Introduction à la manipulation de série temporelle avec R

MAP-STA2 : Séries chronologiques

Yannig Goude yannig.goude@edf.fr

2023-2024

Contents

Introduction 2

Tableaux de données avec R 2
- Data Frames 2
- tibbles 5

Importer des données en R à partir d'un fichier plat 6
- Fonctions de base 6
- Le package readr 7

Importer des données de séries temporelles 7
- Gestion des dates avec les fonctions de base 7
- Gestion des dates avec lubridate 9

Classes ts et msts 10

Classes zoo 10
- Classe xts: eXtensible Time Series 11

Représentation graphique de séries temporelles 11
- plot 11
- plot.ts 12
- plot.zoo 13
- plot.xts 14
- package dygraphs 15

Statistiques de base d'une série temporelle 16

Analyse descriptive avec R 16
- boxplot 17
- histogrammes 17
- croisement par facteurs 18

Exemple 1: données de polution de l'air 19

Exemple 2: données de traffic internet 21

Quelques exemples de sources de données: 23

Introduction

Ce document présente les notions de R spécifiques à la manipulation et l'analyse de base de séries temporelles. Une série temporelle ou chronologique (nous utiliserons les deux termes indifféremment) correspond à une série d'observations réalisées au cours du temps. Ces observations pouvant être régulièrement ou non espacées dans le temps. Historiquement, elles ont été utilisées en astronomie (*on the periodicity of sunspots*, 1906), en météorologie (*time-series regression of sea level on weather*, 1968), en théorie du signal (*Noise in FM receivers*, 1963), en biologie (*the autocorrelation curves of schizophrenic brain waves and the power spectrum*, 1960) ou encore en économie (*time-series analysis of imports, exports and other economic variables*, 1971)

Le but de l'analyse de série temporelle est de décrire/analyser des données de ce type, par exemple:

- en économie: détecter et caractériser des périodes de crises, des corrélations entre différents indicateurs économiques.
- en traitement du signal: reconnaitre des séquences de mots dans un signal audio, reconnaitre le type de musique/ les intruments d'un enregistrement audio.
- en météorologie: détecter des changements dans une série de mesures (changement climatique), prévoir des événements extrêmes.
- dans l'industrie: détecter une anomalie dans une chaîne de production.
- dans le commerce: prévoir des ventes de certains produits.
- dans le tertiaire: prévoir le nombre d'appels dans un call-center.

Tableaux de données avec R

Data Frames

L'objet permettant de stocker des tableaux de données en R est la data frame. Il s'agit d'une liste de vecteur de même taille, potentiellement de type (numeric, character) différent.

```
x1<-c(1:5)
x2<-c("a", "b", "c", "d", "e")
x3<-c(rep(TRUE, 3), rep(FALSE, 2))

data_test<-data.frame(x1, x2, x3)
```

Par défaut le nom des éléments de la liste est celui des vecteurs la constituant. Il est possible de les modifier via la fonction `names()`:

```
names(data_test)<-c("var1", "var2", "var3")
```

```
data_test
```

```
##   var1 var2  var3
## 1    1    a  TRUE
## 2    2    b  TRUE
## 3    3    c  TRUE
## 4    4    d FALSE
## 5    5    e FALSE
```

Lorsqu'on affiche la data frame, la première ligne de la table, appelée header, contient le nom des colonnes. Les observations constituent les lignes de la table.

Pour connaitre les dimensions de la data frame, utiliser `dim()`, `ncol()` ou `nrow()`.

Il est souvent pratique de visualiser les premières lignes de la table à l'aide de `head()`

```
head(data_test, 2)
```

```
##   var1 var2 var3
```

```
## 1    1    a TRUE
## 2    2    b TRUE
```

Une fonction très utile, notamment lorsqu'on manipule des data frame est la fonction **str**, elle permet d'obtenir le type de chacune des variables de la data frame:

```
str(data_test)
```

```
## 'data.frame':    5 obs. of  3 variables:
##  $ var1: int  1 2 3 4 5
##  $ var2: chr  "a" "b" "c" "d" ...
##  $ var3: logi  TRUE TRUE TRUE FALSE FALSE
```

La fonction `summary()` permet de calculer des statistiques de base de la data frame:

```
summary(data_test)
```

```
##       var1       var2              var3        
##  Min.   :1   Length:5           Mode :logical  
##  1st Qu.:2   Class :character   FALSE:2        
##  Median :3   Mode  :character   TRUE :3        
##  Mean   :3                                     
##  3rd Qu.:4                                     
##  Max.   :5                                     
```

Pour accéder à la colonne i de la table, la commande est `data_test$vari` ou `data_test[[i]]` ou `data_test[,"vari"]`. Pour accéder à la cellule de la table correspondant à la ligne i et la colonne j l'instruction est `data_test[i,j]`.

Remarquons que le type d'objet obtenu lorsqu'on sélectionne une variable de la data frame dépend de l'instruction:

```
x<-data_test$var1
str(x)
```

```
##  int [1:5] 1 2 3 4 5
```

```
x<-data_test[,1]
str(x)
```

```
##  int [1:5] 1 2 3 4 5
```

```
x<-data_test[ , "var1", drop = TRUE]
str(x)
```

```
##  int [1:5] 1 2 3 4 5
```

```
x<-data_test[ , "var1", drop = FALSE]
str(x)
```

```
## 'data.frame':    5 obs. of  1 variable:
##  $ var1: int  1 2 3 4 5
```

Pour ajouter une nouvelle variable à une data frame existante:

```
data_test$var4<-c(10:14)
data_test<-data.frame(data_test, var5=rep('statML', 5))
head(data_test)
```

```
##   var1 var2  var3 var4   var5
## 1    1    a  TRUE   10 statML
## 2    2    b  TRUE   11 statML
```

```
## 3    3    c  TRUE   12 statML
## 4    4    d FALSE   13 statML
## 5    5    e FALSE   14 statML
```

Pour supprimer des colonnes dans une data frame, on peut exploiter la fonction `subset`

```
data_test_cut <- subset(data_test, select = -c(var1, var2) )
head(data_test_cut, 2)
```

```
##   var3 var4   var5
## 1 TRUE   10 statML
## 2 TRUE   11 statML
```

ou utiliser les numéros des colonnes de la data frame:

```
data_test_cut <- data_test[, -c(1,2)]
head(data_test_cut, 2)
```

```
##   var3 var4   var5
## 1 TRUE   10 statML
## 2 TRUE   11 statML
```

ou utiliser les noms des variables:

```
drop <- c("var1","var2")
data_test_cut <- data_test[,!(names(data_test) %in% drop)]
head(data_test_cut, 2)
```

```
##   var3 var4   var5
## 1 TRUE   10 statML
## 2 TRUE   11 statML
```

Pour extraire un sous-ensemble de variables ou de colonne on peut de la même façon,

utiliser les numéros des colonnes de la data frame:

```
data_test_sub <- data_test[c(1,2)]
head(data_test_sub, 2)
```

```
##   var1 var2
## 1    1    a
## 2    2    b
```

utiliser les noms de variables:

```
keep <- c("var1","var2")
data_test_sub <- data_test[keep]
head(data_test_sub, 2)
```

```
##   var1 var2
## 1    1    a
## 2    2    b
```

utiliser la fonction `subset()`:

```
data_test_sub <- subset(data_test, select = c(var1, var2) )
head(data_test_sub, 2)
```

```
##   var1 var2
## 1    1    a
## 2    2    b
```

tibbles

Les tibbles sont une variante des data frame, proposée par Hadley Wickham dans la suite de package "tidyverse". Il s'agit d'une version moderne de data frame oubliant certains aspects désuets des data frame (convertion des caractères en facteurs, print pas très beau, amélioration de la détection des champs...). Pour une description détaillée faire `vignette("tibble")`.

Il existe deux grandes différences entre les tibbles et les data frame: l'affichage et le découpage en sous-ensembles.

Affichage

La fonction print de la classe tibbles représente seulement les 10 premières lignes du tibble et un nombre de colonnes pouvant s'afficher à l'écran (on peut également customisé l'affichage avec `options(tibble.print_max = n, tibble.print_min = m)`). Le type de chaque variable est précisé. Considérons l'exemple des données de consommation électrique que nous manipulerons au cours des TP de modélisation suivants.

```
Data0<-read_delim("Data_cours/cdc_conso.csv", delim=';')
Data0
```

```
## # A tibble: 169,488 x 5
##    `Date - Heure`      Date       Heure  Consommation `Qualite donnee`
##    <dttm>              <date>     <time>        <dbl> <chr>
##  1 2009-06-06 04:00:00 2009-06-06 06:00         35984 Définitive
##  2 2009-06-06 07:00:00 2009-06-06 09:00         43188 Définitive
##  3 2009-06-06 09:30:00 2009-06-06 11:30         46959 Définitive
##  4 2009-06-06 10:30:00 2009-06-06 12:30         48622 Définitive
##  5 2009-06-06 18:30:00 2009-06-06 20:30         42578 Définitive
##  6 2009-06-06 22:00:00 2009-06-07 00:00         45315 Définitive
##  7 2009-06-06 22:30:00 2009-06-07 00:30         43440 Définitive
##  8 2009-06-07 01:00:00 2009-06-07 03:00         37109 Définitive
##  9 2009-06-07 02:00:00 2009-06-07 04:00         35001 Définitive
## 10 2009-06-07 03:30:00 2009-06-07 05:30         33886 Définitive
## # i 169,478 more rows
```

On remarque que le type de chaque variable a été inféré lors de l'import des données. Par exemple, la variable Date est bien de type Date (et non une chaine de caractère comme ça l'aurait été avec la fonction d'import de csv du R de base).

Extraction de sous-ensembles

De même qu'avec le type data frame, pour extraire une variable il est possible d'utiliser: $, [["nom"]], [[numero]].

```
Data0$Consommation%>%head
```

```
## [1] 35984 43188 46959 48622 42578 45315
```

```
Data0[["Consommation"]]%>%head
```

```
## [1] 35984 43188 46959 48622 42578 45315
```

```
Data0[[2]]%>%head
```

```
## [1] "2009-06-06" "2009-06-06" "2009-06-06" "2009-06-06" "2009-06-06"
## [6] "2009-06-07"
```

Cela peut être inclu dans un pipe via l'indication ".":

```r
Data0%>%.$Consommation%>%head
```

```
## [1] 35984 43188 46959 48622 42578 45315
```

```r
Data0%>%.[["Consommation"]]%>%head
```

```
## [1] 35984 43188 46959 48622 42578 45315
```

```r
Data0%>%.[[2]]%>%head
```

```
## [1] "2009-06-06" "2009-06-06" "2009-06-06" "2009-06-06" "2009-06-06"
## [6] "2009-06-07"
```

Pour l'extraction de lignes il est possible d'utiliser l'instruction de base:

```r
Data0[1:4,]
```

```
## # A tibble: 4 x 5
##   `Date - Heure`      Date       Heure  Consommation `Qualite donnee`
##   <dttm>              <date>     <time>        <dbl> <chr>
## 1 2009-06-06 04:00:00 2009-06-06 06:00         35984 Définitive
## 2 2009-06-06 07:00:00 2009-06-06 09:00         43188 Définitive
## 3 2009-06-06 09:30:00 2009-06-06 11:30         46959 Définitive
## 4 2009-06-06 10:30:00 2009-06-06 12:30         48622 Définitive
```

pour des requêtes plus complexes il est plus commode (et conseillé) d'utiliser les fonctionnalités du package `dplyr::select()` et `dplyr::filter()`. Notons que, contrairement au R de base ou l'instruction [] peut renvoyer une data.frame ou un vecteur, avec tibble elle renvoie toujour un tibble.

Importer des données en R à partir d'un fichier plat

Fonctions de base

Le type de fichiers plat le plus courant est le fichicher texte ".txt" délimité (tabulation, virgule, point virgule...). L'import de ce type fichiers plat en R se fait via la fonction `read.table()` dont il existe différentes variantes (`read.csv()`, `read.delim()`...) qui se distinguent selon les valeurs par défaut des paramètres suivants:

- le séparateur
- le "header" (la première ligne contient le nom des variables ou pas)
- l'argument fill, fill=TRUE signifie que si les lignes ne sont pas de même taille, les trous sont comblés par des champs vides.

Ces fonctions supposent que les données du fichier texte soient organisées telles que les lignes soient les observations, les colonnes les variables.

Un autre type de fichier couramment utilisé est le fichier excel ".xls". Il existe le package `XLConnect` dont la fonction `readWorksheetFromFile()` réalise l'import de feuilles excel.

```r
library(XLConnect)
XLdata <- readWorksheetFromFile("<file name and extension>", sheet = 1)
```

Le format JSON (JavaScript Object Notation) est un format de données populaire utilisé pour la lecture, le stockage et l'échange d'information (notamment sur le web, Yahoo et Google utilisent JSON dès 2005 et 2006). La plupart des languages informatiques peuvent générer et lire le format JSON. Il est donc devenu très populaire pour le stockage, la lecture et le partage d'information dans les applications et services web. Il existe le package `rjson` dont la fonction `fromJSON()` réalise l'import de fichier.

```r
library(rjson)
JsonData <- fromJSON(file= "<filename.json>" )
```

Pour importer et manipuler des données au format XML, il existe le package `XML`, la fonction `xmlTreeParse()` permet d'analyser directement des données du web:

```
library(XML)
xmlfile <- xmlTreeParse("<Your URL to the XML data>")
```

il faut ensuite manipuler les données ainsi importées (toujours au format XML) à l'aide des fonctions `xmlSApply()`, `xmlRoot()`... pour se construire une data.frame.

Le package readr

Il existe dans readr 7 fonctions pour l'import de fichiers plats.

Import function	Description
read_csv()	lit des fichiers dont le séparateur est une virgule
read_csv2()	lit des fichiers dont le séparateur est un point virgule
read_fwf()	lit des fichiers dont les champs sont de largeur fixée
read_table()	lit des fichiers dont le séparateur est un espace
read_log()	lit des fichiers de type Apache log
read_tsv()	lit des fichiers dont le séparateur est la tabulation
read_delim()	lit des fichiers dont le séparateur est à préciser

Pour plus de détails sur les arguments de ces fonctions voir l'aide `?read_csv`.

Pour inférer le type d'une variable, les fonctions développées dans readr procèdent à une heuristique consistant à lire les 1000 premières observations et à déduire via des règles simples le type de chaque variable.

L'algorithme essaye les types suivants:

- logical: si le champ contient seulement "F", "T", "FALSE", ou "TRUE".
- integer: si le champ contient seulement des charactères numériques et -.
- double: si le champ contient seulement des réels valides (incluant les valeurs de type 4.5e-5).
- number: si le champ contient seulement des réels valides avec des marqueurs de groupe (virgule, millier, millions...)
- time: correspond au format par défaut time_format.
- date: correspond au format par défaut date_format.
- date-time: ISO8601 date.

si le champ n'est dans aucun de ces cas la, il reste une chaîne de charactère.

Cette heuristique peut parfois être mise en défaut et il faut alors agir plus spéficiquement. La fonction `problems()` permet d'obtenir un tibble des erreurs d'analyse associé à chaque colonne et permet de remonter à la source de certains problèmes de type anomalie dans les données, valeurs manquantes etc.

Plusieurs arguments sont en faveur d'utiliser ces fonctions d'import plutôt que celles de base:

- reproductibilité sur un autre PC: les fonctions du R de base sont souvents dépendantes de paramètres locaux (OS, variables d'environnement)
- rapidité: environ 10 fois plus rapides que les fonction de R base. bar de progrès très utile pour les gros volumes de données.
- importe directement en type tibble.

Importer des données de séries temporelles

Gestion des dates avec les fonctions de base

Avant toute analyse d'une série temporelle, il est nécessaire d'importer les données dans un format reconnu par R. Un format classique d'échange de données est le fichier texte. Comme nous l'avons vu, l'import en R

de ce type de fichier se fait par les commandes R:

```
read.table
read.csv
scan
```

par exemple, prenons le fichier `internet-traffic-data-in-bits-fr.csv` disponible à l'adresse http://robjhyndman.com/TSDL/ et correspondant au traffic internet mesuré par un fournisseur d'accès à internet du Royaume-Uni de novembre 2004 à janvier 2005 au pas 5 minutes. Ce fichier est de type csv, et est donc importable via les instructions suivantes:

```
data<-read.csv("Data_cours/internet-traffic-data-in-bits-fr.csv",
               dec=".",sep=",",header=F,skip=1)
```

L'option `dec=` défini le type de décimale (par exemple , ou .), `sep=` le type de séparateur entre les entrées de la table (par exemple ; ou `\t` pour un fichier tabulé), `header=` un booléen précisant si la première ligne du fichier correspond au nom des variables ou pas, `skip=n` indique si l'importation commence après les n premières lignes (utile si les premières lignes du fichier texte sont une description des données).

La particularité des données de série temporelle est leur indexation par le temps. Une bonne gestion de ces données passe donc bien souvent par la gestion d'un format de date adéquate. Nous reviendrons sur ces formats plus en détail dans la suite.

Pour préciser la classe d'une colonne de données, il est possible d'utiliser l'argument `colClasses` de la fonction `read.csv` et préciser s'il s'agit le type "Date","numeric","character" ou "factor". Dans notre exemple, l'instruction:

```
data<-read.csv("internet-traffic-data-in-bits-fr.csv",
               dec=".",sep=",",header=F,skip=1,colClasses=c("Date","numeric"))
```

précise que la 1ère colonne des données est de la classe Date, et la 2e de la classe numeric. Toutefois, par défaut le format de la classe Date créé est au format "%Y-%m-%d" (ex: 2004-11-19). Pour gérer des formats plus spécifiques, comme c'est le cas ici, une possibilité est soit d'importer les données au format caractère puis de convertir la chaine de charactère correspondant à la date à l'aide des fonctions `as.POSIXct` et `strptime`:

```
data<-read.csv("internet-traffic-data-in-bits-fr.csv", dec=".",sep=",",header=F,skip=1)
tail(data$V1)
Date=as.POSIXct(strptime(data$V1, "%Y-%m-%d %H:%M:%S"))
Traffic<-as.numeric(data$V2)
data<-data.frame(Date,Traffic)
```

soit de définir un format personalisé en amont de l'importation de la table:

```
setClass('myDate')
setAs("character","myDate", function(from) as.POSIXct(strptime(from, "%Y-%m-%d %H:%M:%S")))
data<-read.csv("Data_cours/internet-traffic-data-in-bits-fr.csv",
               dec=".",sep=",",header=F,skip=1,colClasses =c("myDate","numeric"),
               col.names=c("Date","Traffic"))
```

Une fois importé, il est indispensable de vérifier la cohérence des données. Pour cela, utilisez la fonction `head` (resp. `tail`) qui permettent d'afficher les premières (resp. dernières) lignes des données, et la fonction `summary` qui calcul les statistiques de base pour chacune des variables de la table.

```
head(data)
```

```
##                  Date  Traffic
## 1 2004-11-19 09:30:00 4838.665
## 2 2004-11-19 09:35:00 4845.177
## 3 2004-11-19 09:40:00 5157.996
## 4 2004-11-19 09:45:00 5637.876
```

```
## 5 2004-11-19 09:50:00 5520.690
## 6 2004-11-19 09:55:00 5626.337
```

```
tail(data)
```

```
##                      Date  Traffic
## 19883 2005-01-27 10:20:00 6438.786
## 19884 2005-01-27 10:25:00 6515.512
## 19885 2005-01-27 10:30:00 6291.171
## 19886 2005-01-27 10:35:00 6305.519
## 19887 2005-01-27 10:40:00 6422.144
## 19888 2005-01-27 10:45:00 6511.014
```

```
summary(data)
```

```
##       Date                        Traffic
##  Min.   :2004-11-19 09:30:00   Min.   : 1060
##  1st Qu.:2004-12-06 15:48:45   1st Qu.: 2364
##  Median :2004-12-23 22:07:30   Median : 3494
##  Mean   :2004-12-23 22:07:30   Mean   : 3867
##  3rd Qu.:2005-01-10 04:26:15   3rd Qu.: 4889
##  Max.   :2005-01-27 10:45:00   Max.   :10671
```

Gestion des dates avec lubridate

Lubridate permet de simplifier la gestion des dates en automatisant ce processus. Lubridate contient un certaint nombre de fonctions qui analysent et détecte le format date de vos données.

Par exemple:

```
library(lubridate)
ymd("20171006"); mdy("06-10-2017"); dmy("06/10/2017"); ymd_hms("2017-10-06 14:30:00");
```

```
## [1] "2017-10-06"

## [1] "2017-06-10"

## [1] "2017-10-06"

## [1] "2017-10-06 14:30:00 UTC"
```

Dans le cas de date incluant l'heure il est important de préciser le fuseau horaire (liste des time zones ici: https://en.wikipedia.org/wiki/List_of_tz_database_time_zones):

```
ymd_hms("2017-10-06 14:30:00", tz = "GMT")
```

```
## [1] "2017-10-06 14:30:00 GMT"
```

Le package lubridate propose des fonctions utiles pour la gestion des fuseaux horaires, comme `with_tz()` et `force_tz()`.

```
date_today<-ymd_hms("2017-10-06 14:30:00", tz = "GMT")
with_tz(date_today, "America/Guyana")
```

```
## [1] "2017-10-06 10:30:00 -04"
```

Il est également possible d'extraire (ou modifier) de l'information d'une date à l'aide des fonctions `second()`, `minute()`, `hour()`, `day()`, `wday()`, `yday()`, `week()`, `month()`, `year()`, et `tz()`.

```
date_today<-ymd_hms("2017-10-06 14:30:00", tz = "GMT")
hour(date_today)
```

```
## [1] 14
```

```
hour(date_today)<-15
date_today
```

```
## [1] "2017-10-06 15:30:00 GMT"
```

```
wday(date_today, label=T)
```

```
## [1] Fri
## Levels: Sun < Mon < Tue < Wed < Thu < Fri < Sat
```

Classes ts et msts

Pour manipuler et modéliser (modèles SARIMA, lisssages exponentiels, représentation graphique...) des séries temporelles en R, il est intéressant d'utiliser la classe:

```
ts(data = NA, start = 1, end = numeric(), frequency = 1)
```

L'argument `frequency` correspond au nombre d'observations par saison (en supposant que la série en question ne possède qu'une saisonnalité). Il prend par exemple les valeurs suivantes si on suppose que le cycle des données est annuel:

Type de données	frequency
Annuelles	1
Trimestrielles	4
Mensuelles	12
Hedbdomadaires	52

attention l'argument `frequency` n'a pas ici le même sens que la définition de fréquence en physique c'est à dire l'inverse de la période. Par exemple, dans le cas des données de traffic internet, la saisonnalité majeure étant journalière on peut convertir ces données en série temporelle ainsi:

```
Traffic.ts<-ts(data$Traffic,start=1,frequency=24*12)
```

les arguments `start` et `end` correspondent à la date de début et de fin de la période d'observation des données.

La classe `msts` est implémentée dans le package `forecast`, disponible sur le CRAN. Elle est intéressante pour modéliser les saisonnalités multiples. Sa syntaxe est proche de celle de `ts`:

```
msts(data, seasonal.periods=, ts.frequency=)
```

mais l'argument `seasonal.periods=` est un vecteur correspondant aux différentes saisonnalités. `ts.frequency` joue le même rôle que pour la classe `ts`. Par exemple, sur les données de traffic internet, en incluant des saisonnalités journalières et hebdomadaires:

```
Traffic.mts <- msts(data$Traffic, seasonal.periods=c(24*12,7*24*12), ts.frequency=24*12, start=1)
```

Classes zoo

le package `zoo` implémente une classe S3 permettant de gérer les observations indexées par le temps. Il permet notamment de traiter les séries temporelles à pas de temps irrégulier.

```
library(zoo)
zoo(x = NULL, order.by = index(x), frequency = NULL)
```

la fonction `zoo` permet de crer un objet `zoo` en précisant les observations dans l'argument `x` et leurs index via l'argument `order.by`. Typiquement l'index correspond à la date et l'heure de l'observation. Par exemple, sur les données de traffic internet:

```
Traffic.zoo<-zoo(data$Traffic,order.by=data$Date)
```

Classe xts: eXtensible Time Series

le package `xts` est une extension du package `zoo`. Il permet d'unifier plusieurs classes de séries temporelles existant en R (ts,zoo…) et donne la possibilité à l'utilisateur de spécicifer ses propres attributs. Nous l'utiliserons principalement en lien avec le package `dygraphs` permettant une visualisation interactive (zoom, fenêtre glissante…) des séries temporelles.

la syntaxe est donc très proche de celle de `zoo`:

```
library(xts)
xts(x = NULL,order.by = index(x),frequency = NULL,unique = TRUE,tzone = Sys.getenv("TZ"))
```

et, sur l'exemple des données de traffic internet:

```
Traffic.xts<-xts(data$Traffic,order.by=data$Date)
```

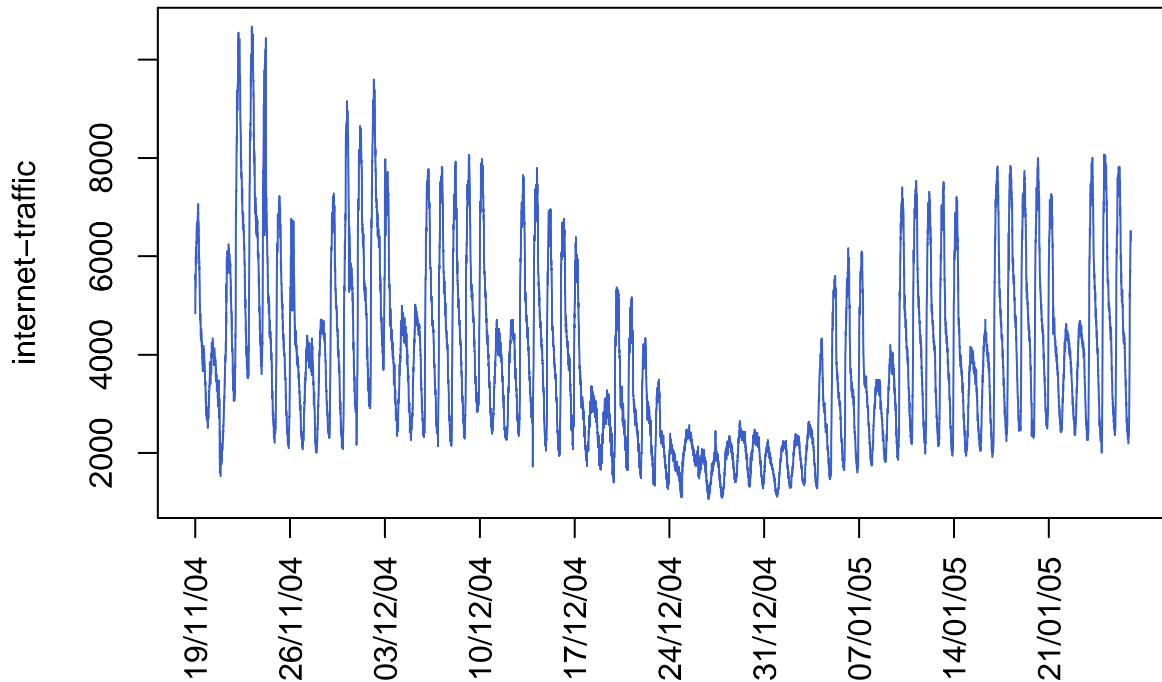
Représentation graphique de séries temporelles

Une fois importée, il est intéressant de représenter graphiquement une série temporelle. Pour représenter un objet en R, la fonction de base est `plot()` qui se décline ensuite de différentes manières selon les objects qu'on représente.

plot

Une bonne utilisation du format de date et un paramétrage correct de la fonction `plot` permet de tracer des graphiques de séries temporelles éléguants tout en étant très générique (au prix de quelques lignes de code, comparé à l'usage des fonction `ts` et `msts`). Pour notre exemple l'instruction:

```
plot(data$Date,data$Traffic,type='l',xaxt="n",xlab='',ylab="internet-traffic",col='royalblue3')
axis.POSIXct(1, at = seq(data$Date[1], tail(data$Date,1),"weeks"), format="%d/%m/%y",las=2)
```

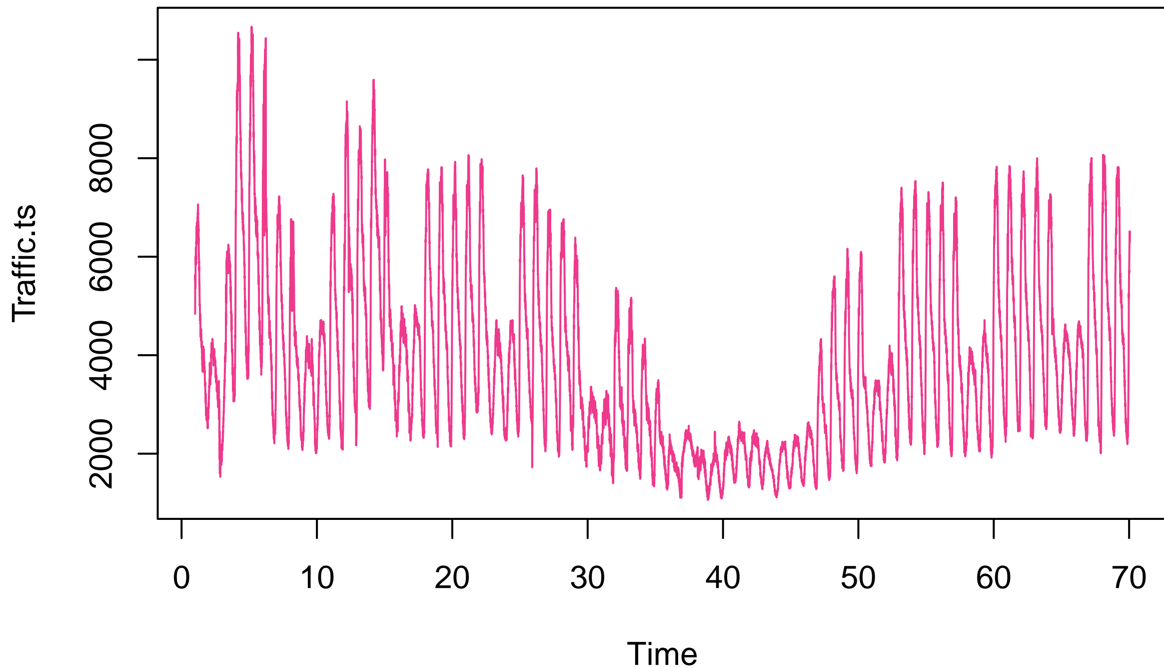


permet d'obtenir un graphique de la série temporelle sous forme de courbe (il s'agit en fait d'interpolation linéaire des points 5 minutes) grâce à l'option `type='l'`. L'axe des abscisses représente les dates, annotées semaine par semaine via la fonction `axis.POSIXct`. Le format des dates est spécifié par l'argument `format="%d/%m/%y"` (voir l'aide de `?strptime` pour plus de détails sur les formats dates). Enfin, l'option `las=2` indique d'afficher les dates verticalement (valeurs possibles: 0 parallel à l'axe, 1 horizontal, 2 perpendiculaire à l'axe, 3 vertical).

plot.ts

Exploitons ici la classe `ts` et la fonction `plot.ts` associée en reprenant l'exemple des données de trafic internet. Les données étant échantillonnées au pas de temps 5 minutes et présentant un cycle journalier évidant nous choissons comme paramètre: `frequency=24*12`.

```
Traffic.ts<-ts(data$Traffic,start=1,frequency=24*12)
plot(Traffic.ts,col='violetred2')
```

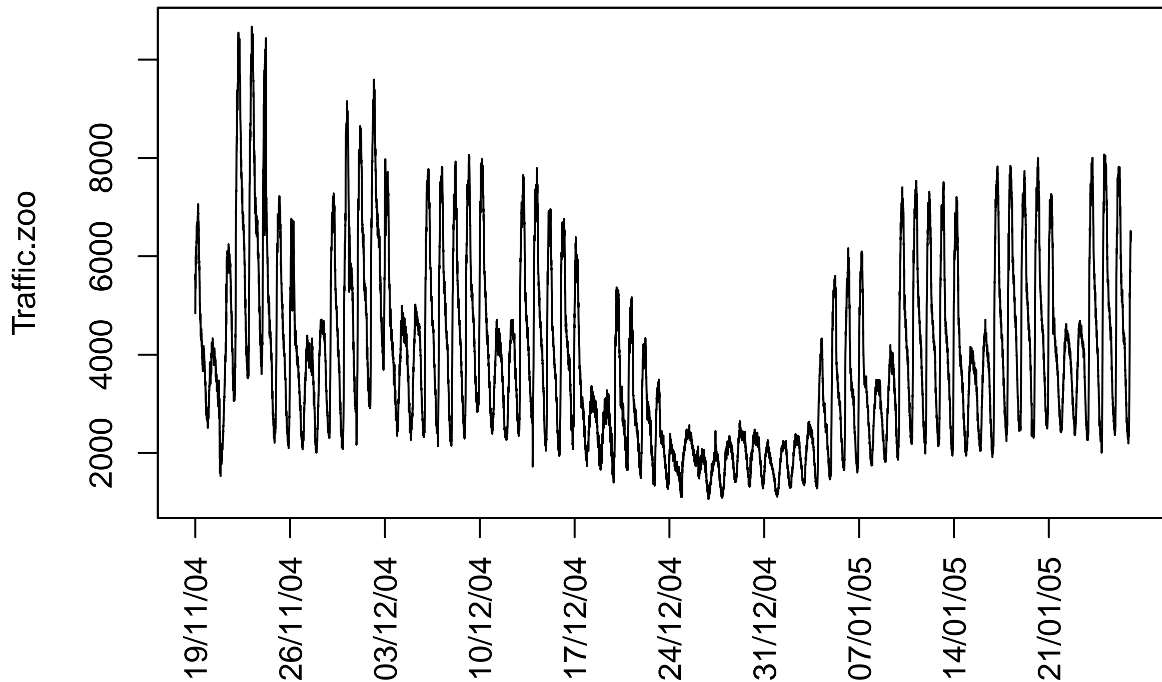


Ce graphique représente la série temporelle en fonction du temps par une courbe. Le temps est indiqué en nombre de périodes c'est à dire ici le nombre de jours.

plot.zoo

Exploitons ici la classe `zoo` et la fonction `plot.zoo` associée en reprenant l'exemple des données de trafic internet.

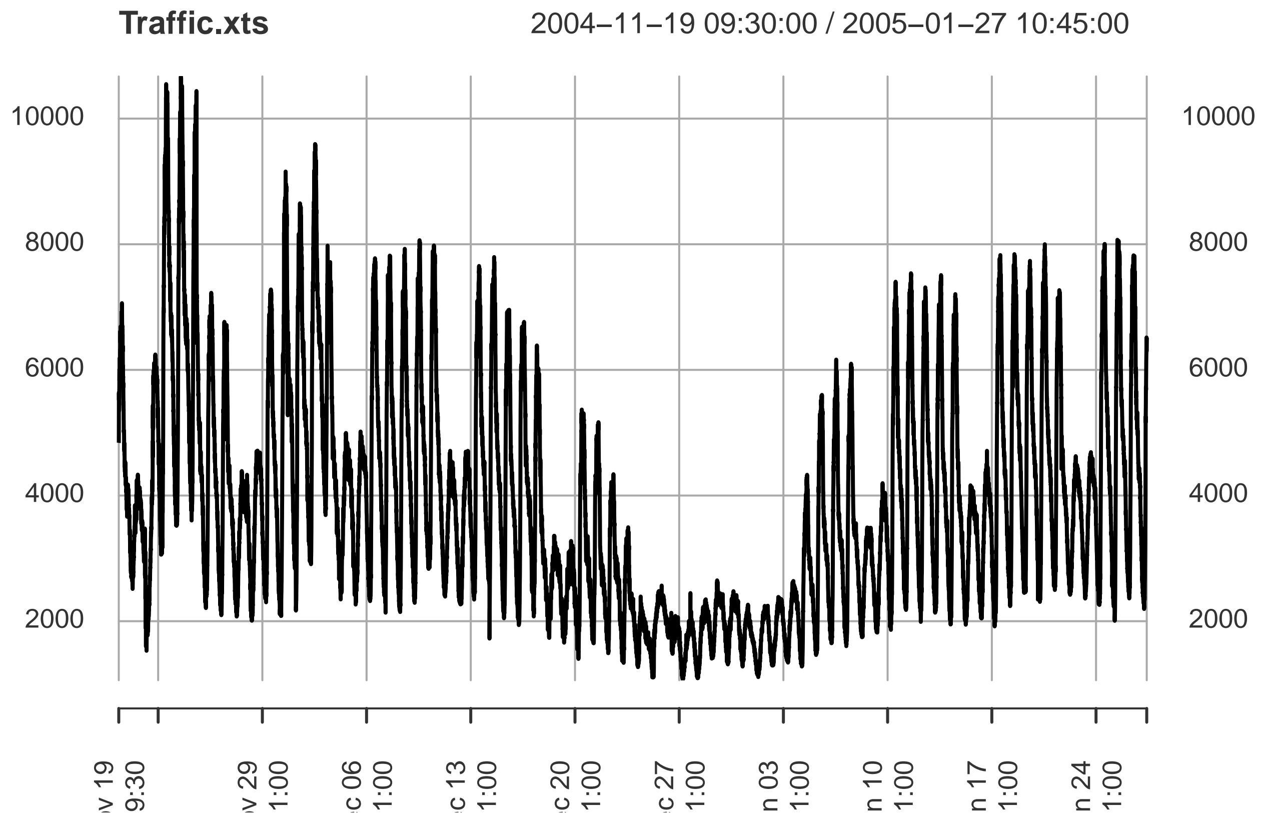
```
Traffic.zoo<-zoo(data$Traffic,order.by=data$Date)
plot(Traffic.zoo, xaxt = "n",xlab='')
t<-time(Traffic.zoo)
axis.POSIXct(1, at = seq(t[1], tail(t,1),"weeks"), format="%d/%m/%y",las=2)
```



plot.xts

Exploitons ici la classe `xts` et la fonction `plot.xts` associée en reprenant l'exemple des données de trafic internet. Cette fonction est très pratique et permet notamment une gestion aisée du format date des abscissses.

```
Traffic.xts<-xts(data$Traffic,order.by=data$Date)
plot(Traffic.xts,type = "l",major.format="%d/%m/%y",las=2)
```



package dygraphs

Le package `dygraphs` permet de visualiser de manière interactive une série temporelle. On peut par exemple sélectionner des périodes de temps, zoomer, faire "glisser" une fenêtre d'observation dans le temps etc. Il se base sur la classe `xts`.

```
library(dygraphs)
library(xts)
```

Considérons des données de consommation électrique au pas hebdomadaire sur plusieurs années. Nous voulons representer la consommation et la température sur un même graphe dynamique afin d'étudier leurs corrélations au cours du temps.

La première étape consiste à construire les objets `xts` associés à ces deux variables Load et Temp.

```
Load=xts(data0$Load,order.by=data0$Date)
Temp=xts(data0$Temp,order.by=data0$Date)
```

Ensuite, construire une matrice de séries temporelles (standardisées pour des raisons de lisibilité):

```
Load.sd=Load/sd(Load)
Temp.sd=Temp/sd(Temp)
time.series=cbind(Load.sd,Temp.sd)
names(time.series)=c("Load.sd","Temp.sd")
```

On obtient ensuite le graphe dynamique par l'instruction suivante (voir la démo en cours pour le résultat).

```
dygraph(time.series)
```

Si on veut ajouter une fenêtre de sélection glissante:

```r
dygraph(sd.time.series)%>% dyRangeSelector()
```

Statistiques de base d'une série temporelle

Pour décrire une série temporelle, plusieurs indicateurs statistiques de base sont utiles:

- la **tendance centrale** d'une série $(y_t)_{1\leq t\leq n}$ est donnée par la moyenne empirique: $\bar{y}_n = \frac{1}{n}\sum_{t=1}^n y_t$
- la variance empirique décrit la **dispersion** de la série autour de sa moyenne: $\sigma_n = \sqrt{\frac{1}{n}\sum_{t=1}^n (y_t - \bar{y}_n)^2}$
- l'auto-covariance ou l'autocorrélation empirique indique le degrés de **dépendance** (linéaire) entre des réalisations successives.

La fonction d'autocovariance empirique est définie par:

$$\gamma_n(h) = \frac{1}{n-h}\sum_{t=1}^{n-h}(y_t - \bar{y}_n)(y_{t+h} - \bar{y}_n)$$

et la fonction d'auto-corrélation empirique:

$$\rho_n(h) = \frac{\gamma_n(h)}{\gamma(0)}$$

en remarquant que $\gamma_n(0) = \sigma_n^2$.

Ainsi $\rho_n(h)$ estime la corrélation entre y_t et y_{t+h} en supposant que cette corrélation existe et qu'elle est "stable" dans le temps (nous verrons plus tard le concept de stationnarité associé à cette idée).

- l'autocorrélation partielle (PACF) permet de quantifier la dépendance linéaire entre deux réalisations successives mais conditionnellement aux réalisations intermédiaires. En effet, deux réalisations peuvent être corrélées articificiellement car corrélées à une même autre réalisation. Pour calculer la PACF d'ordre h il faut considérer le modèle de régression linéaire:

$$y_t = \alpha_1 y_{t-1} + \alpha_2 y_{t-2} + ... + \alpha_h y_{t-h} + \varepsilon_t$$

L'autocorrélation partielle est définie par $\alpha(h)$, et son estimateur est le coefficient de l'estimateur des moindres carrés ordinaires correspondant. Soit en notant $Y = (y_1, y_2, ..., y_n)$, X la matrice dont la i^e ligne est $(y_{i-1}, y_{i-2}, ..., y_{i-h})$ et $\alpha = (\alpha_1, \alpha_2, ..., \alpha_h)$ le problème des MCO s'écrit:

$$\min_{\alpha} ||Y - X\alpha||^2$$

et on a:

$$\widehat{\alpha} = (X'X)^{-1}X'Y$$

Une autre possibilité pour calculer efficacement les autocorrélations partielles à partir des corrélations est d'utiliser l'algorithme de Durbin-Levinson (cf scéance 4: Processus Stationnaires).

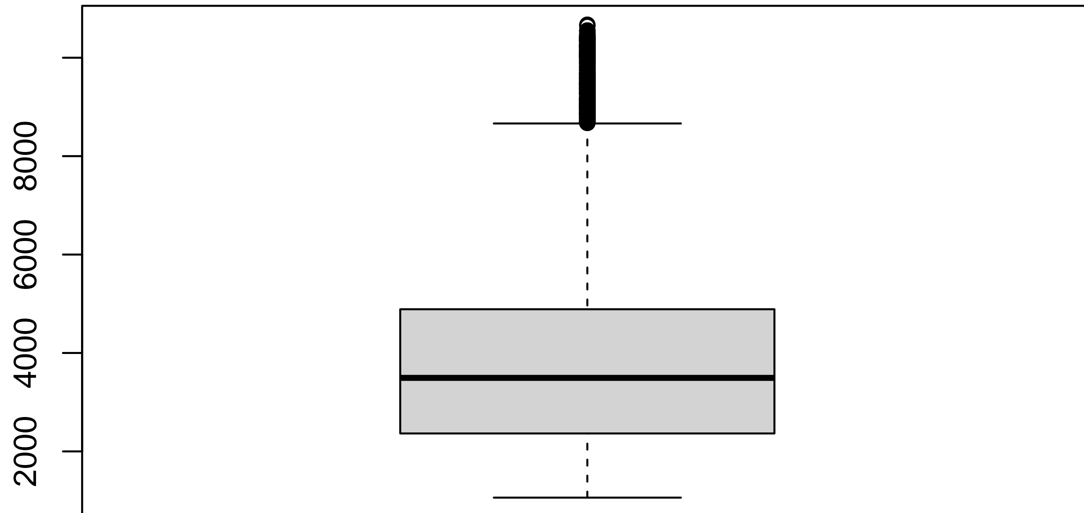
Analyse descriptive avec R

En plus des statistiques de base présentées ci-dessus, l'analyse d'une série temporelle passe bien souvent par une analyse descriptive dont nous présentons certains élements.

boxplot

La boxplot permet de représenter de manière synthétique les caractéristiques centrales et de dispertion d'une population: médiane, quartiles, minimum, maximum ou déciles. Dans le cas d'une série temporelle cette population correspond aux différentes réalisations dans le temps d'un processus.

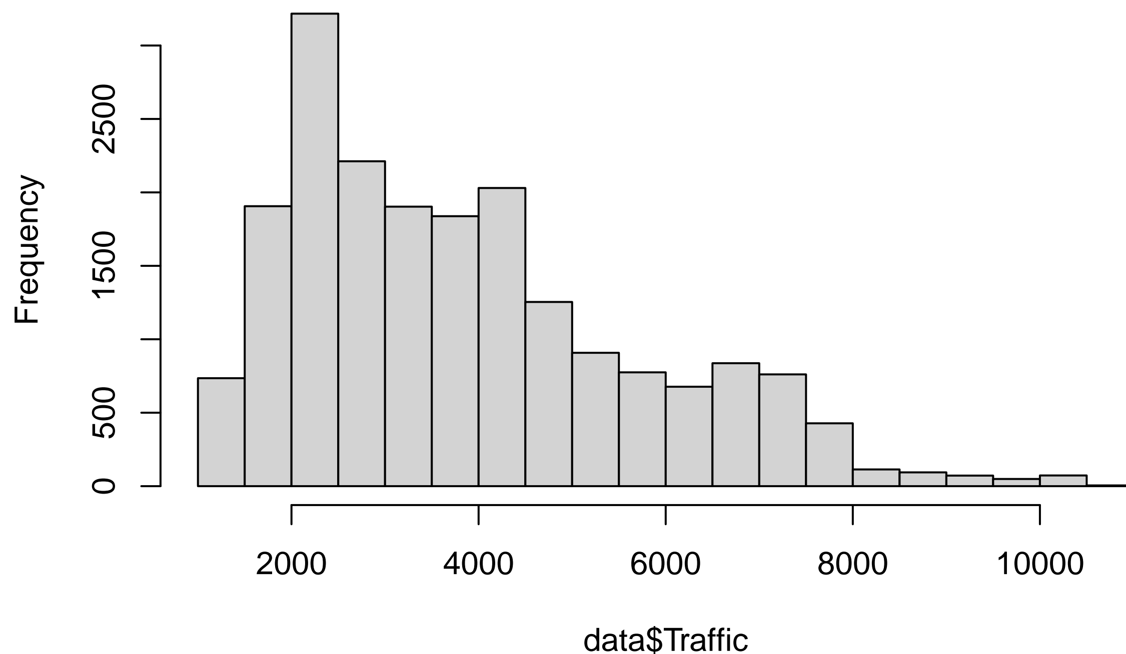
```
boxplot(data$Traffic)
```



histogrammes

L'histogramme donne une estimation de la densité correspondant aux différentes réalisations dans le temps d'un processus.

```
hist(data$Traffic,breaks=20)
```

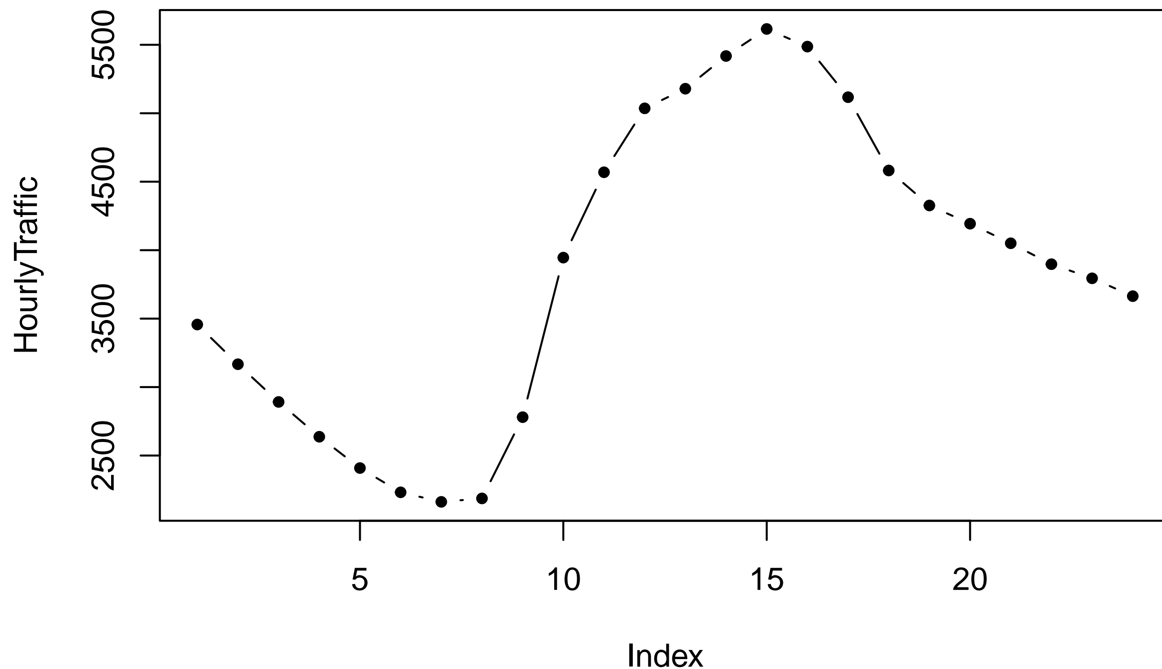


croisement par facteurs

Dans bien des cas, il peut être utile de croiser certains indicateurs descriptifs de la série avec des facteurs potentiellement explicatifs de la série. Par exemple, si l'on observe une série de relevés de température on peut vouloir calculer sa moyenne par saison ou par mois.

Dans le cas de la série de traffic internet, voilà un exemple de calcul de la moyenne du traffic par heure:

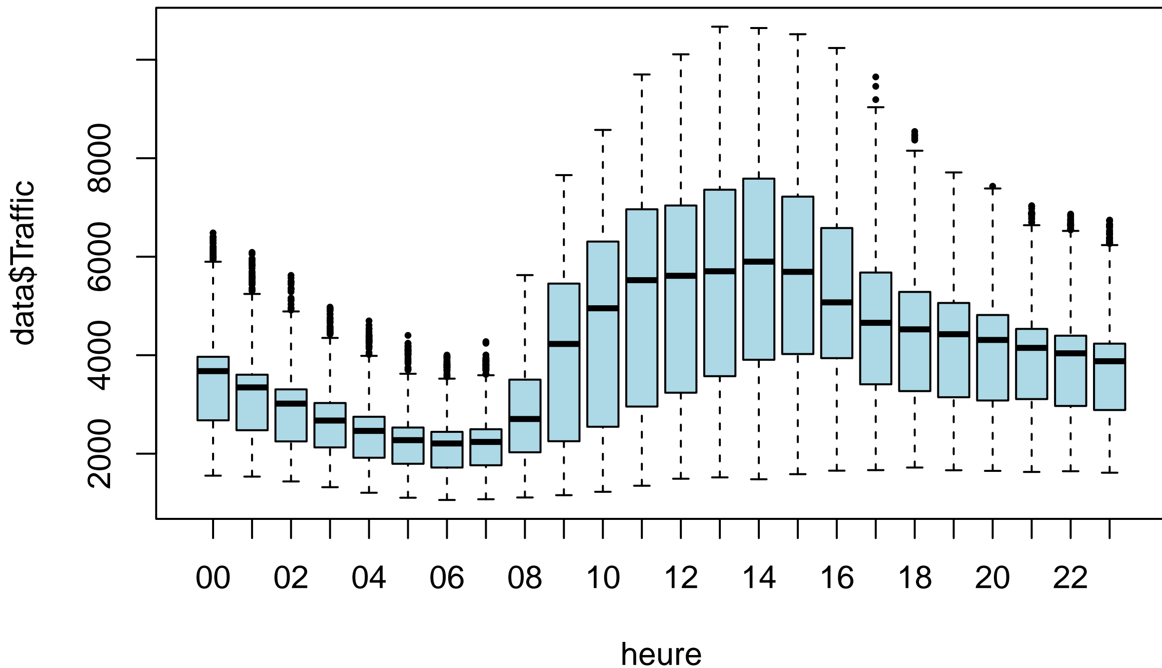
```
heure<-as.factor(format(data$Date, '%H'))
HourlyTraffic<-tapply(data$Traffic,heure,mean)
plot(HourlyTraffic,type='b',pch=20)
```



La fonction `format` permet ici d'extraire l'heure de la date et la fonction `tapply` d'appliquer une fonction à un vecteur par niveau de facteur.

Il est possible de décliner des boxplots par niveau de facteur:

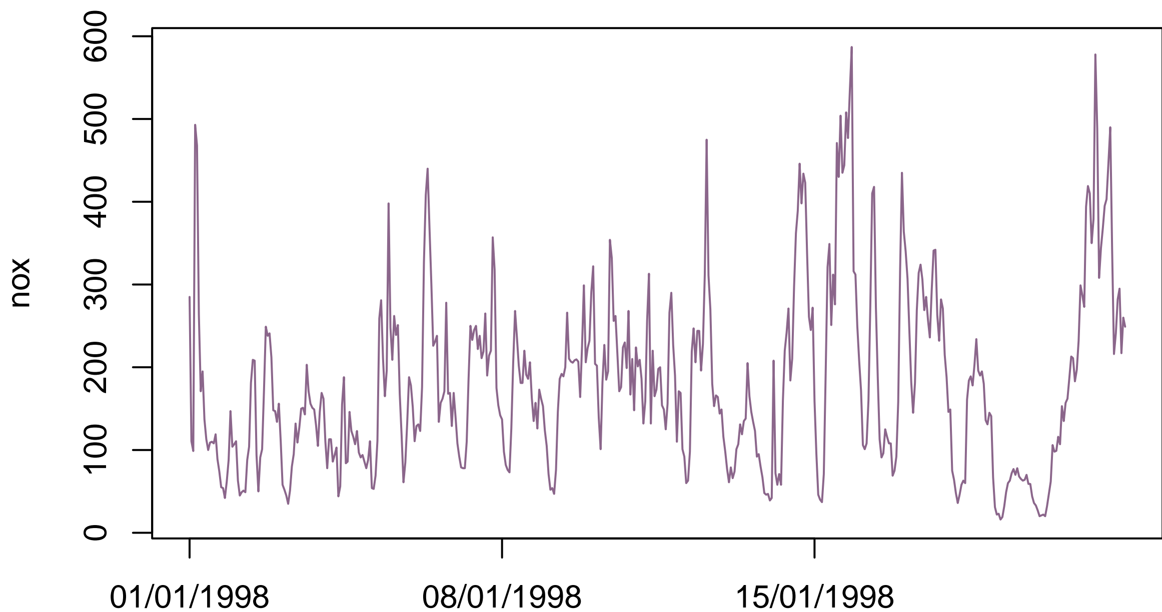
```
boxplot(data$Traffic~heure,col="lightblue",pch=20,cex=0.5)
```



Exemple 1: données de polution de l'air

Prenons l'exemple de données de polution de l'air mesurées à Londre à Marylebone Road (source: http://www.openair-project.org/). Les données sont stockées dans une table au format .RDS (les classes des variables ont déjà été définies). Il s'agit de données au pas horaire. Nous considérons ici les mesures de volume d'oxyde d'azote (nox: monoxyde d'azote plus dioxyde d'azote, voir: http://www.airparif.asso.fr/pollution/differents-polluants).

```
data<-readRDS("Data_cours/air_pollution_london_short.RDS")
plot(data$date,data$nox,type='l',xaxt="n",xlab='',ylab="nox",col="plum4")
axis.POSIXct(1, at = seq(data$date[1], tail(data$date,1),"weeks"), format="%d/%m/%Y",las=1)
```



Le calcul des statistiques de base sur cette série s'effectue ainsi:

```r
mean(data$nox)
```

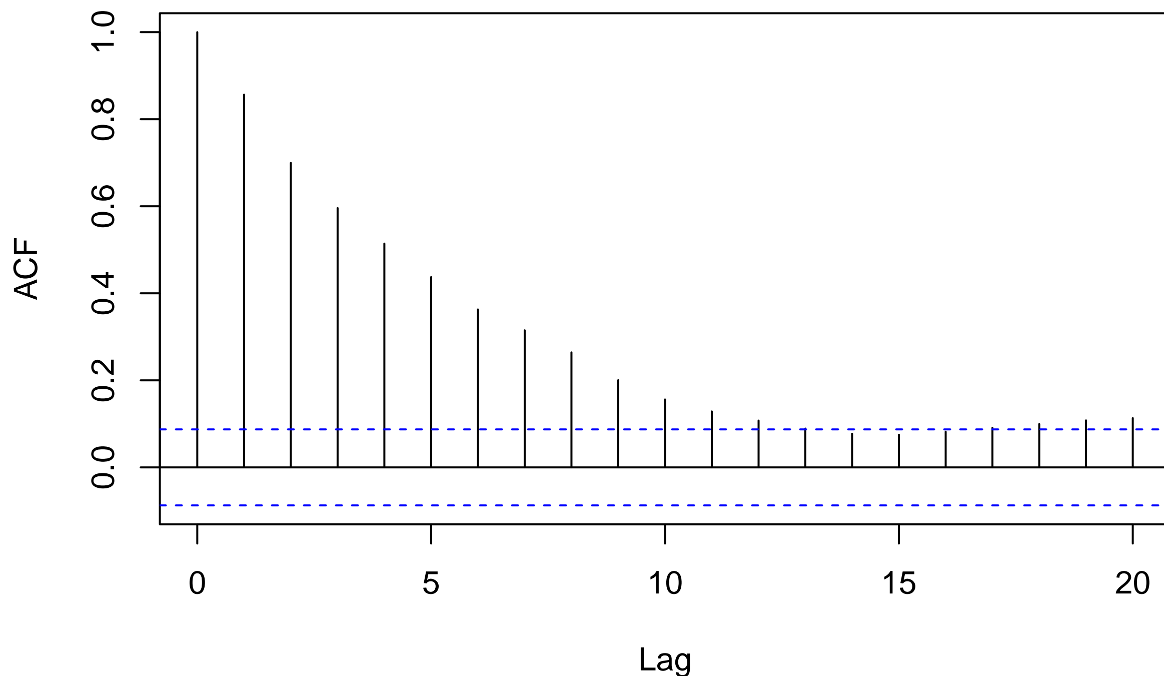
```
## [1] 179.7251
```

```r
sd(data$nox)
```

```
## [1] 108.3547
```

l'estimation de la fonction d'auto-corrélation se fait via la fonction `acf`, qui par défaut représente ces autocorrélations sur un graphique appelé l'autocorrélogramme.

```r
acf(data$nox,lag.max=20)
```

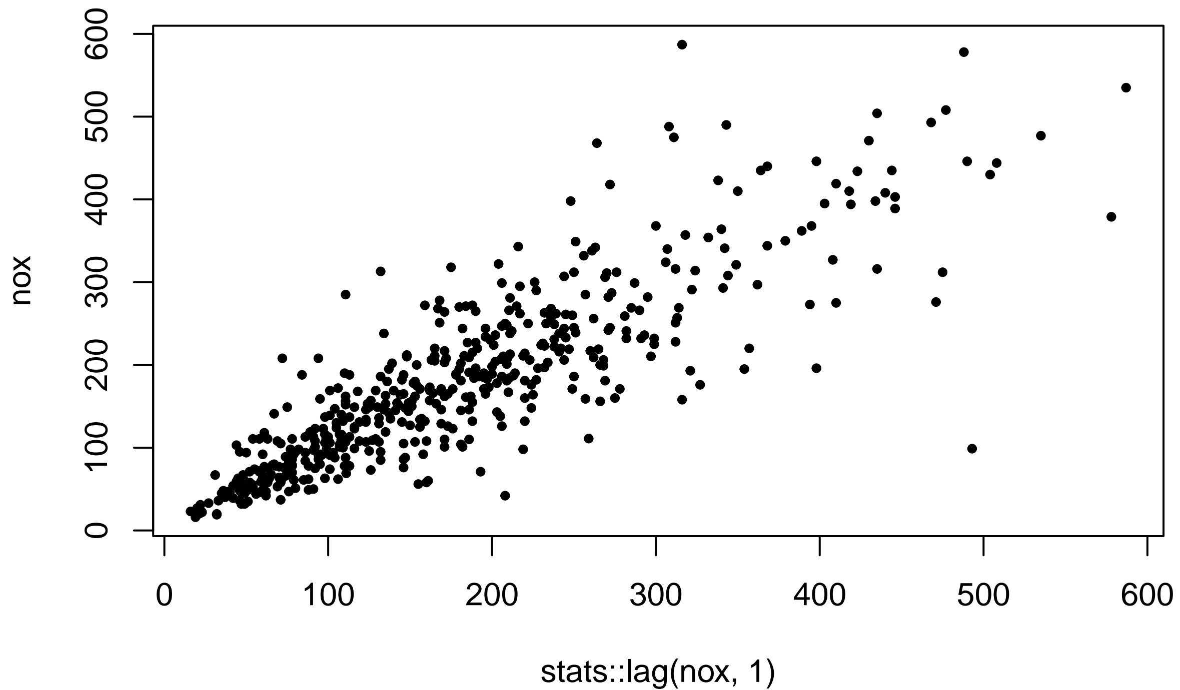



```r
acf(data$nox,lag.max=20,plot=FALSE)
```

```
##
## Autocorrelations of series 'data$nox', by lag
##
##     0     1     2     3     4     5     6     7     8     9    10    11    12
## 1.000 0.856 0.700 0.596 0.514 0.437 0.363 0.315 0.264 0.200 0.156 0.129 0.108
##    13    14    15    16    17    18    19    20
## 0.089 0.077 0.075 0.082 0.091 0.100 0.108 0.113
```

On peut constater une forte auto-corrélation d'ordre 1 (0.85). Le nuage de point de la série en fonction de la série retardée d'une heure le confirme:

```r
nox<-ts(data$nox)
plot(stats::lag(nox,1),nox,pch=20,cex=0.8)
```



Exemple 2: données de traffic internet

Reprenons les données de traffic internet et exploitons la classe `ts` pour leur analyse.

```
Traffic.ts<-ts(data$Traffic,start=1,frequency=24*12)
```

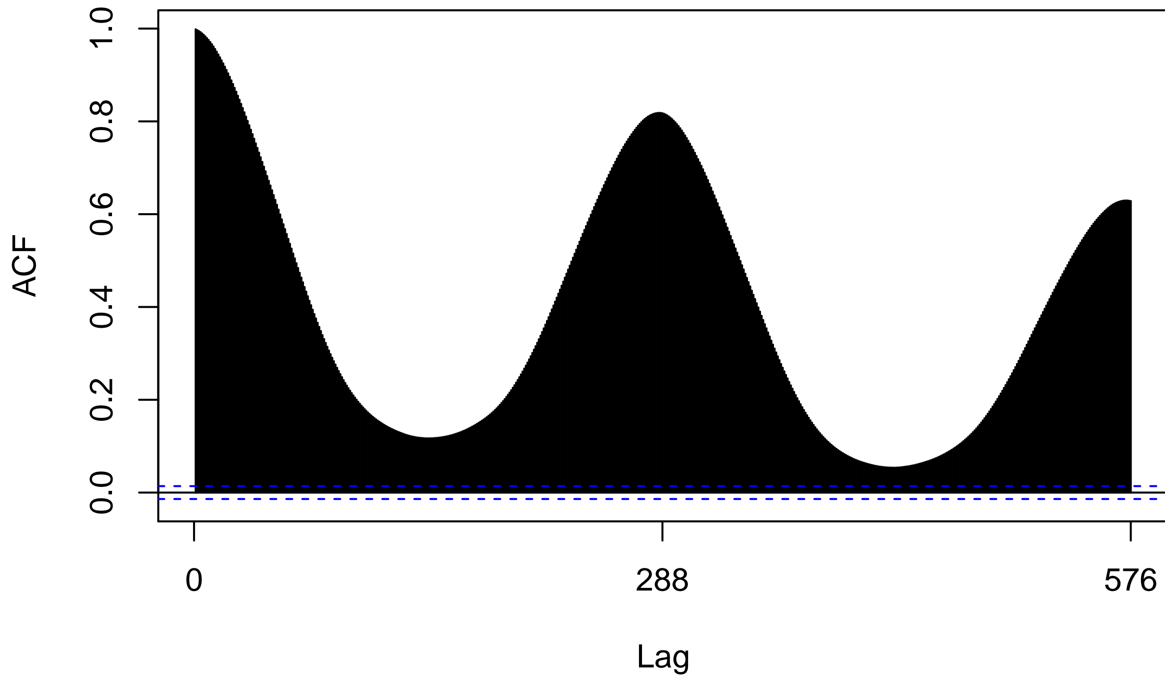
Une fois les données converties au format `xts`, la fonction `summary` permet d'obtenir les quartiles de la série:

```
summary(Traffic.ts)
```

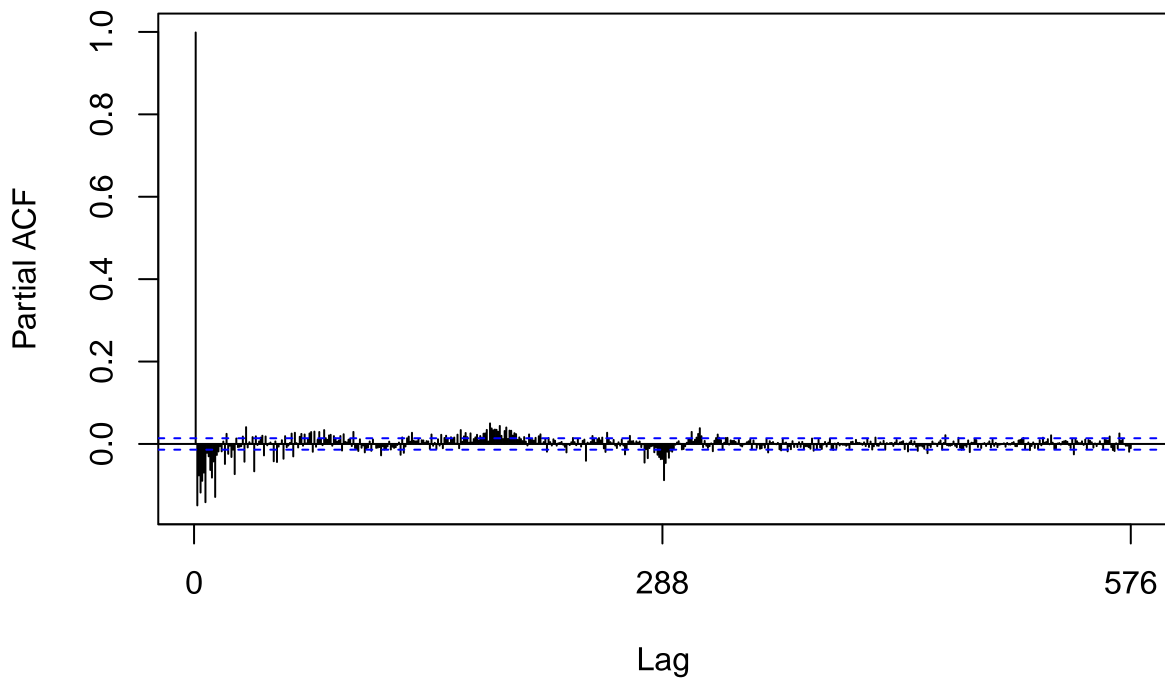
```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1060    2364    3494    3867    4889   10671
```

l'estimation de la fonction d'auto-corrélation se fait via la fonction `Acf` variante de la fonction R de base implémentée dans le package `forecast` et optimisée pour les objets `ts` et `mts`:

```
Acf(Traffic.ts)
```




```
Pacf(Traffic.ts)
```



Quelques exemples de sources de données:

- données de consommation/production électrique en europe: https://www.entsoe.eu, https://rte-opendata.opendatasoft.com/
- données macro-économiques/sociales: https://data.oecd.org/fr/energie.htm, insee
- données de fréquentation de station vélib. à Paris & Lyon: https://maxhalford.github.io/blog/openbikes-challenge/
- données météo: https://www.ncdc.noaa.gov/cdo-web/
- données de pollution atmosphérique: https://www.epa.gov/outdoor-air-quality-data, http://www.open air-project.org
- open data gouvernemental: http://www.data.gouv.fr/fr
- données de transport: http://www.data.gouv.fr/fr/datasets/trafic-annuel-entrant-par-station-2013/
- données financières: https://cran.r-project.org/web/packages/tidyquant/index.html
- plateforme de compétition de data science: https://www.kaggle.com, https://challengedata.ens.fr/, https://www.datascience.net/fr/home/, https://www.crowdanalytix.com/community