Lecture 4: Spatial data visualization using terra

Geovisualization

Joaquin Cavieres

1. Introduction

In this class we will see a spatial data visualization using the package "terra". When we say "spatial data", we are talking about geographical locations in a spatial domain, for example, places on earth. In strict rigor this type of data should be called "geospatial data" but commonly is named as "spatial data".

To start you have to install the package "terra" writing: `install.packages("terra")`.

2. Spatial data

The spatial phenomena generally can be considered as discrete objects with clear boundaries or continuous phenomena that can be observed everywhere. Some examples of this:

- Discrete spatial objects: river, road, country, town, research site,
- Continuous phenomena (also known as "spatial fields"): elevation, temperature, air quality.

Spatial data commonly are represented by vector data, this mean, a description of the geometry or shape of those objects. Vector data models can represent all types of features with high accuracy, and for this, points, lines, and polygons are accurate when we want to define the location and size of all topographic features. Examples of a vector data could be:

a) A city may be represented by points
b) A road may be represented by a collection of lines
c) A state may be represented as a polygon

but, what is a vector data? here a little definition.

2.1. Vector data

The main vector data types can be represent by points, lines and polygons. In all of them, the geometry of those data structures consists of sets of coordinate pairs (x, y) (Longitude-Latitude or Easting-Northing).

Points are the simplest case. Each point has one coordinate pair, and n associated variables. For example, a point might represent a place where a rat was trapped, and the attributes could include the date it was captured, the person who captured it, the species size and sex, and information about the habitat.

Lines are a little more complex.

The base graphics function to create a plot in R is the `plot()` function. However, the exact function being called will depend upon the parameters used. In simple terms, `plot()` function plots two vectors against each other.

The syntax to use the `plot()` is the following:

```
plot(x,y,type,main,xlab,ylab,pch,col,las,bty,bg,cex,...),
```

where the parameters (arguments) of the functions are:

Parameter	**Description**
x	The coordinates of points in the plot
y	The y coordinates of points in the plot
type	The type of plot to be drawn
main	An overall title for the plot
xlab	The label for the x axis
ylab	The label for the y axis
pch	The shape of points
col	The foreground color of symbols as well as lines
las	The axes label style
bty	The type of box round the plot area
bg	The background color of symbols (only 21 through 25)
cex	The amount of scaling plotting text and symbols
...	Other graphical parameters

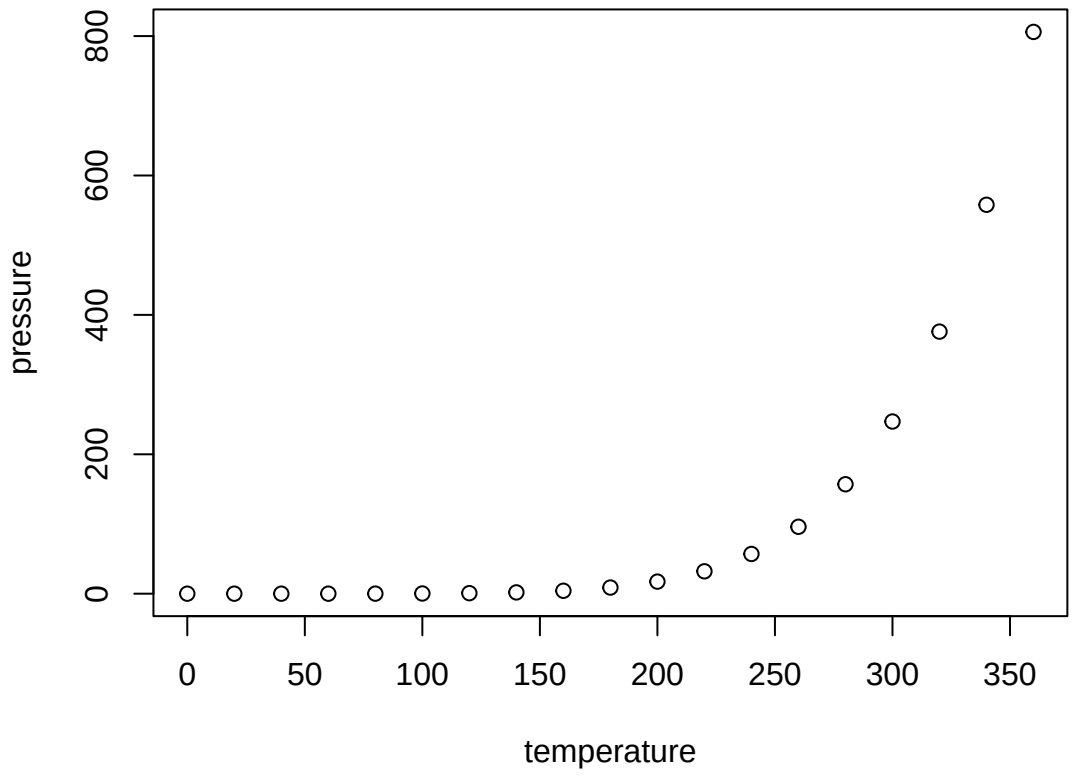
To create a simple plot with this function, we need a set of data to work with. So, let's consider the built-in pressure dataset as an example. It contains observations of the vapor pressure of mercury over a range of temperatures.

```
# First six observations of the 'Pressure' dataset
head(pressure)
```

```
##   temperature pressure
## 1           0   0.0002
## 2          20   0.0012
## 3          40   0.0060
## 4          60   0.0300
## 5          80   0.0900
## 6         100   0.2700
```

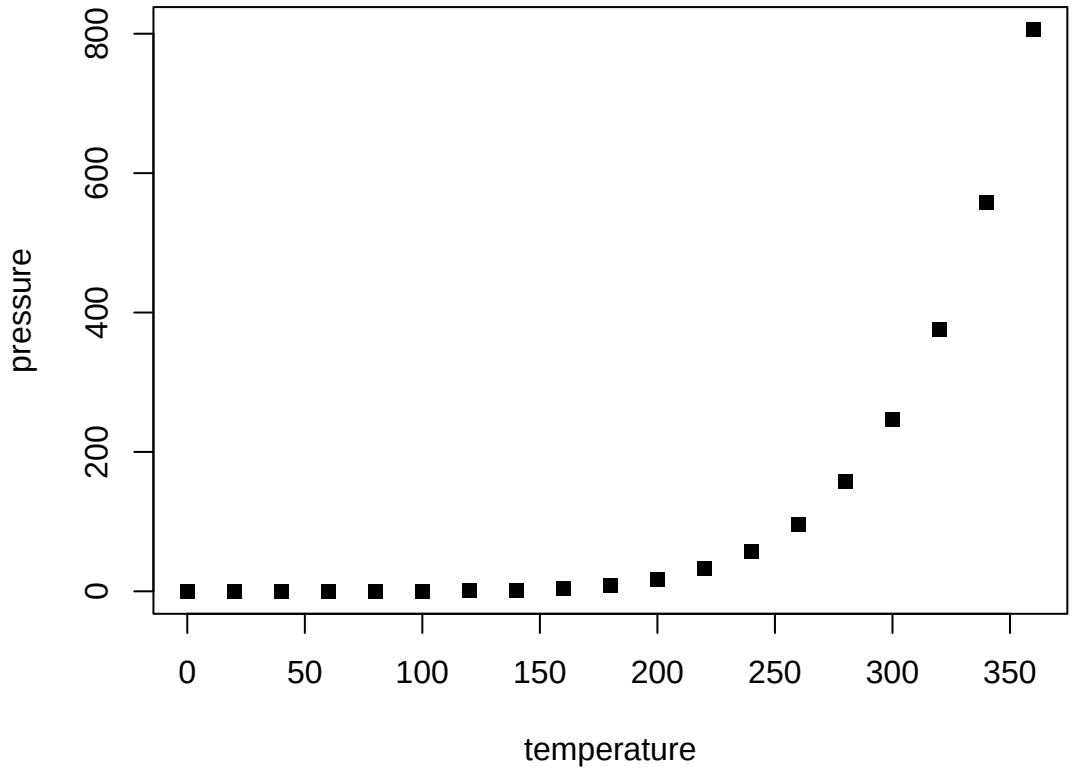
So, we can create the plot doing:

```
plot(pressure)
```



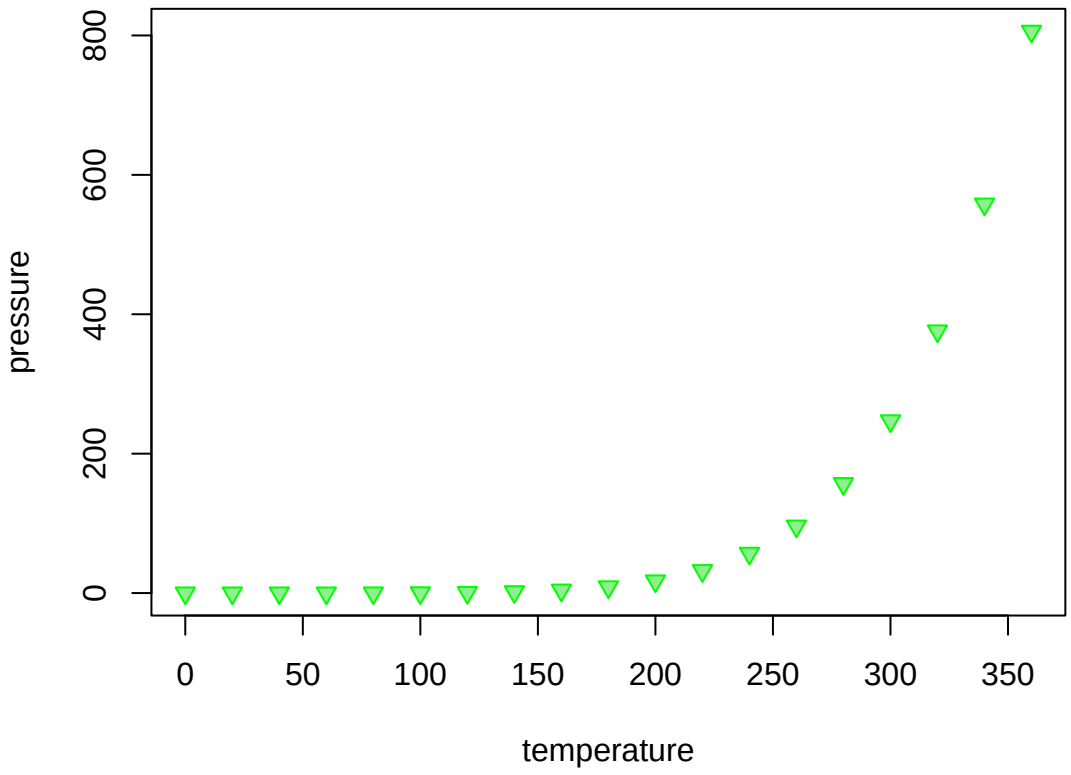
Now we will start adding characteristics to our plot, first changing the **shape** and **size** of the points. In this link of sthda.com you can see the different types of symbols to specify the points, for example:

```
plot(pressure, pch=15)
```



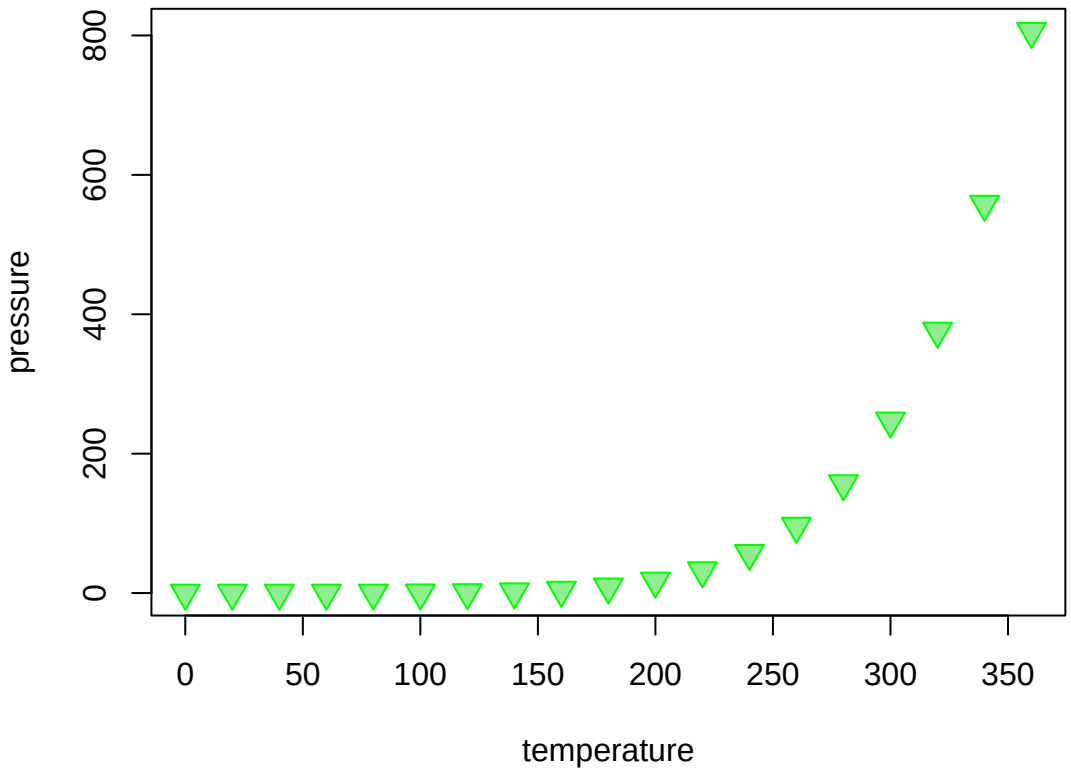
You can specify border color using `col` argument and fill color using `bg` argument.

```
plot(pressure, pch=25, col="green", bg="lightgreen")
```



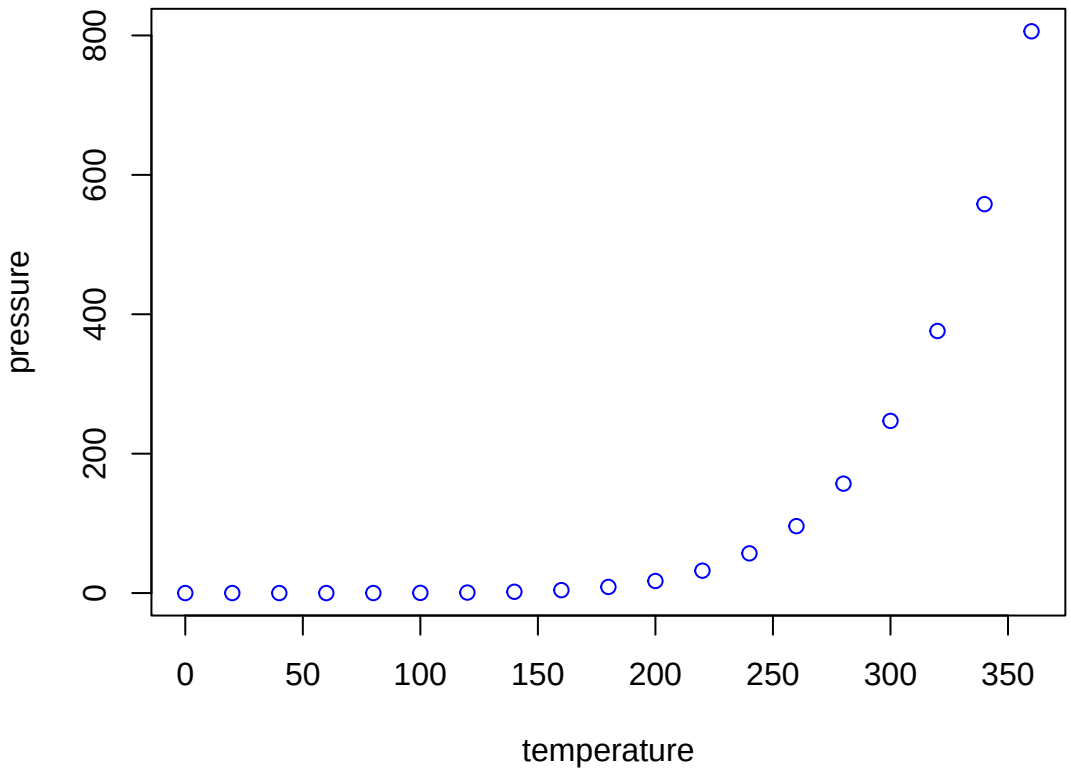
To change the size of the plotted characters, use `cex` (character expansion) argument.

```
plot(pressure, pch=25, col="green", bg="lightgreen", cex = 1.5)
```



We also can change the color of the plot using the argument `col`, for example:

```
plot(pressure, col="blue")
```



In R there are a predefined number of colors that you can use, but a more complete pallete of colors is this link: http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf.

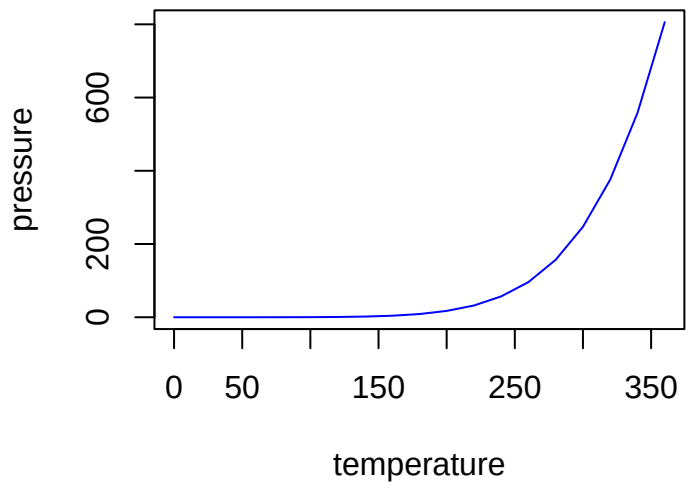
1.2. Different types of plot

You can change the type of plot changing the `type` argument, for example here there is a list of some type of plots that you can make.

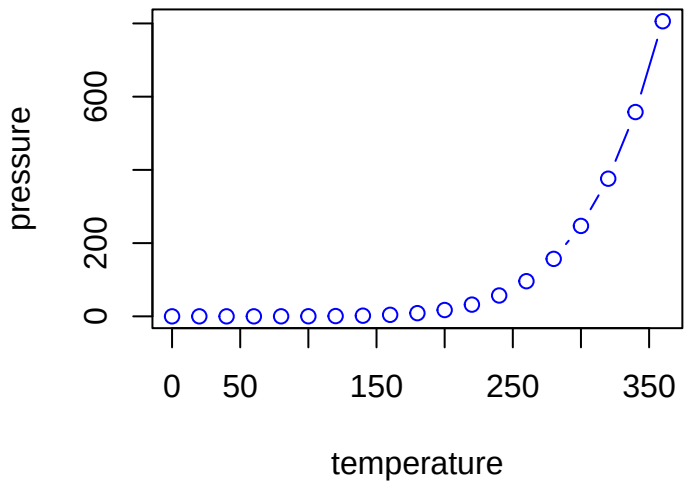
Parameter	Description
'p'	Points
'l'	Lines
'b'	Both points and lines
'c'	The lines part alone of b
'o'	Both points and lines overplotted
'h'	Histogram like (or high-density) vertical lines
's'	Step plot (horizontal first)
'S'	Step plot (vertical first)
'n'	No plotting

Thus, if we want to create a plot with lines between data points, use `type = 'l'`, to draw both lines and points, use `type = 'b'`

```
plot(pressure, col="blue", type = "l")
```




```
plot(pressure, col="blue", type = "b")
```



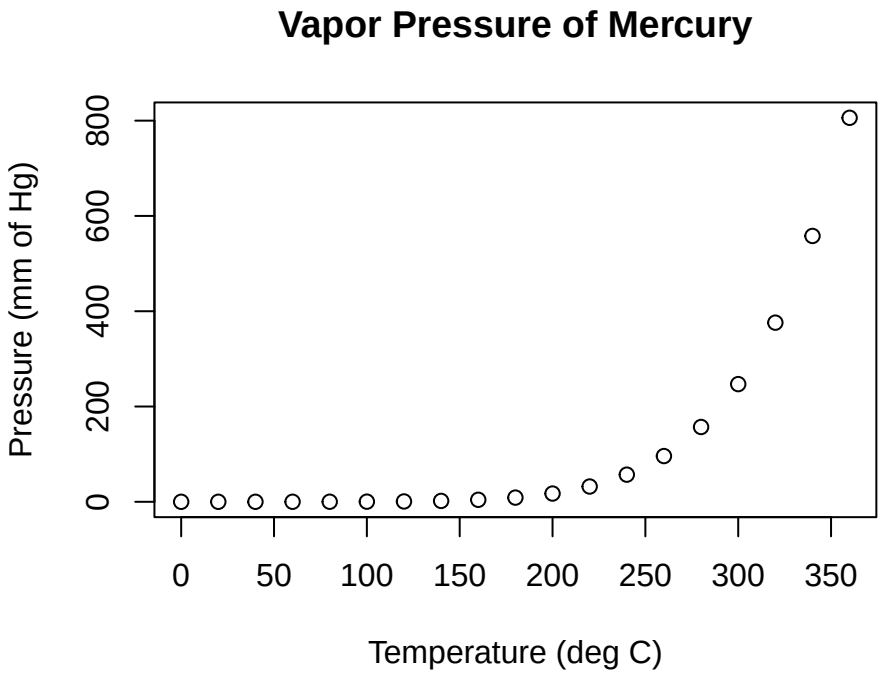
1.3. Adding titles and axis labels

You can create your own title and change the names for axis labels easily by specifying:

Argument	Description
`main`	Main title in the plot
`xlab`	x-axis label
`ylab`	y-axis label

So, for our plot:

```
plot(pressure,
     main = "Vapor Pressure of Mercury",
     xlab = "Temperature (deg C)",
     ylab = "Pressure (mm of Hg)")
```



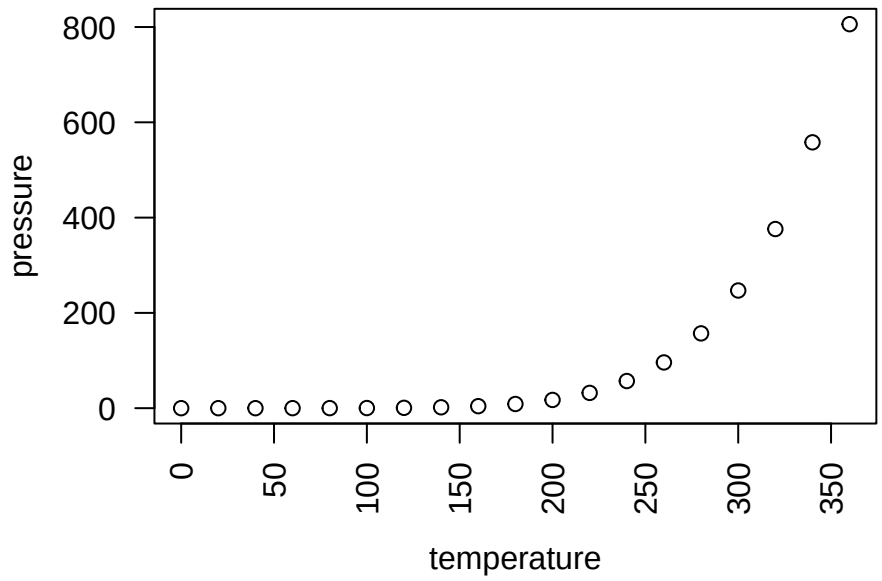
1.4. The axes label style

If you want to change the axes label style, it can be done by specifying the `las` (label style) argument. With this argument you can change the orientation angle of the labels.

Argument	Description
0	The default, parallel to the axis
1	Always horizontal
2	Perpendicular to the axis
3	Always vertical

For example,

```
plot(pressure, las = 2)
```

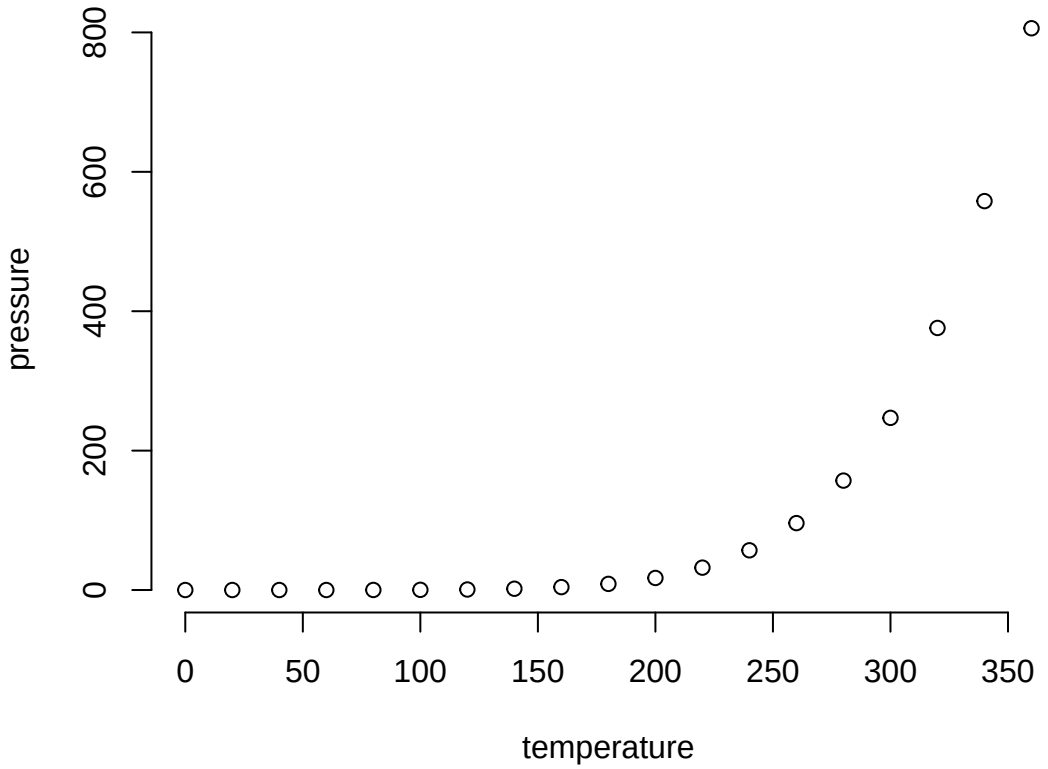


1.5. The box type

Changing `bty` (box type) argument to change the type of box.

Argument	Description
'o'	Draws a complete rectangle around the plot (default)
'n'	Draws nothing around the plot.
'l', '7', 'c', 'u', ']'	Draws a shape around the plot area.

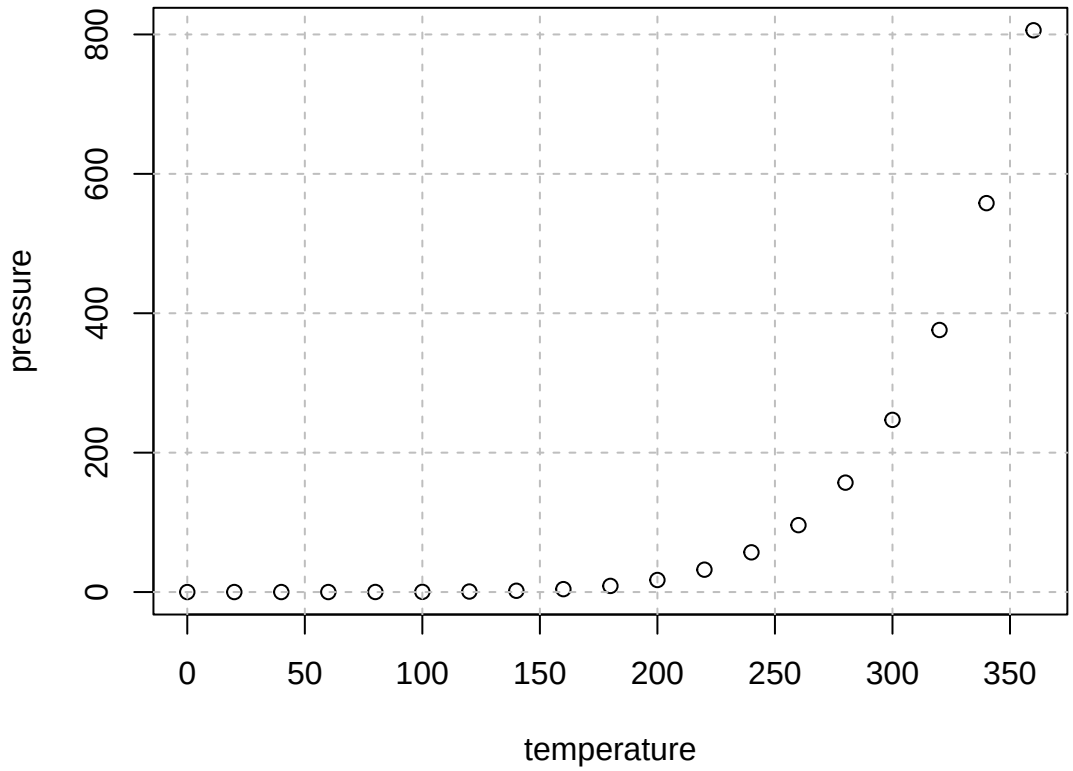
```
plot(pressure, bty="n")
```



1.6. Add a grid

You can add a grid calling the function `grid()` after of `plot()` function, for example:

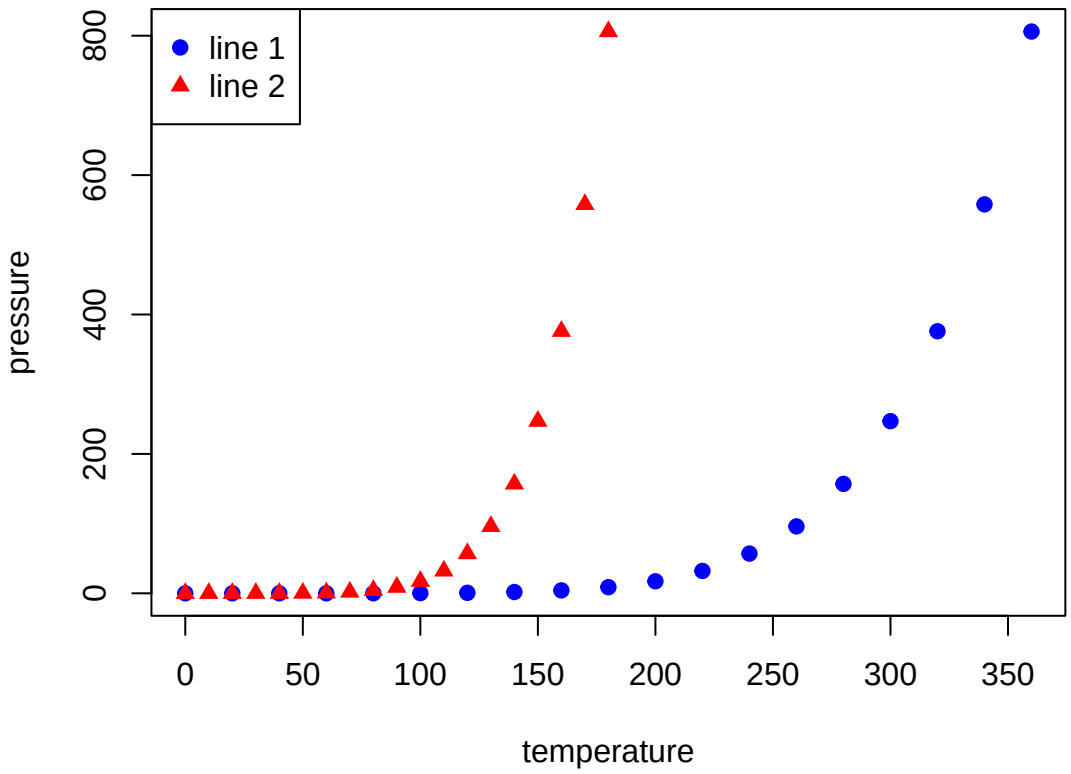
```
plot(pressure)
grid(col = "gray", lty = 2)
```



1.7. Add a legend

You can include a legend to your plot – a little box that decodes the graphic for the viewer - calling the `legend()` function after of the `plot()`.

```
plot(pressure, col="blue", pch=19)
points(pressure$temperature/2, pressure$pressure,col="red", pch=17)
legend("topleft", c("line 1","line 2"), pch=c(19,17), col=c("blue","red"))
```

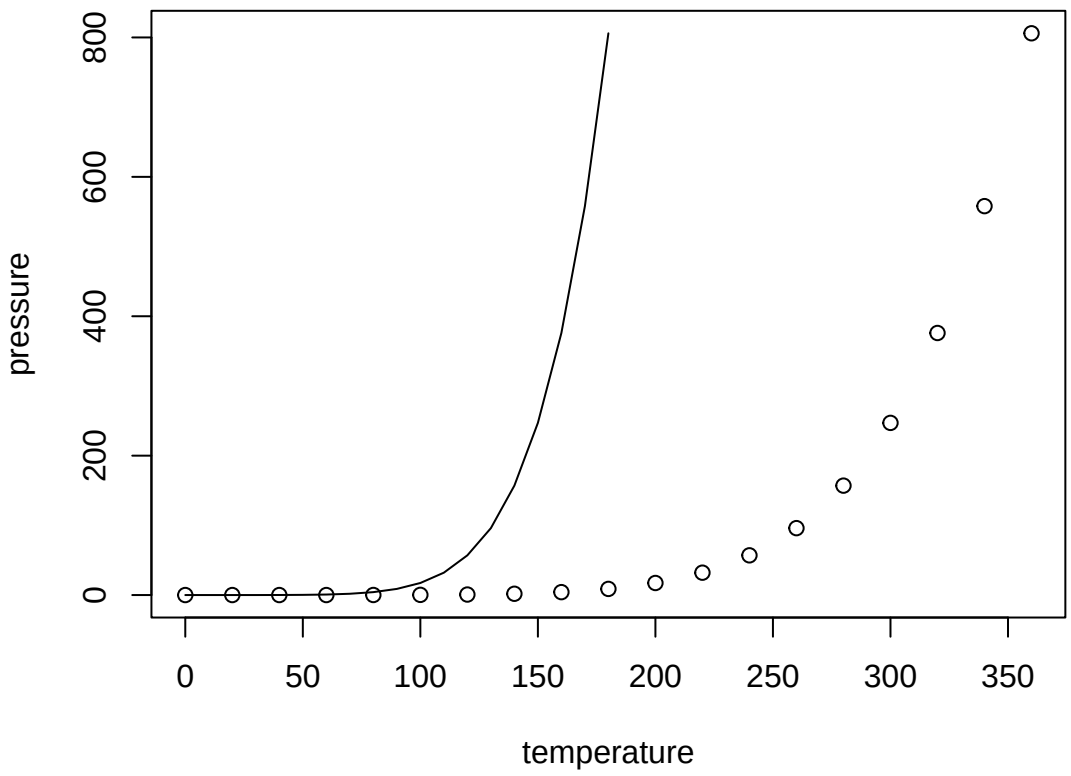


You also can change the position of the legend modifying the following keywords: 'bottomright', 'bottom', 'bottomleft', 'left', 'topleft', 'top', 'topright', 'right' and 'center'.

1.8. Add lines to a plot

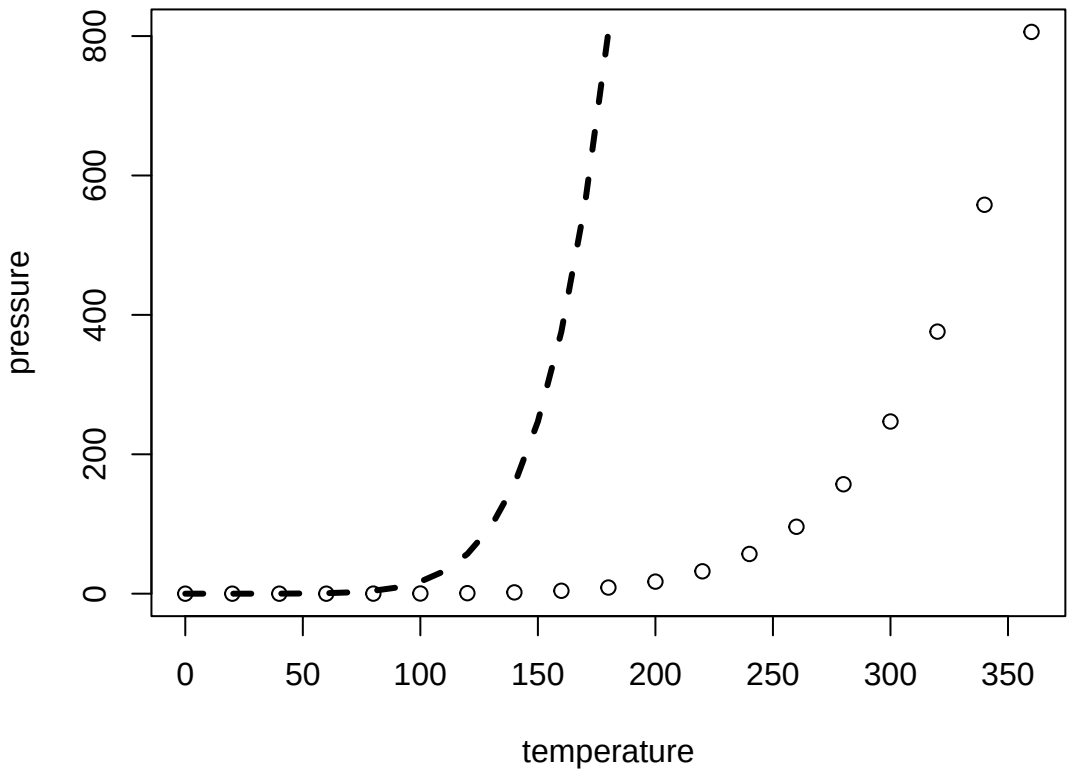
Adding lines to a plot is very similar to adding points, only you have to write the function `lines()`.

```
plot(pressure)
lines(pressure$temperature/2, pressure$pressure)
```



and you can change the type of lines adding the argument `lty`, or the width of the line using the `lwd` argument as:

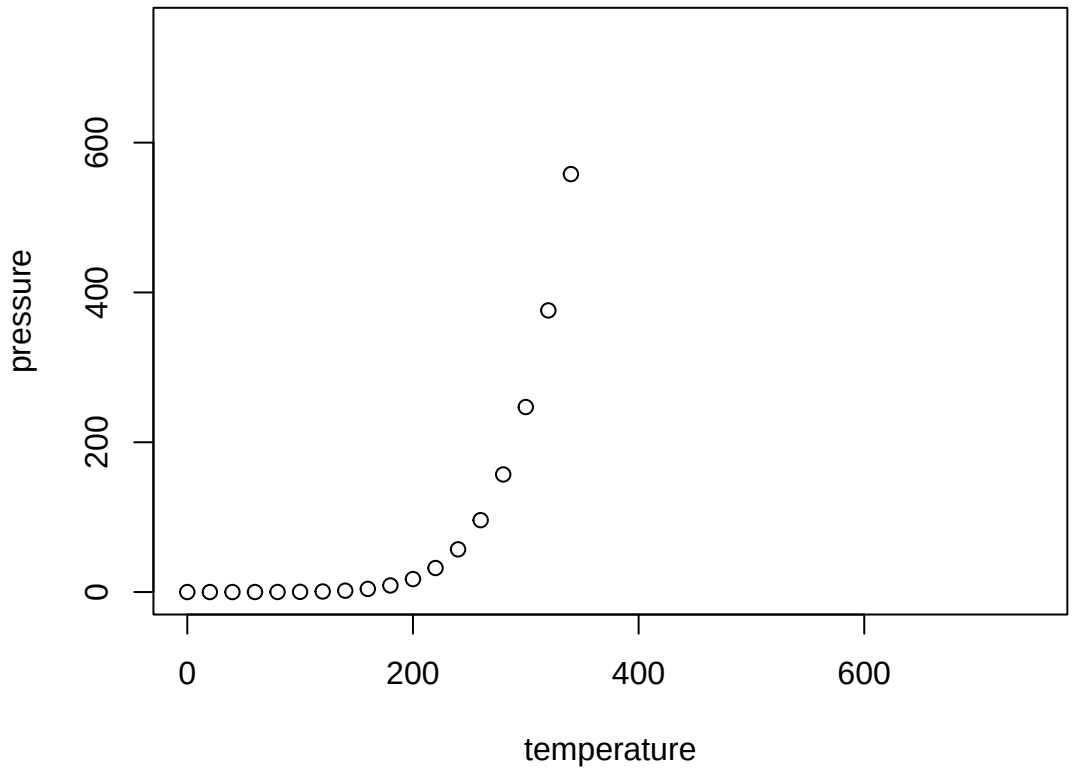
```
plot(pressure)
lines(pressure$temperature/2, pressure$pressure, lwd=3, lty=2)
```


pressure
0
200
400
600
800
0
50
100
150
200
250
300
350
temperature

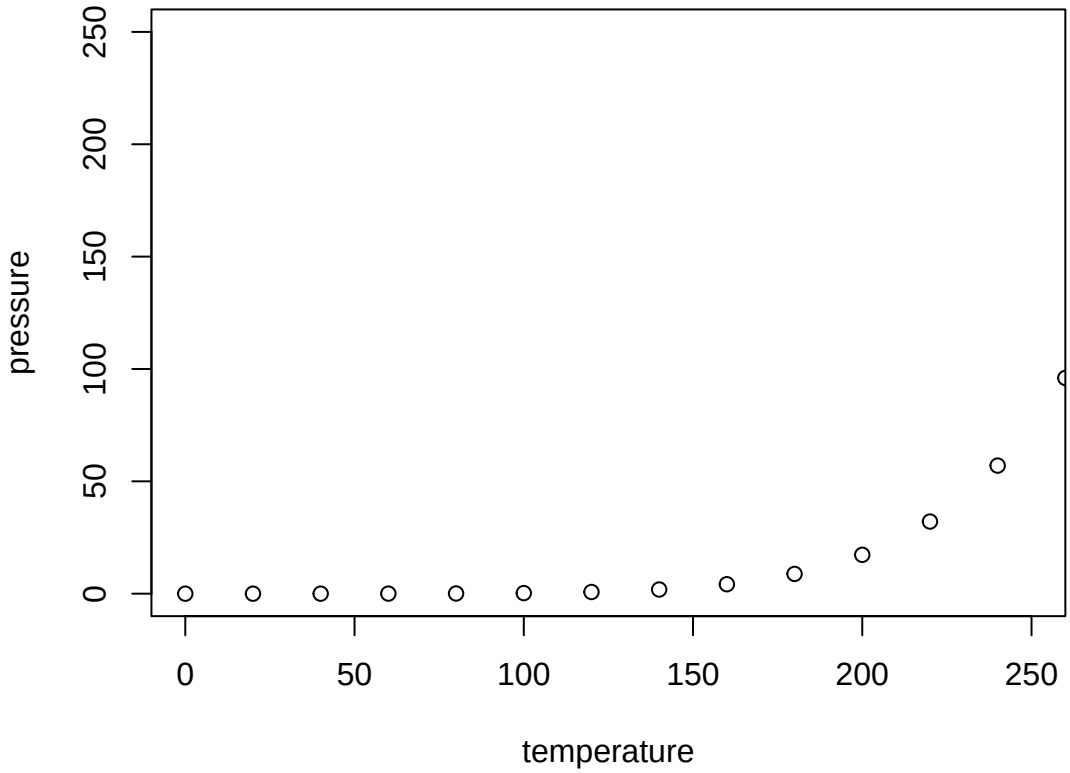
1.9. Limits for the axis

The `plot()` function works with the best size and scale of each axis to fit the plot area. However, you can change the limits of the axis using the functions `xlim` and `ylim` arguments.

```
plot(pressure, ylim=c(0, 750), xlim=c(0, 750))
```



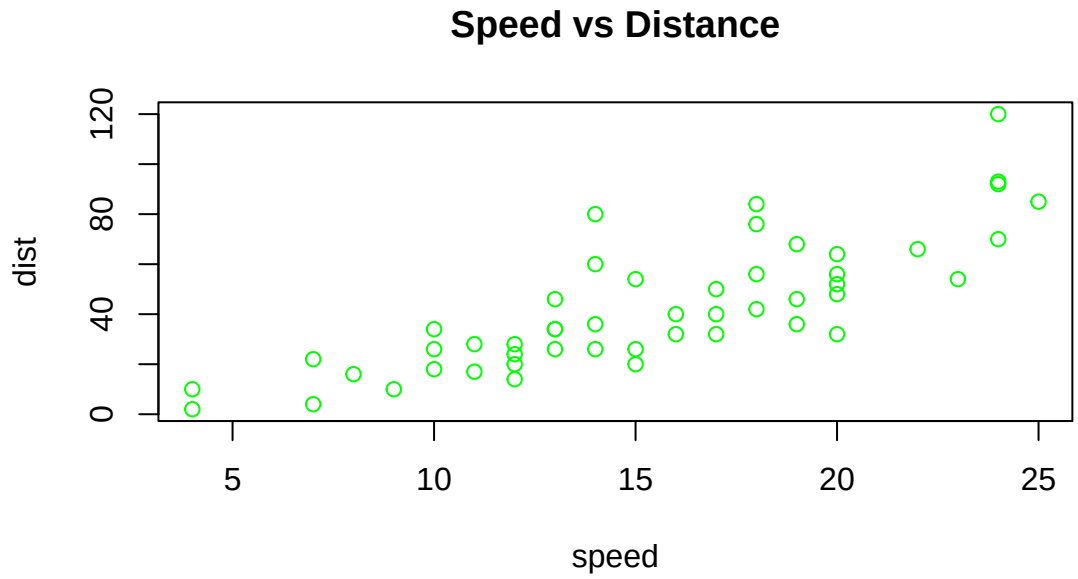

```
plot(pressure, ylim=c(0, 250), xlim=c(0, 250))
```

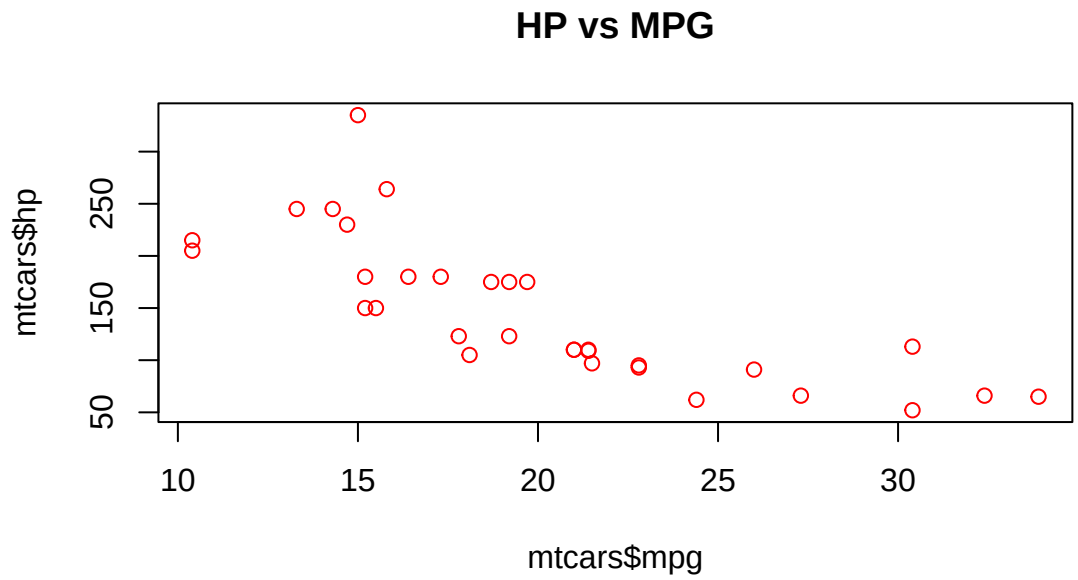

pressure
0
50
100
150
200
250
0
50
100
150
200
250
temperature

1.10. Multiple plots in a single page

We can display multiple plot in a single page using the function `mfrow`, for example:

```
par(mfrow = c(2, 1))
plot(cars, main="Speed vs Distance", col="green")
plot(mtcars$mpg, mtcars$hp, main="HP vs MPG", col="red")
```





Once your plot is complete, you need to reset your `par()` options. Otherwise, all your subsequent

plots will appear one below the other.

```
par(mfrow = c(1,1))
```

2. Visualizing time series data

2.1. Introducing to exploratory time series data

For this purpose we will use the "Nile" data set.

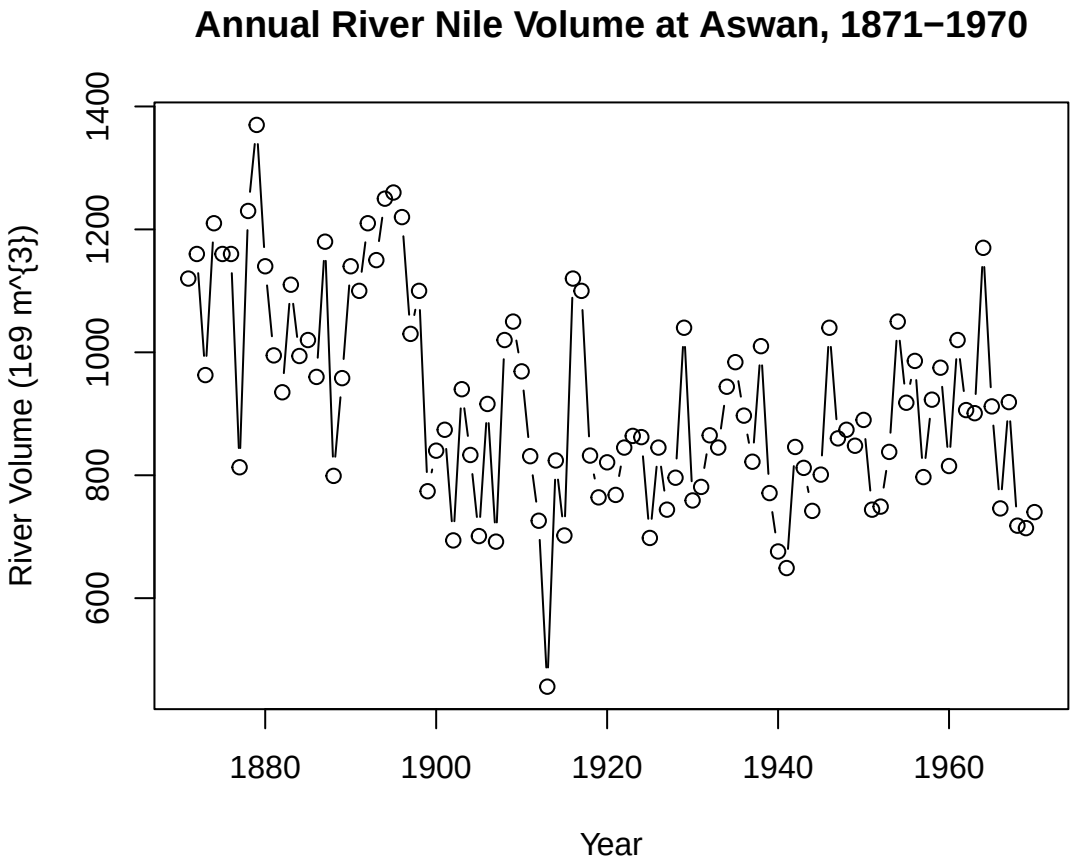
```
# Print the Nile dataset
print(Nile)
```

```
## Time Series:
## Start = 1871
## End = 1970
## Frequency = 1
##   [1] 1120 1160  963 1210 1160 1160  813 1230 1370 1140  995  935 1110  994 1020
##  [16]  960 1180  799  958 1140 1100 1210 1150 1250 1260 1220 1030 1100  774  840
##  [31]  874  694  940  833  701  916  692 1020 1050  969  831  726  456  824  702
##  [46] 1120 1100  832  764  821  768  845  864  862  698  845  744  796 1040  759
##  [61]  781  865  845  944  984  897  822 1010  771  676  649  846  812  742  801
##  [76] 1040  860  874  848  890  744  749  838 1050  918  986  797  923  975  815
##  [91] 1020  906  901 1170  912  746  919  718  714  740
```

2.2. Basic time series plots

Here we can use the basic arguments used in the `plot()` function.

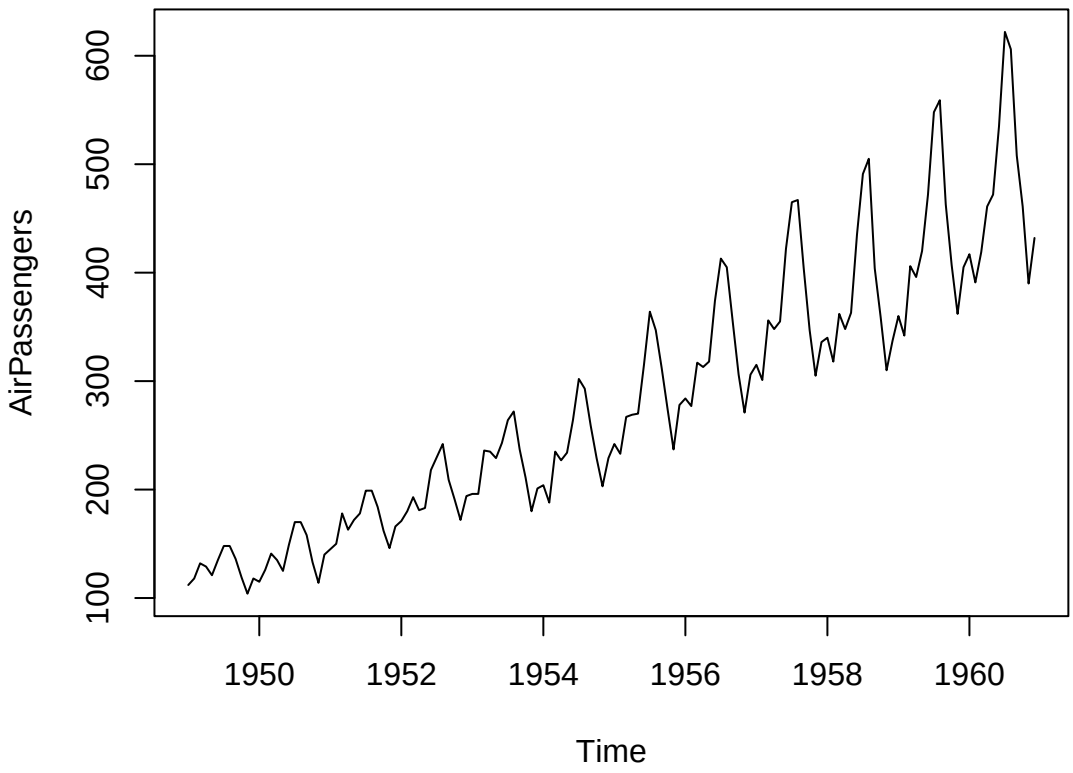
```
# Plot the Nile data with xlab, ylab, main, and type arguments
plot(Nile, type = "b", xlab = "Year", ylab = "River Volume (1e9 m^{3})",
     main = "Annual River Nile Volume at Aswan, 1871-1970")
```



2.3. Sampling frequency

The `start()` and `end()` functions return the time index of the first and last observations, respectively. The `time()` function calculates a vector of time indices, with one element for each time index on which the series was observed. For this example we will use the data set "AirPassengers", which reports the monthly total international airline passengers (in thousands) from 1949 to 1960.

```
# Plot AirPassengers
plot(AirPassengers)
```




```
# View the start and end dates of AirPassengers
start(AirPassengers)
```

```
## [1] 1949    1
```

```
end(AirPassengers)
```

```
## [1] 1960   12
```

```
# Use time(), deltat(), frequency(), and cycle() with AirPassengers
time(AirPassengers)
```

```
##           Jan      Feb      Mar      Apr      May      Jun      Jul      Aug
## 1949 1949.000 1949.083 1949.167 1949.250 1949.333 1949.417 1949.500 1949.583
## 1950 1950.000 1950.083 1950.167 1950.250 1950.333 1950.417 1950.500 1950.583
## 1951 1951.000 1951.083 1951.167 1951.250 1951.333 1951.417 1951.500 1951.583
```

```
## 1952 1952.000 1952.083 1952.167 1952.250 1952.333 1952.417 1952.500 1952.583
## 1953 1953.000 1953.083 1953.167 1953.250 1953.333 1953.417 1953.500 1953.583
## 1954 1954.000 1954.083 1954.167 1954.250 1954.333 1954.417 1954.500 1954.583
## 1955 1955.000 1955.083 1955.167 1955.250 1955.333 1955.417 1955.500 1955.583
## 1956 1956.000 1956.083 1956.167 1956.250 1956.333 1956.417 1956.500 1956.583
## 1957 1957.000 1957.083 1957.167 1957.250 1957.333 1957.417 1957.500 1957.583
## 1958 1958.000 1958.083 1958.167 1958.250 1958.333 1958.417 1958.500 1958.583
## 1959 1959.000 1959.083 1959.167 1959.250 1959.333 1959.417 1959.500 1959.583
## 1960 1960.000 1960.083 1960.167 1960.250 1960.333 1960.417 1960.500 1960.583
##           Sep      Oct      Nov      Dec
## 1949 1949.667 1949.750 1949.833 1949.917
## 1950 1950.667 1950.750 1950.833 1950.917
## 1951 1951.667 1951.750 1951.833 1951.917
## 1952 1952.667 1952.750 1952.833 1952.917
## 1953 1953.667 1953.750 1953.833 1953.917
## 1954 1954.667 1954.750 1954.833 1954.917
## 1955 1955.667 1955.750 1955.833 1955.917
## 1956 1956.667 1956.750 1956.833 1956.917
## 1957 1957.667 1957.750 1957.833 1957.917
## 1958 1958.667 1958.750 1958.833 1958.917
## 1959 1959.667 1959.750 1959.833 1959.917
## 1960 1960.667 1960.750 1960.833 1960.917
```

The `deltat()` function returns the fixed time interval between observations and the `frequency()` function returns the number of observations per unit time.

```
deltat(AirPassengers)
```

```
## [1] 0.08333333
```

```
frequency(AirPassengers)
```

```
## [1] 12
```

Finally, the `cycle()` function returns the position in the cycle of each observation.

```
cycle(AirPassengers)
```

```
##      Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
## 1949   1   2   3   4   5   6   7   8   9  10  11  12
## 1950   1   2   3   4   5   6   7   8   9  10  11  12
## 1951   1   2   3   4   5   6   7   8   9  10  11  12
## 1952   1   2   3   4   5   6   7   8   9  10  11  12
## 1953   1   2   3   4   5   6   7   8   9  10  11  12
```

```
## 1954   1   2   3   4   5   6   7   8   9  10  11  12
## 1955   1   2   3   4   5   6   7   8   9  10  11  12
## 1956   1   2   3   4   5   6   7   8   9  10  11  12
## 1957   1   2   3   4   5   6   7   8   9  10  11  12
## 1958   1   2   3   4   5   6   7   8   9  10  11  12
## 1959   1   2   3   4   5   6   7   8   9  10  11  12
## 1960   1   2   3   4   5   6   7   8   9  10  11  12
```

3. Basic time series objects

3.1. Displaying time series: packages

A time series can be defined as a sequence of observations registered at consecutive time instant (Lamigueiro, O.P., 2014), and when these time instants are evenly spaced, the distance between then is called sampling interval. So, the time series visualization try to reveal changes of one or more quantitative variables through time, and to display relationships between variables and their behavior across the time.

To do this we will use the following packages (libraries): zoo and xts.

3.1.1. The package zoo

The package "zoo" (Zeileis and Grothendieck, 2005) provides an S3 class with methods for indexed totally ordered observations. Objects of class zoo area created by the function `zoo` from a numeric vector, matrix, or factor that is ordered by some index vector.

The package has the generic functions as: `summary(), print(), str(), head(), tail()`. The `agregate()` function splits the zoo object into subset along a coarser index grid, uses the function `sum()` for each subset, and returns the aggregated zoo object. Besides, the zoo package allows us to deal with missing observations, for example:

1.- `na.omit()` removes incomplete observations 2.- `na.contiguous()` extracts the longest consecutive stretch of non-missing values 3.- `na.approx()` replaces missing values by linear interpolation 4.- `na.locf()` replace missing values by the most recent non-NA value prior to it

3.1.2. The package xts

The "xts" package (Ryan and Ulrich, 2013) extends the zoo class object definition to provide a general time series object. The index vector for the time series values must be: `Date, POSICct, chron, yearmon, yearqtr` or `timeData`. Considering this, the subset operator "[" is able to extract data using the time format notation `CCYY-MM-DD HH:MM:SS`. Besides, this package allows us use other functions as:

1.- `endpoints()` which identifies the endpoints with respect to time 2.- `to.period()` changes the periodicity to a coarser time index 3.- The functions `period()` and `apply()` evaluate a function over a set of non-overlapping time periods.

3.2. Visualization

For this example we will use the "aranjuez" data. It contains 8 years of daily data (n=2898 records) from Aranjuez meteorological station, 10 variables. However, first you have to set the directory where is your data.

3.2.1. Set the Working Directory

Before reading any data, you must set the R working directory to the location of the data. For this you have tu use the following command:

`setwd("...")`: will set the current working directory to a specific location

where ... ia the path where is saved your data. To be sure about this, you can use the function `getwd()` that will print out the current directory by console.

When specifying the pathname, R reads forward slashes, whereas Windows reads backward slashes. So, you must change the slashes \ by the slashes /.

For example, if your data are stored in the folder "mydata" in the desktop of your computer, the path in your R script should look like this:

```
setwd("C:/Users/Desktop/mydata")
```

3.2.2. Reading R Data Files

- RData Files: Using the function `load()`, for example:

```
load("name_of_my_data.rdata")
```

- RDS Files: Using the function `readRDS()`

```
dataRDS <- readRDS("name_of_my_data.rds")
```

- .txt files (tab-delimited): Using the functions: `read.table()`

Common Parameters:

- Header: TRUE when first row includes variable names. The default is FALSE.
- Sep: A string indicating what is separating the data. The default is " ".

```
data_txt <-read.table("survey.dat", header=TRUE, sep= "\t")
```

- .csv files (comma-delimited): Using the function `read.csv()`

Common Parameters:

- Header: `TRUE` when first row includes variable names. The default is `FALSE`.

```
data_csv <-read.csv("survey.csv", header=TRUE)
```

- Reading Excel Data Files (XLSX or XLS): Using the function `read_excel()`

Common Parameters:

- Sheet: The name of the sheet or its location number.

It may be easier to use Excel to save individual sheets as CSV files and then read the CSV files into R. However, reading the XLSX and XLS extensions is possible in R:

```
install.packages("readxl")
library(readxl)
data_excel <- read_excel("survey.xlsx", sheet = 1)

data_excel <- read_excel("survey.xlsx", sheet = "sheetname")
```

This creates an R `tibble` object (the newer version of an R dataframe). If you are more comfortable with R dataframes, please use:

```
df_excel <- as.data.frame(data_excel)
```

We will continue with our example using the "aranjuez.RData".

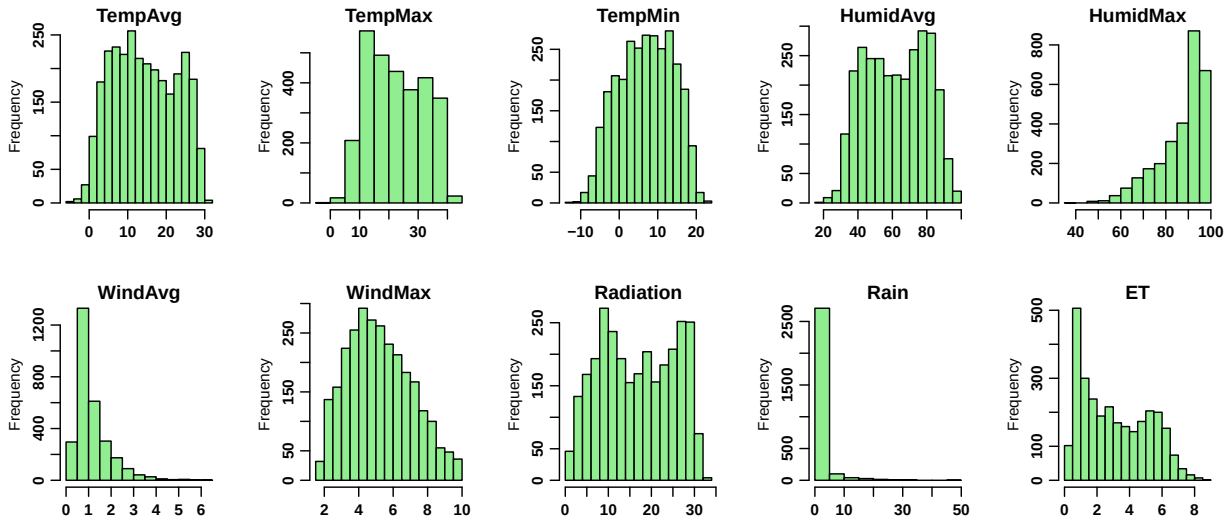
```
library(zoo)
load('aranjuez.RData')
dim(aranjuez)
```

```
## [1] 2898   10
```

```
head(aranjuez, 5)
```

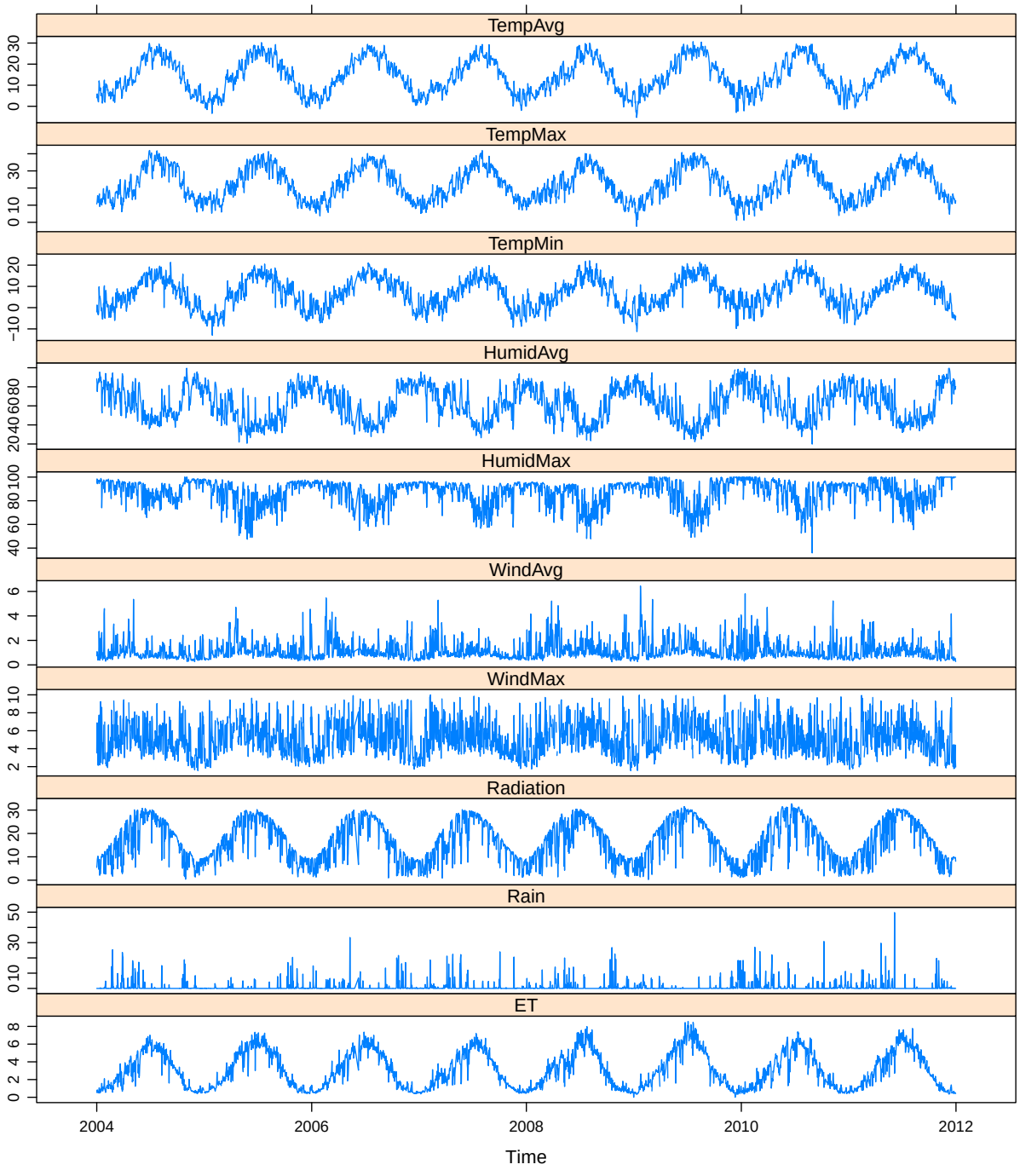
```
##            TempAvg TempMax TempMin HumidAvg HumidMax WindAvg WindMax Radiation
## 2004-01-01    4.04   10.71   -1.97      88.3      95.9    0.75    3.53      5.49
## 2004-01-02    5.78   11.52    1.25      83.3      98.5    1.08    6.88      6.54
## 2004-01-03    5.85   13.32    0.38      75.0      94.4    0.98    6.58      8.81
## 2004-01-04    4.41   15.59   -2.58      82.0      97.0    0.63    3.70      9.79
## 2004-01-05    3.08   14.58   -2.97      83.2      97.0    0.39    2.24     10.30
##            Rain   ET
## 2004-01-01    0 0.54
## 2004-01-02    0 0.77
## 2004-01-03    0 0.84
## 2004-01-04    0 0.69
## 2004-01-05    0 0.52
```

```
par(mar=c(4,4,1,1), mgp=c(2,0.7,0), font.axis=2, mfrow=c(2,5))
for(j in 1:10) hist(aranjuez[,j], main=colnames(aranjuez)[j], xlab="",
                    col="lightgreen")
```



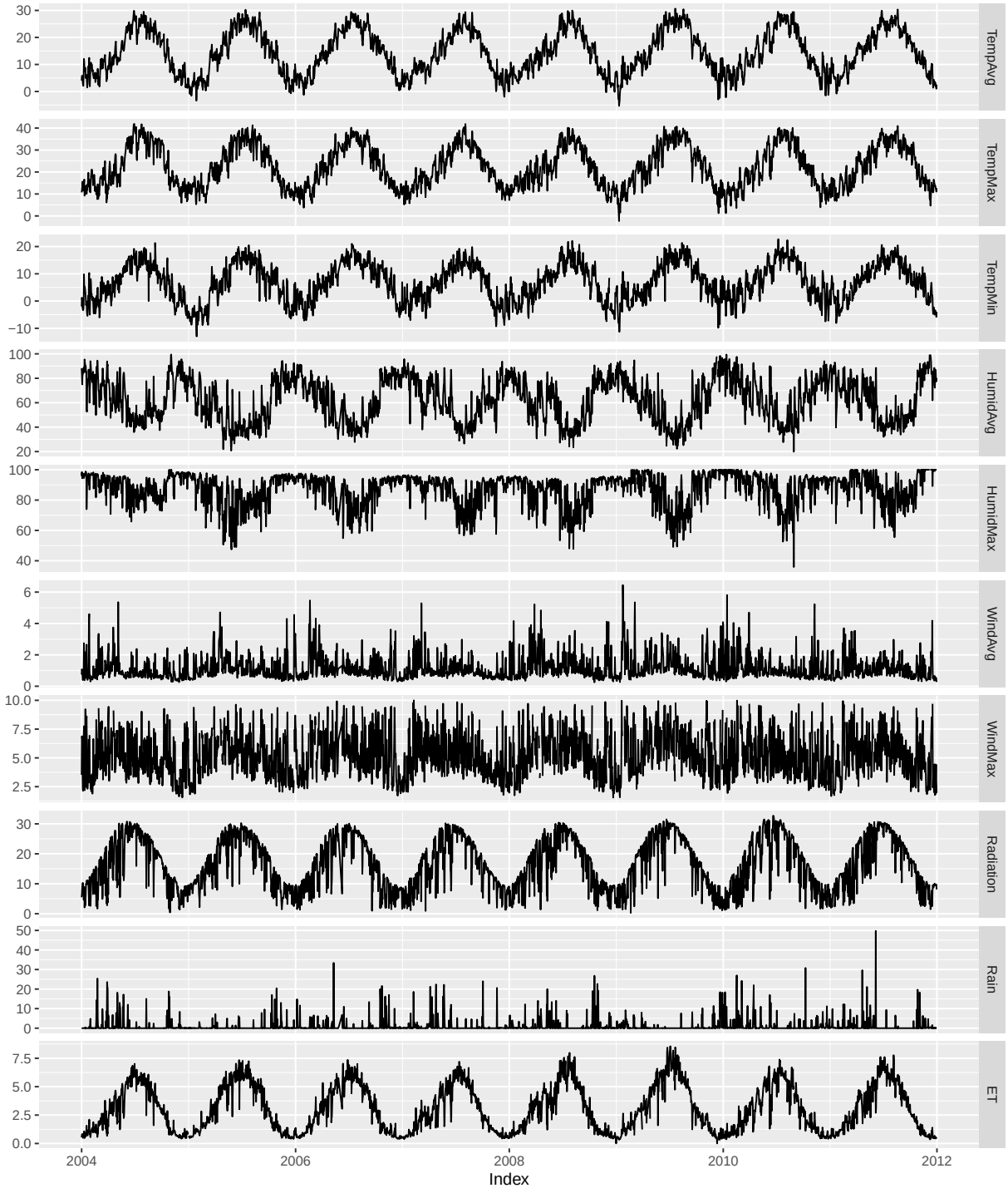
This multivariate time series data also can be displayed wiht the function `xyplot()` of the "lattice" package.

```
library(lattice)
xyplot(aranjuez, layout = c(1, ncol(aranjuez)))
```



or using the package "ggplot2" by the function `autoplot()`

```
library(ggplot2)
autoplot(aranjuez) + facet_free()
```



But, what is the characteristic of the "aranjuez" data:

```
str(aranjuez)
```

```
## 'zoo' series from 2004-01-01 to 2011-12-31
##   Data: num [1:2898, 1:10] 4.04 5.78 5.85 4.41 3.08 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:2898] "2004-01-01" "2004-01-02" "2004-01-03" "2004-01-04" ...
##   ..$ : chr [1:10] "TempAvg" "TempMax" "TempMin" "HumidAvg" ...
##   Index:  Date[1:2898], format: "2004-01-01" "2004-01-02" "2004-01-03" "2004-01-04" "2004-0
```

```
aranjuez[1]
```

```
##            TempAvg TempMax TempMin HumidAvg HumidMax WindAvg WindMax Radiation
## 2004-01-01    4.04   10.71   -1.97     88.3     95.9    0.75    3.53      5.49
##            Rain   ET
## 2004-01-01    0 0.54
```

```
coredata(aranjuez)[1,]
```

```
##   TempAvg   TempMax   TempMin  HumidAvg  HumidMax   WindAvg   WindMax Radiation
##      4.04     10.71     -1.97     88.30     95.90      0.75      3.53      5.49
##      Rain        ET
##      0.00      0.54
```

```
aranjuez$TempAvg[1:5]
```

```
## 2004-01-01 2004-01-02 2004-01-03 2004-01-04 2004-01-05
##       4.04       5.78       5.85       4.41       3.08
```

References

- Lamigueiro, O. P. (2014). Displaying time series, spatial, and space-time data with R. CRC Press.