5 - Analyse Graphique

Séries Temporelles avec R - Initiation

Anna Smyk, Tanguy Barthelemy

Insee - Département des Méthodes Statistiques

Section 1

Introduction

Sommaire

Présentation de différentes foncionnalités graphiques à travers les packages :

- **{base}**
- **{ggplot2}**
- **{dygraphs}**
- **{highcharter}**

Overview

En **R**, il existe de multiples outils pour faire des graphiques. **R** est d'ailleurs connu et apprécié pour ses affichages et son dynamisme.

Dans cette séquence, nous allons présenter plusieurs packages R permettant de faire des graphiques.

Aussi, nous allons recentrer la thématique autour des graphiques de séries temporelles.

Section 2

base - plot

base - plot

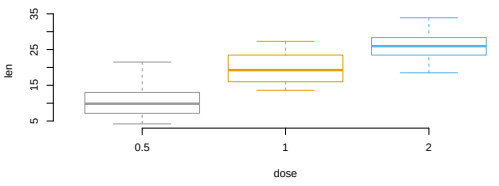
Lorsqu'on commence R, on utilise des syntaxes basiques, simples et qui ne requiert pas des connaissances aigues dans les packagesR.

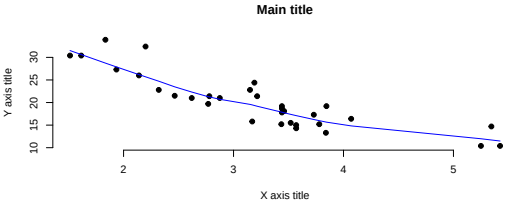
Ce sont les promesses de la fonction `plot()`.

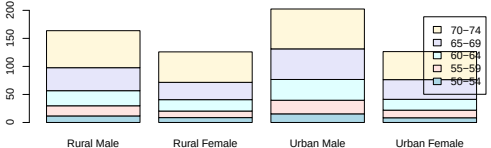
Souvent en formation R, on recommande d'apprendre **{ggplot2}**. Nous en parlerons un peu plus tard. Pour ma part, je continue régulièrement de faire des plots simples et efficaces.

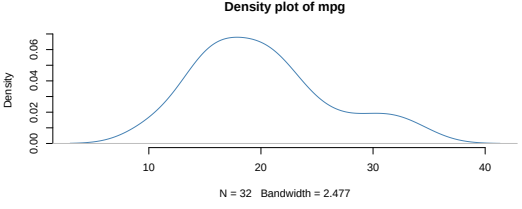
Les différents types de graphiques

Le packages **{base}** propose des graphiques très variés :









Plots de séries temporelles

Concernant les données temporelles, nous nous concentrerons sur les fonctions :

- `plot()`
- `lines()`
- `points()`
- `legend()`
- `par()`

Fonction `plot()`

C'est la fonction centrale des outils d'affichage de graphiques en R.

Cette méthode s'adapte au type d'objet qui est présenté (`data.frame()`, vecteur, série temporelle...). C'est grâce au système de méthode S3.

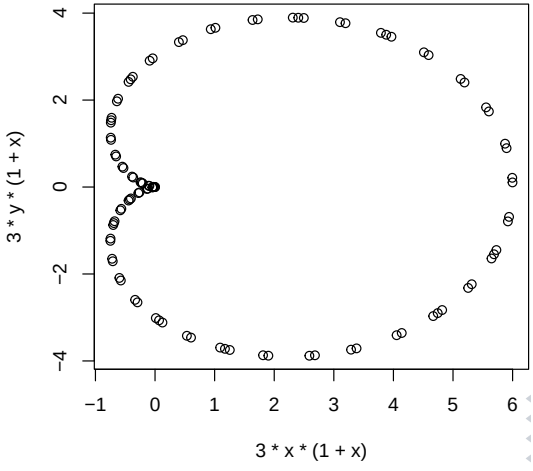
La syntaxe classique est

```
plot(x = observations, y = index)
```

avec `observations` sur l'axe des x et `index` sur l'axe des y.

Exemple

```
x <- sin(1:100)
y <- cos(1:100)
plot(x = 3 * x * (1 + x),
     y = 3 * y * (1 + x))
```



Arguments

La fonction `plot()` propose de multiples arguments pour personnaliser un graphique et ajouter des informations.

Les principaux arguments :

- `type` : type de trait (pointillé, ligne, points...)
- `main` : titre du graphique
- `xlab` et `ylab` : nom des axes
- `lwd` : largeur du trait
- `col` : couleur de la courbe

Pour les arguments suivants, reportez vous à la documentation de la fonction plot()

lines() et points()

Ces fonctions se comportent comme `plot()` à 2 différences près :

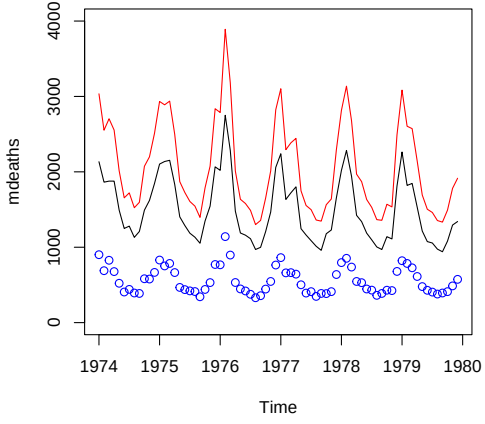
- Un nouvel appel ne créé pas de nouveau graphique

> **Nouveau graphique**
>
> Chaque appel à la fonction `plot()` créé un nouveau graphique. Les fonctions `lines()` et `points()` permettent d'ajouter des couches à un graphique déjà existant.

- Ces fonctions ne génèrent que des lignes ou des points.

Exemples

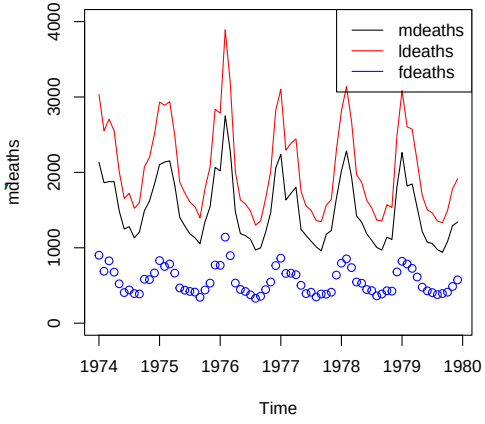
```
plot(mdeaths, ylim = c(0, 4000))
lines(ldeaths, col = "red")
points(fdeaths, col = "blue")
```



legend()

La fonction legend permet d'ajouter une légende au graphique.

```
plot(mdeaths, ylim = c(0, 4000))
lines(ldeaths, col = "red")
points(fdeaths, col = "blue")
legend(
    x = "topright",
    legend =c("mdeaths", "ldeaths", "fdeaths"),
    col = c("black", "red", "blue"),
    lwd = 1
)
```



par()

Enfin la fonction `par()` permet de mettre en forme les graphiques.

Par exemple :

- la disposition des figures,
- les polices d'écriture,
- la forme des axes,
- les marges...

Ces paramètres graphiques seront appliqués à toutes les figures.

Pros and cons

Les plots du packages de base ont plusieurs avantages :

- La syntaxe est simple et ils sont facilement réalisables.
- On sait exactement ce qui est affiché.
- Ils sont nuls en dépendance.

Mais il y a aussi des inconvénients

- Ils sont peu esthétiques.
- La personnalisation est plus compliquée.

Section 3

tidyverse - {ggplot2}

tidyverse - {ggplot2}

Pour se convaincre de la richesse de l'univers **{ggplot2}**, il suffit de jeter un oeil à la page The R Graph Gallery.

```
library("ggplot2")
```

Syntaxe et logique

Les figures **{ggplot2}** sont basés sur des couches vectorielles empilées les unes sur les autres et composant finalement notre graphique final.

Contrairement aux autres packages du tidyverse (qui utilise le pipe `%>%`), **{ggplot2}** propose d'empiler ses couches au moyen de l'opérateur `+`.

La syntaxe sera alors toujours plus ou moins la suivante :

```
ggplot(data = mydata, mapping = aes(myvariables)) +
    ...
```

Fonctions graphiques

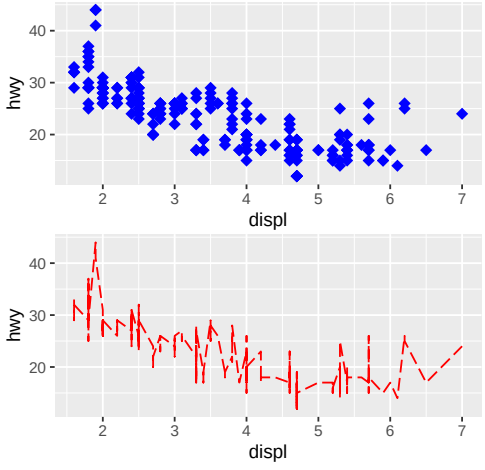
- `ggplot()` commence toujours la syntaxe. Il créé le cadre et spécifie le jeu de données utilisé.
- L'argument `aes` précise les variables utilisées pour le graphiques (x, y, couleurs, formes...)
- Les fonctions d'ajout d'éléments graphique s'appellent généralement `geom_XXX()` :
  - `geom_point()` pour tracer des points,
  - `geom_line()` pour des lignes,
  - `geom_bar()` pour des barres

Il existe d'autres fonctions pour afficher des objets (`geom_histogram()` pour un histogramme, `geom_boxplot()` pour un boxplot, `geom_density()` pour une densité...)

Exemple

```
ggplot(mpg, aes(x = displ,
                y = hwy)) +
    geom_point(shape = 18,
               size = 3,
               color = "blue")

ggplot(mpg, aes(x = displ,
                y = hwy)) +
    geom_line(color = "red",
              linetype = 5)
```



Séries temporelles avec {ggplot2}

Comme précisé précédemment, **{ggplot2}** ne travaille qu'avec des jeux de données donc `data.frame()` ou `tibble()`. Ainsi le code suivant ne fonctionnera pas et renverra une erreur :

```
ggplot(mdeaths)
```

```
Error in `fortify()`:
! `data` must be a <data.frame>, or an object coercible by `fortify()`,
  or a valid <data.frame>-like object coercible by `as.data.frame()`.
Caused by error in `.prevalidate_data_frame_like_object()`:
! `dim(data)` must return an <integer> of length 2.
```

Il faut donc adapter le code et nos objets pour travailler avec.

Travailler avec un jeu de données

Première idée : passer notre ts en jeu de données.

Un jeu de données est une forme de tableau *wide* avec 1 ligne par observation et toutes les colonnes sont des variable. Une colonne joue le rôle d'identifiant de l'observation.

Travailler avec un jeu de données

Ici les observations sont les périodes temporelles ainsi il faut ajouter une colonne `date` pour identifier chaque ligne.

```
mts_deaths <- cbind(mdeaths, ldeaths, fdeaths)
mts_date <- cbind(date = time(mdeaths), mdeaths, ldeaths, fdeaths)
head(mts_date, n = 4)
```

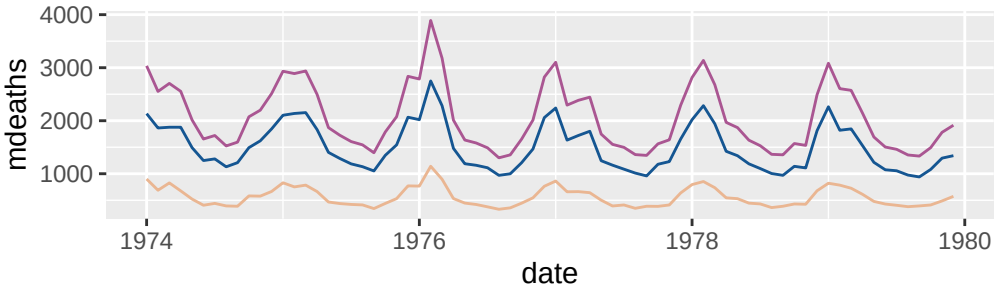
```
         date mdeaths ldeaths fdeaths
[1,] 1974.000    2134    3035     901
[2,] 1974.083    1863    2552     689
[3,] 1974.167    1877    2704     827
[4,] 1974.250    1877    2554     677
```

Il est aussi possible de travailler avec des objets `Date`.

Travailler avec un jeu de données

On peut maintenant afficher le graphique :

```
ggplot(data = mts_date, mapping = aes(x = date)) +
    geom_line(aes(y = mdeaths), color = "#155692") +
    geom_line(aes(y = ldeaths), color = "#AA5692") +
    geom_line(aes(y = fdeaths), color = "#EAB692")
```



mts, df ou tibble ?

Il est possible de convertir notre mts en data.frame ou tibble :

```
df_deaths <- as.data.frame(mts_date)
head(df_deaths)
```

```
      date mdeaths ldeaths fdeaths
1 1974.000    2134    3035     901
2 1974.083    1863    2552     689
3 1974.167    1877    2704     827
4 1974.250    1877    2554     677
5 1974.333    1492    2014     522
6 1974.417    1249    1655     406
```

```
library("tibble")
tibble_deaths <- as_tibble(mts_date)
head(tibble_deaths)
```

```
# A tibble: 6 x 4
   date mdeaths ldeaths fdeaths
  <dbl>   <dbl>   <dbl>   <dbl>
1 1974     2134    3035     901
2 1974.    1863    2552     689
3 1974.    1877    2704     827
4 1974.    1877    2554     677
5 1974.    1492    2014     522
6 1974.    1249    1655     406
```

Mais le résultat graphique sera le même.

Travailler avec un tsibble

Il est aussi possible de travailler avec des objets `tsibble`.

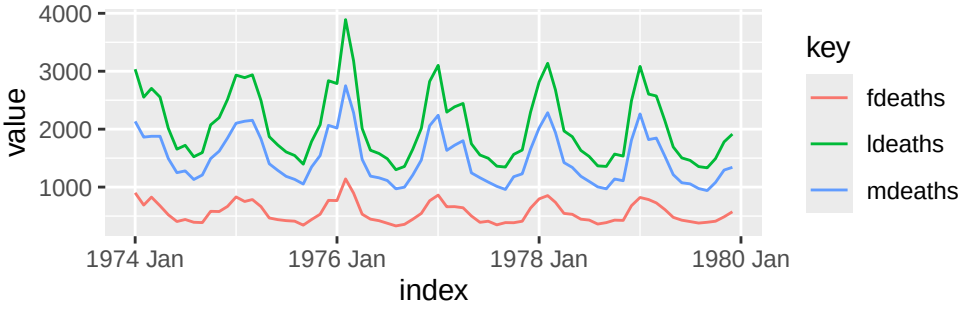
```
library("tsibble")
tsibble_deaths <- as_tsibble(mts_deaths)
head(tsibble_deaths, n = 4)
```

```
# A tsibble: 4 x 3 [1M]
# Key:       key [1]
     index key     value
     <mth> <chr>   <dbl>
1 1974 Jan mdeaths  2134
2 1974 Feb mdeaths  1863
3 1974 Mar mdeaths  1877
4 1974 Apr mdeaths  1877
```

Travailler avec un tsibble

Le graphique se compose alors plus simplement :

```
ggplot(data = tsibble_deaths,
       mapping = aes(x = index, y = value, color = key)) +
    geom_line()
```



Pros and cons

{ggplot2} offre des fonctionnalités intéressantes pour la création de graphiques :

- Figures très variés
- Belles couleurs et beau graphisme
- Grande communauté donc plein de documentation et d'aide

Mais il y a aussi des inconvénients

- La syntaxe de **{ggplot2}** est très spécifique et (pour ma part) il y a toujours obligation de s'y replonger à chaque fois pour chaque détail graphique.
- Les données doivent être formattées d'une certaine façon.
- Cela impose aussi d'ajouter une dépendance à notre projet.

Section 4

{dygraphs}

{dygraphs}

Si vous cherchez du dynamisme dans vos graphiques, le package **{dygraphs}** est sûrement fait pour vous.

La documentation est un tutoriel qui initie et fait une démonstration à l'univers dygraphs. Une partie de The R Graph Gallery est dédiée à l'utilisation du package.

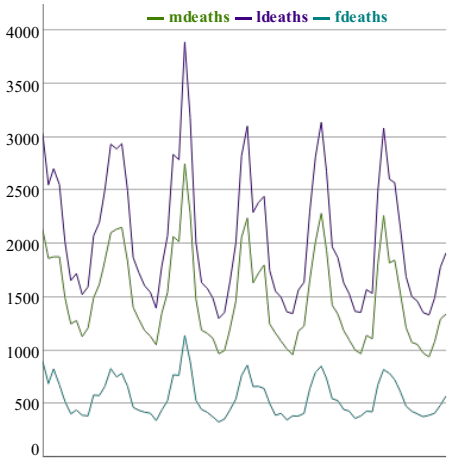
Utilisation

Ce package est spécialisé dans l'affichage de données temporelles. La syntaxe est une syntaxe proche du tidyverse avec l'utilisation de pipe %>% entre les fonctions (qui commencent toutes par dy ...).

```
library("dygraphs")
```

Exemple

```
dygraph(mts_deaths)
```



Personnalisation

{dygraphs} propose des fonctions supplémentaires pour personnaliser vos graphiques :

- `dyOptions()` : Certains paramètres graphiques
- `dySeries()` : Ajout de séries supplémentaires
- `dyRangeSelector()` : selection de zoom sur le graphique
- `dyAxis()` : paramètres des axes

Pros and cons

Pour les séries temporelles, **{dygraphs}** offre un grand éventail d'affichages :

- Graphiques clairs
- Syntaxe très compréhensible
- Personnalisation facile grâce au fichiers de démonstration

Mais il y a aussi des inconvénients

- Graphiques uniquement liés aux séries temporelles
- Cela impose aussi d'ajouter une dépendance à notre projet

Section 5

{highcharter}

{highcharter}

Le package **{highcharter}** propose aussi des fonctionnalités de dynamisme. Je ne rentre pas dans les détails.

La documentation du package.

Section 6

rjdverse

rjdverse

Le rjdverse est l'ensemble des packages liés à JDemetra+. Il existe 2 packages

Version 2 :

- **{ggdemetra}**

Version 3 :

- **{ggdemetra3}**

{ggdemetra} et {ggdemetra3}

Ces 2 packages sont développés par Alain Quartier-la-Tente. Ils permettent d'implémenter les méthodes ggplot2 sur les modèles d'ajustement saisonnier de JDemetra+.

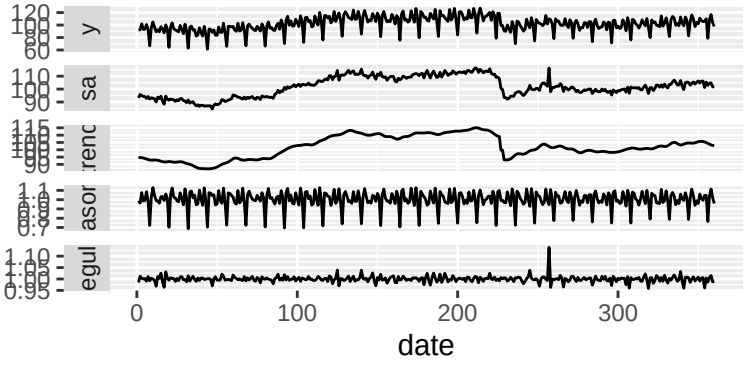
```
library("ggdemetra")
library("ggdemetra3")
```

Version 2 - {ggdemetra}

Les fonction s'appellent `geom_XXX` D'autres fonctions comme `autoplot`, `siration`, `trendcycle`... sont implémentées.

Exemple

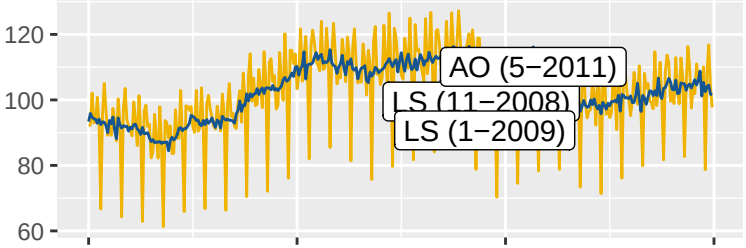
```
x <- RJDemetra::jx13(ipi_c_eu[, "FR"])
autoplot(x)
```



geom_XXX()

```
ggplot(data = ipi_c_eu_df, mapping = aes(x = date, y = FR)) +
    geom_line(color =  "#F0B400") +
    labs(title = "Seasonal adjustment of the French industrial production ind
         x = "time", y = NULL)  +
    geom_sa(color = "#155692", message = FALSE) +
    geom_outlier(geom = "label", message = FALSE)
```

Seasonal adjustment of the French industrial p



Section 7

Exercices pratiques

Exercices pratiques

Exercice 1 : Créer votre premier graphique **{dygraphs}**. Exercice 2 : Essayer d'ajuster avec les fonctions de **{ggdemetra}** le jeu de données `AirPassengers`.

Section 8

Références

Références

Documentation utiles :

- Les chapitres 8 et 9 du Guide d'introduction par le r-project
- Doc JS de dygraphs
- Le R Graphics Cookbook
- la documentation de ggplot2