l'introduction de cette séance, vous devez être en mesure de programmer sans difficultés une fonction `t.test.permut`, qui permet de réaliser un test t par permutations. En bonus, vous pouvez ajouter des arguments qui permettent de contrôler le nombre de permutations qui doivent être réalisées.

4.6 Solution des mises en application

Test par permutation

On commence par écrire une fonction `resample`, qui mélange l'attribution des valeurs à un des deux groupes.

```
> resample = function(df)
  {
          df$group = sample(df$group)
          return(df)
  }
```

Puis on écrit la fonction `t.test.permut`, qui effectue les permutations à proprement parler

```
> t.test.permut = function(df,n=9999)
  {
          baseStat = t.test(value~group,df)$statistic
          N = 0
          for(repl in c(1:n))
                  if(t.test(value~group,resample(df))$statistic<=baseStat)
                          N = N + 1
          return(N/(n+1))
  }
```

On peut générer un jeu de données de test :

```
> value = c(rnorm(100,mean=0),rnorm(100,mean=1))
> group = c(rep(a,100),rep(b,100))
> test_df = as.data.frame(cbind(value = value, group = group))
> test_df$value = as.numeric(as.vector(test_df$value))
```

puis vérifier que la *p-value* est inférieure à 0.05 :

```
> print(t.test.permut(test_df, n = 9))
[1] 0
```

Pour éviter les problèmes de lenteur des boucles `for`, on peut écrire la même fonction basée sur la fonction `replicate` :

```
> t.test.permut = function(df,n=9999)
  {
          baseStat = t.test(value~group,df)$statistic
          return(sum(replicate(n,
                  t.test(value~group,resample(df))$statistic<=baseStat)
          )/(n+1))
  }
```

La différence sur 10000 réplicats est assez minime (3 secondes environ). Mais dans un contexte ou il faut analyser plusieurs fois des jeux de données, ces petits écarts finissent par faire une différence importante.

5 Graphiques

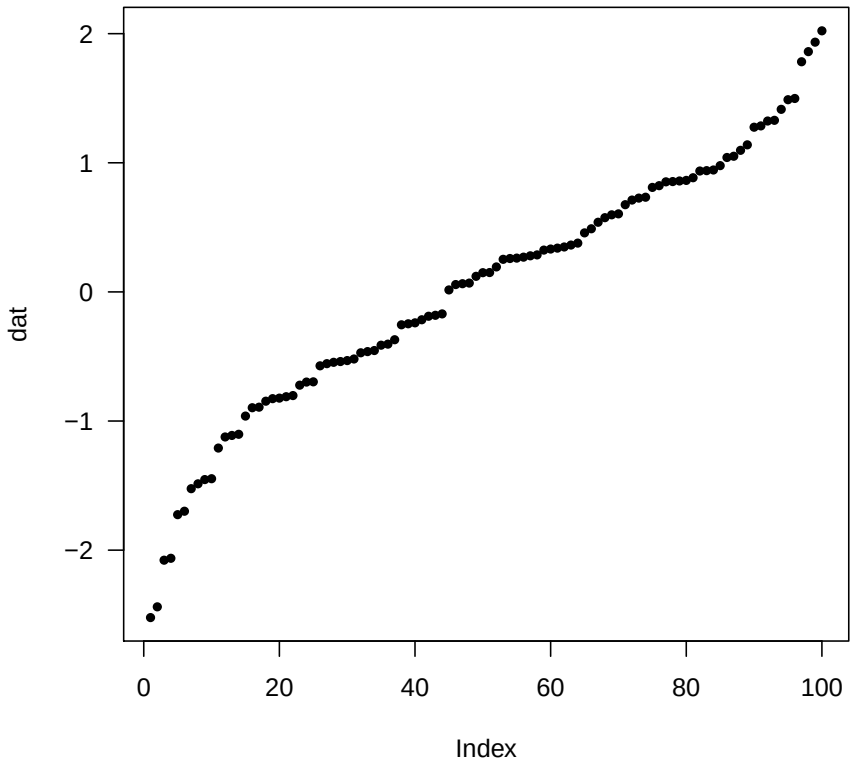
5.1 Quelques généralités sur les graphiques

5.2 Principaux types de visualisations

Nuages de points

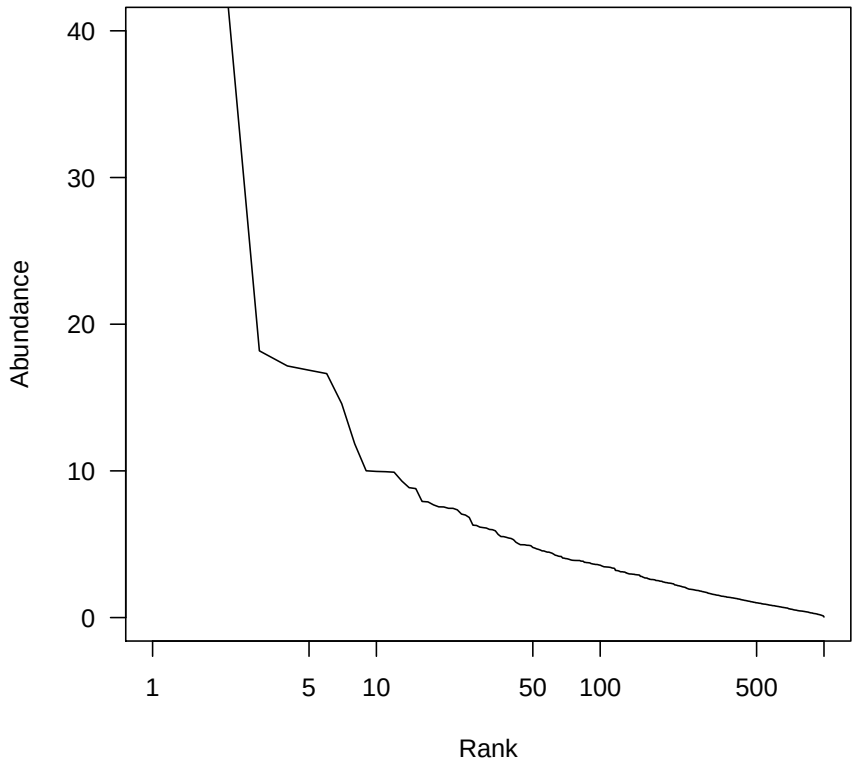
La manière la plus simple de représenter un objet dans R est d'utiliser la fonction `plot`. Par exemple,

```
> dat = sort(rnorm(100))
> plot(dat)
```



R va prendre en charge le calcul d'une grande partie des paramètres nécéssaires à la visualisation, comme par exemple les limites des différents axes, et l'espacement entre les valeurs sur les axes. Il est possible de manuellement spécifier l'ensemble de ces paramètres. Par exemple, on peut vouloir changer les étiquettes des axes x et y par quelque chose de plus explicite. Un exemple typique est un graphique très utilisé en écologie, la *rank-abundance curve*.

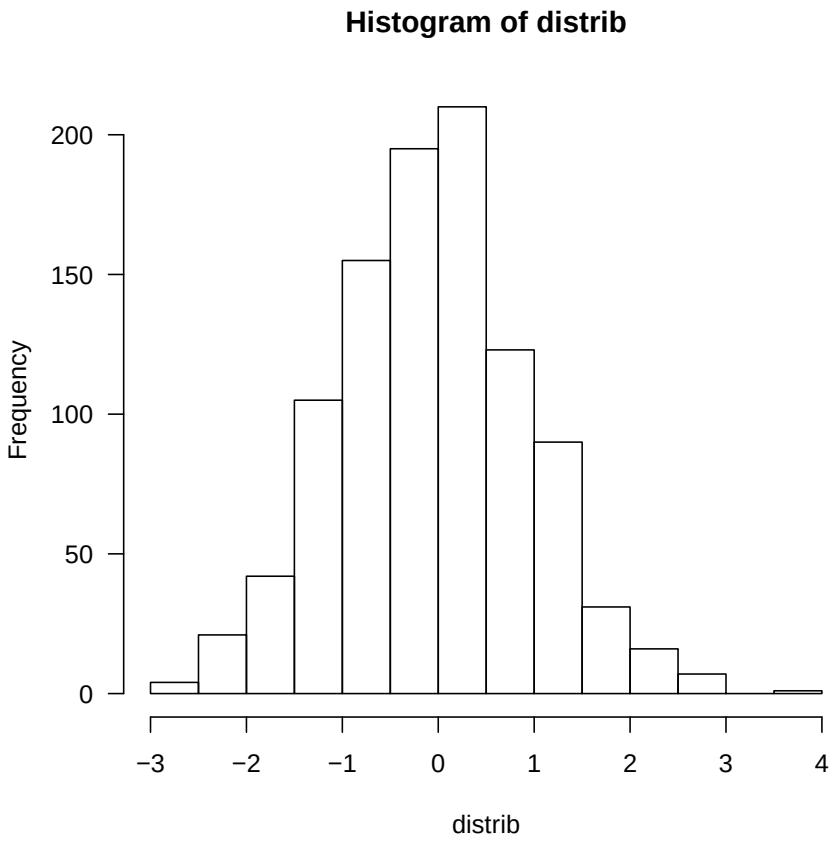
```
> abun = rlnorm(1000)
> plot(sort(abun,decr=TRUE),xlab=Rank,ylab=Abundance,
                  xlim=c(1,1000),ylim=c(0,40),
                  type=l,log=x)
```



Histogrammes

R offre la possibilité de représenter facilement des distributions, *via* des commandes particulières. La plus simple d'utilisation est `hist`, qui permet de représenter un histograme.

```
> distrib = rnorm(1000)
> hist(distrib)
```



Boxplots

5.3 Ajout d'éléments sur un graphique

Autres séries de données

Légendes et axes

Annotations

5.4 Enregistrement des figures

5.5 Mise en application