Introduction to Spatial Data Programming - R

Lesson 03
Time series

Michael Dorman
Geography and Environmental Development
dorman@post.bgu.ac.il

Last updated: 2018-12-20 17:01:33

Ben-Gurion University of the Negev

Contents

- Dates
- Graphical functions
- Function definition

Aims

- Working with data which represent time (date)
- Learn how to visualize our data with graphical functions
- Learn to define custom functions

Dates

- In R, there are several special classes for representing **times** and **time-series** (time+data)
  - **Times**
    - `Date`
    - `POSIXct`
    - `POSIXlt`
  - **Time series** (time+data)
    - `ts`
    - `zoo` (package `zoo`)
    - `xts` (package `xts`)
- The simplest data structure for representing times is `Date`, used to represent **dates** (without time of day)

Dates

- For example, we can get the **current** date as a `Date` object with `Sys.Date` -

```
x = Sys.Date()
x
## [1] "2018-12-20"
```

```
class(x)
## [1] "Date"
```

- We can convert `character` values in the **standard** date format `YYYY-MM-DD` to `Date`, using `as.Date` -

```
as.Date("2014-10-20")
## [1] "2014-10-20"
```

Dates

- When the `character` values is in a **non-standard** format, we need to specify the format definition with `format`, using the various component symbols
- Full list of format symbols in `?strptime`

Table 1: Common `Date` format components

Symbol	Meaning
`%d`	day (`"15"`)
`%m`	month, numeric (`"08"`)
`%b`	month, 3-letter (`"Aug"`)
`%B`	month, full (`"August"`)
`%y`	Year, 2-digit (`14`)
`%Y`	Year, 4-digit (`2014`)

Dates

- For example -

```
Sys.setlocale("LC_TIME", "C")
## [1] "C"
```

```
as.Date("07/Aug/12")
## Error in charToDate(x): character string is not in a st
as.Date("07/Aug/12", format = "%d/%b/%y")
## [1] "2012-08-07"
```

```
as.Date("2012-August-07")
## Error in charToDate(x): character string is not in a st
as.Date("2012-August-07", format = "%Y-%B-%d")
## [1] "2012-08-07"
```

Dates

- The opposite conversion, with `format`, lets us **extract** specific date components out of a `Date` object -

```
d = as.Date("1955-11-30")
d
## [1] "1955-11-30"
```

```
format(d, "%d")
## [1] "30"
```

```
format(d, "%B")
## [1] "November"
```

```
as.numeric(format(d, "%Y"))
## [1] 1955
```

```
format(d, "%m/%Y")
## [1] "11/1955"
```

Dates

- `Date` objects act like **numeric** vectors with respect to certain **operations** that make sense on dates
- **Logical** operators -

```
Sys.Date() > as.Date("2013-01-01")
## [1] TRUE
```

- **Subtraction** -

```
as.Date("2013-01-01") - as.Date("2012-01-01")
## Time difference of 366 days
```

```
as.Date("2014-01-01") - as.Date("2013-01-01")
## Time difference of 365 days
```

Dates

- Creating **sequences** with the `seq` function -

```
seq(
  from = as.Date("2018-10-14"),
  to = as.Date("2019-01-11"),
  by = 7
)
##  [1] "2018-10-14" "2018-10-21" "2018-10-28"
##  [4] "2018-11-04" "2018-11-11" "2018-11-18"
##  [7] "2018-11-25" "2018-12-02" "2018-12-09"
## [10] "2018-12-16" "2018-12-23" "2018-12-30"
## [13] "2019-01-06"
```

Time series

- Let's define two `numeric` vectors: **water level** in Lake Kinneret, in **May** and in **November**, in each year during **1991-2011** -

```
may = c(
  -211.92,-208.80,-208.84,-209.12,-209.01,-209.60,
  -210.24,-210.46,-211.76,-211.92,-213.13,-213.18,
  -209.74,-208.92,-209.73,-210.68,-211.10,-212.18,
  -213.26,-212.65,-212.37
)
nov = c(
  -212.79,-209.52,-209.72,-210.94,-210.85,-211.40,
  -212.01,-212.25,-213.00,-213.71,-214.78,-214.34,
  -210.93,-210.69,-211.64,-212.03,-212.60,-214.23,
  -214.33,-213.89,-213.68
)
```

Time series

- Question 1: what is the average water level in May? in November?
- Question 2: was the water level ever below -213 (red line) in May? in November?
- Question 3: in which year or years was the water level below -213 in May? in November?

Time series

- Question 2: was the water level ever below -213 (red line) in May? in November?

```
any(may < -213)
## [1] TRUE
```

```
any(nov < -213)
## [1] TRUE
```

- Note: make sure there is a space between < and -, otherwise the combination is interpreted as an assignment operator <-

Time series

- Question 3: in which year or years was the water level below −213 in May? in November?

```
year = 1991:2011
year
##  [1] 1991 1992 1993 1994 1995 1996 1997 1998 1999
## [10] 2000 2001 2002 2003 2004 2005 2006 2007 2008
## [19] 2009 2010 2011
```

```
year[nov < -213]
## [1] 2000 2001 2002 2008 2009 2010 2011
```

```
year[may < -213]
## [1] 2001 2002 2009
```

- Note: we created a logical vector `nov < -213` and used it to subset the corresponding `year` vector

Tables

```
data.frame(year, may, nov)
```

```
##    year     may     nov
## 1  1991 -211.92 -212.79
## 2  1992 -208.80 -209.52
## 3  1993 -208.84 -209.72
## 4  1994 -209.12 -210.94
## 5  1995 -209.01 -210.85
## 6  1996 -209.60 -211.40
## 7  1997 -210.24 -212.01
## 8  1998 -210.46 -212.25
## 9  1999 -211.76 -213.00
## 10 2000 -211.92 -213.71
## 11 2001 -213.13 -214.78
## 12 2002 -213.18 -214.34
## 13 2003 -209.74 -210.93
## 14 2004 -208.92 -210.69
## 15 2005 -209.73 -211.64
## 16 2006 -210.68 -212.03
## 17 2007 -211.10 -212.60
## 18 2008 -212.18 -214.23
## 19 2009 -213.26 -214.33
## 20 2010 -212.65 -213.89
## 21 2011 -212.37 -213.68
```

Generic functions

- Some of the functions we learned about are **generic functions**
- Generic functions are functions that can accept arguments of different classes. What the function does depends on the class, according to the **method** defined for that class
- Advantages -
  - Easier to remember function names
  - Possible to run the same code on different types of objects
- For example, `mean`, `print` and `plot` (below) are examples of generic functions
- When the `print` function gets a **vector** it prints the values, but when it gets a **raster** `RasterLayer` object its prints a summary of its properties

Graphical functions

- The **graphical function** `plot`, given a vector, displays its values in a two dimensional plot where -
  - Vector **indices** are on the x-axis
  - Vector **values** are on the y-axis
- For example -

```
plot(nov, type = "b")
```

- Note: `type="b"` means **both** points and lines; Other useful options include -
  - `type="p"` for points
  - `type="l"` for lines
  - `type="o"` for overplotted lines and points

Graphical functions

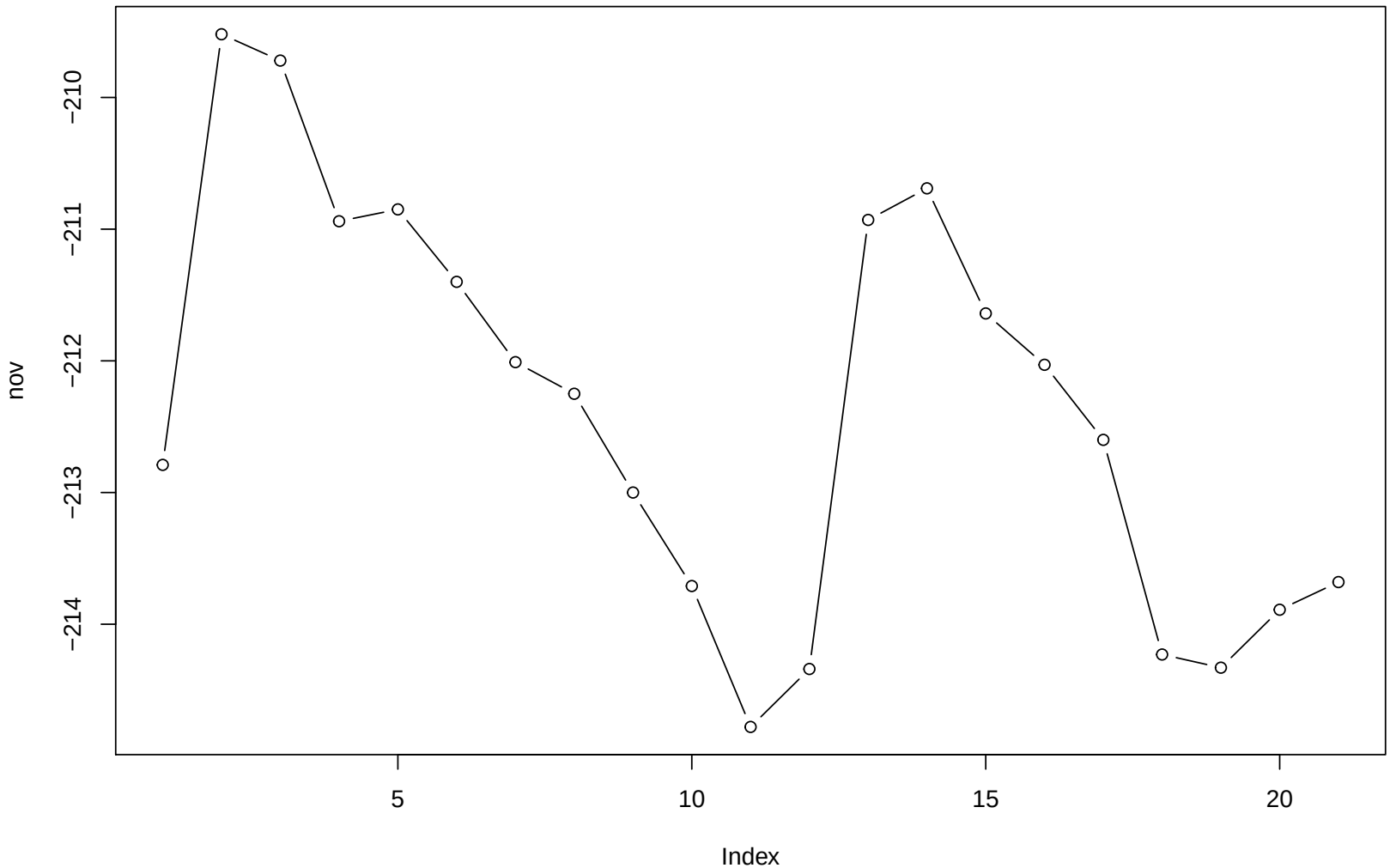

Figure 1: Plot of the `nov` vector

Graphical functions

- If we pass **two** vectors to `plot`, the first appears on the x-axis and the second - on the y-axis
- For example, we can put the years of water level measurement on the x-axis as follows -

```
plot(year, nov, type = "b")
```

Graphical functions

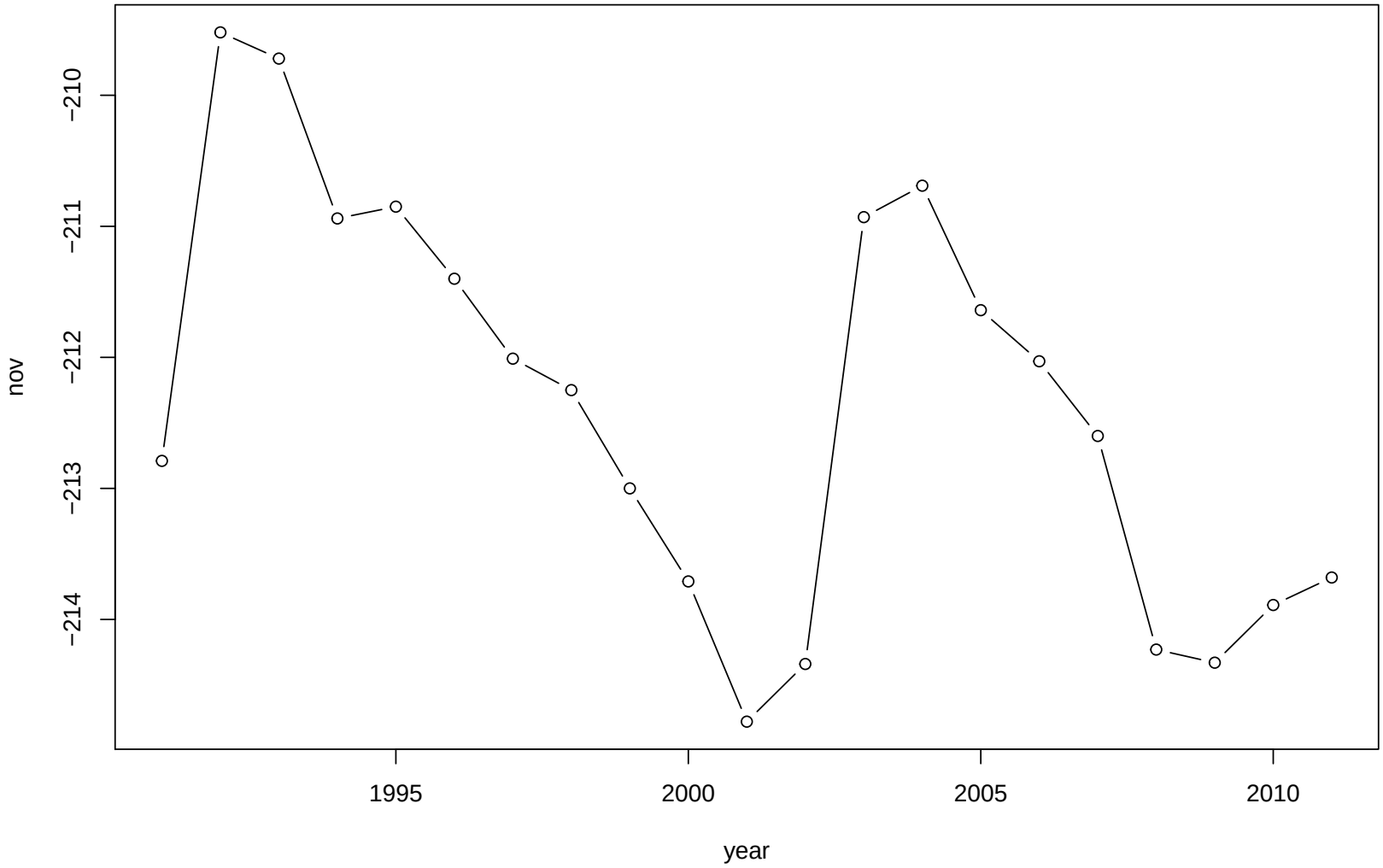

Figure 2: `nov` as function of `year`

Graphical functions

- We can add a **horizontal line** displaying the red line using `abline` with the `h` parameter -

```
plot(year, nov, type = "b")
abline(h = -213)
```

- Note that `abline` draws in an **existing** graphical device, which is initiated with `plot`

Graphical functions

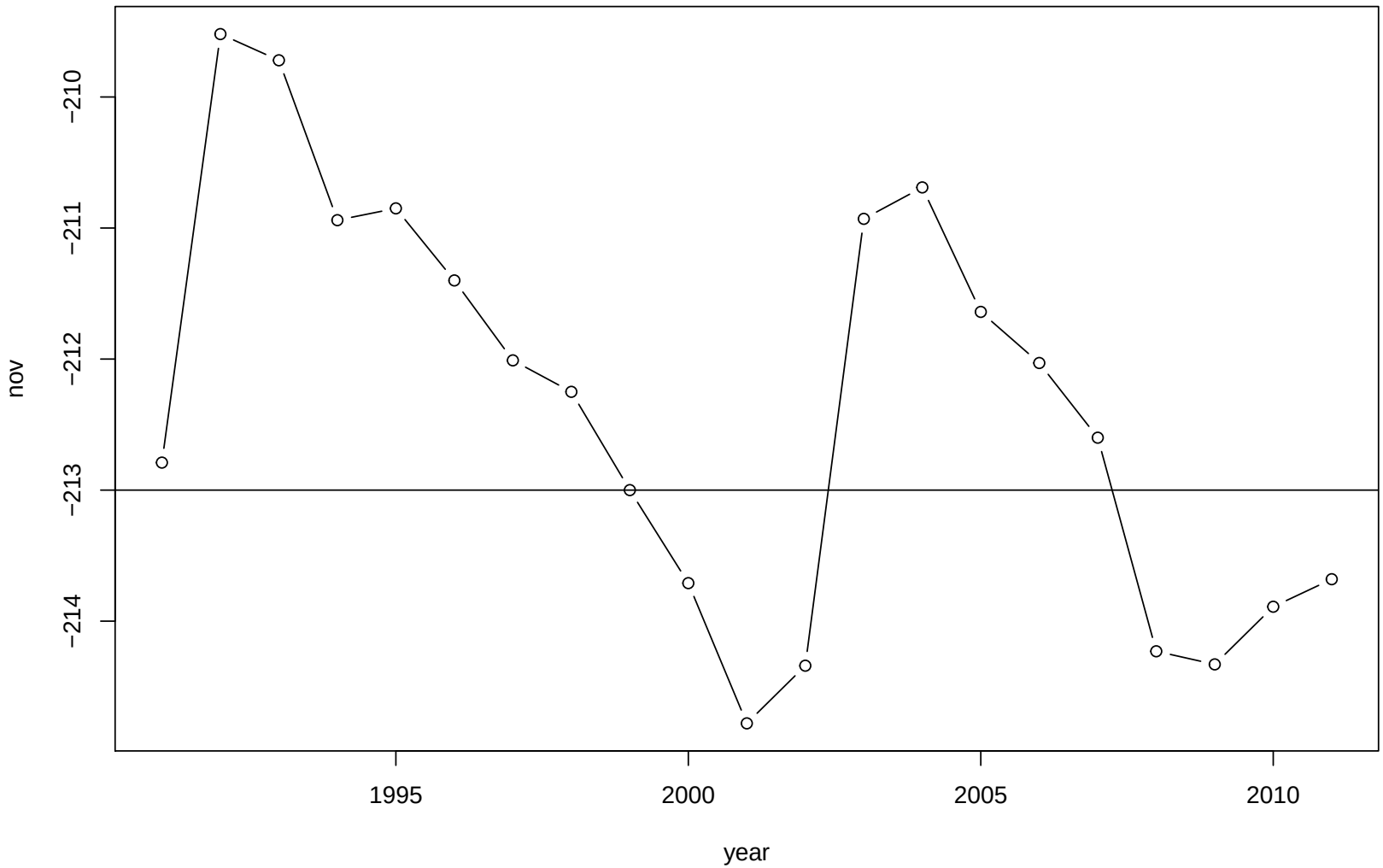

Figure 3: Adding a horizontal line with `abline`

Graphical functions

- **Additional** "layers" can be added to an existing plot using functions such as `points` and `lines`
- We can also use **graphical parameters** to specify different style for each layer, such as `col` to determine point / line color
- In addition, we will set the y-axis **limit** with `ylim` to make sure both time series are within the displayed range
- `ylim` accepts a vector of length two: the minimum and the maximum -

```
plot(
  year, nov,
  ylim = range(c(nov, may)),
  type = "b", col = "red"
)
lines(year, may, type = "b", col = "blue")
abline(h = -213)
```

Graphical functions

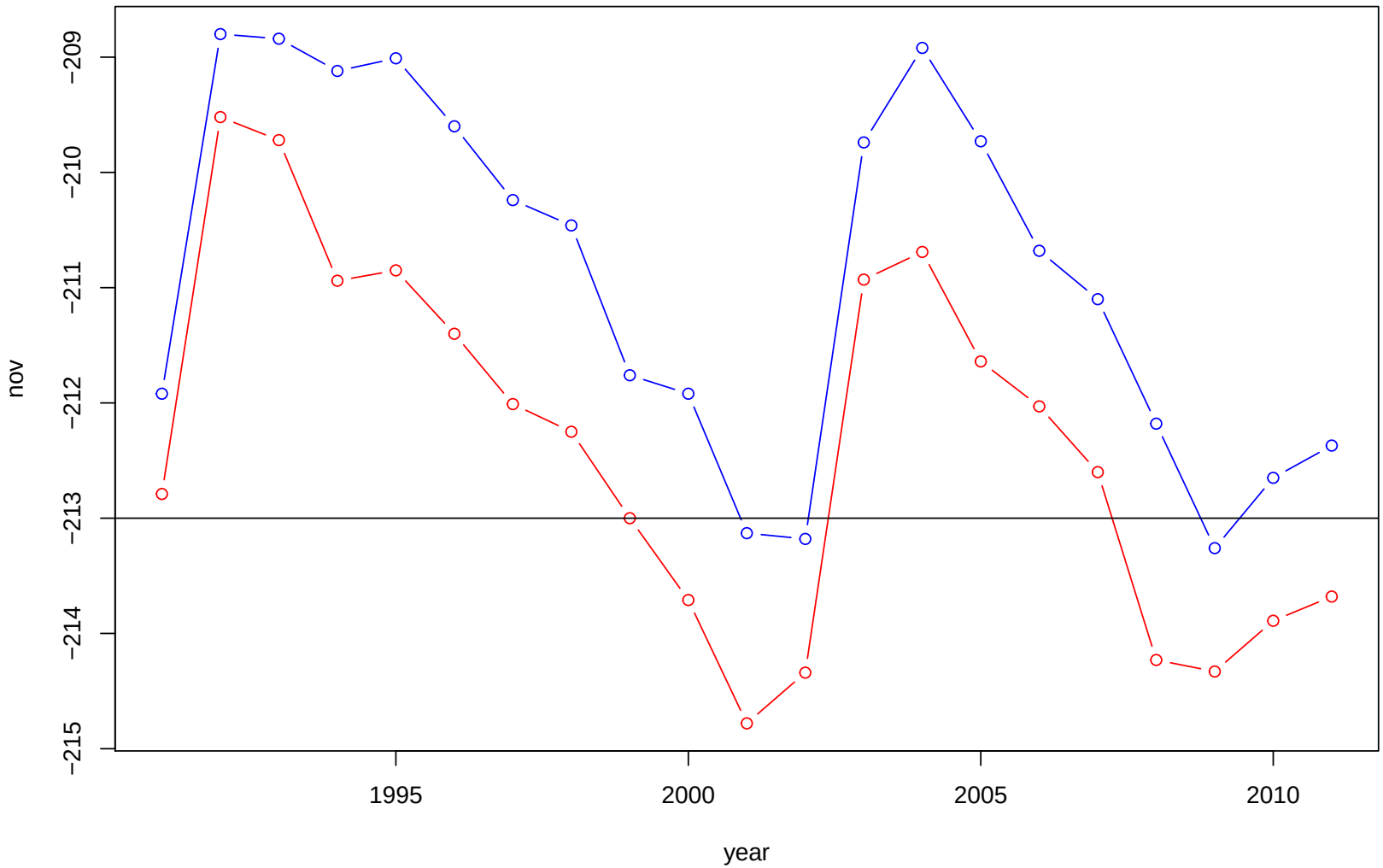

Figure 4: Adding a second series with `lines`

Graphical functions

- Finally, we can set the axis **labels** using the `xlab` and `ylab` parameters of the `plot` function -

```
plot(
  year, nov,
  xlab = "Year",
  ylab = "Water level (m)",
  ylim = range(c(nov, may)),
  type = "b", col = "red"
)
lines(year, may, type = "b", col = "blue")
abline(h = -213)
```

Graphical functions

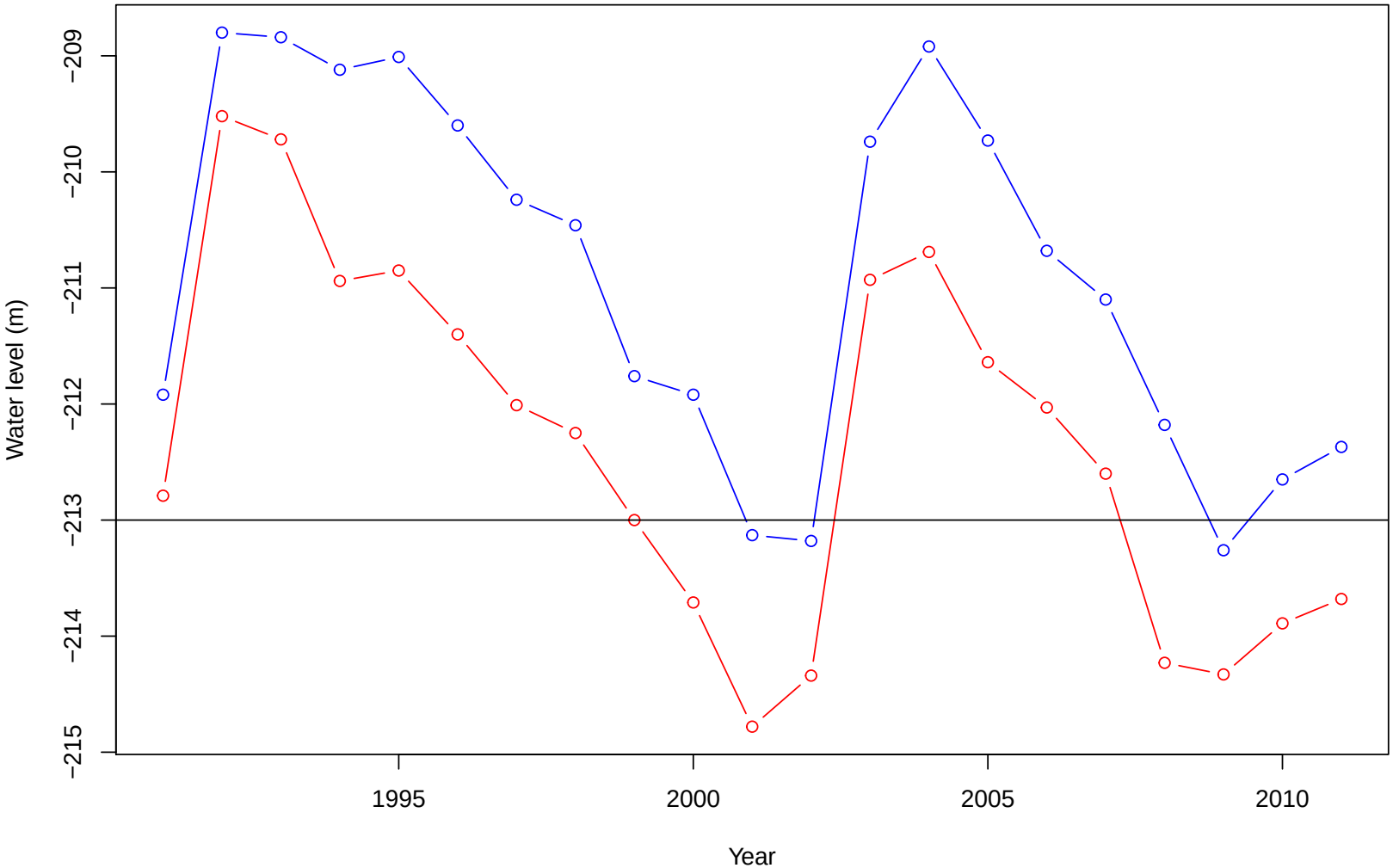

Figure 5: Setting axis labels

Consecutive differences

- The `diff` function can be used to create a vector of **differences** between consecutive elements -

```
d_nov = c(NA, diff(nov))
d_nov
##  [1]    NA  3.27 -0.20 -1.22  0.09 -0.55 -0.61
##  [8] -0.24 -0.75 -0.71 -1.07  0.44  3.41  0.24
## [15] -0.95 -0.39 -0.57 -1.63 -0.10  0.44  0.21
```

- Note: we add `NA` to match the input vector `nov`

Consecutive differences

- Now we can find out which year had the biggest water level **increase** or **decrease** -

```
year[which.max(d_nov)]
## [1] 2003
```

```
year[which.min(d_nov)]
## [1] 2008
```

- Note: `which.min` and `which.max` ignore `NA` values

Function definition

- In **Lesson 01** we learned that a function call is an instruction to execute a certain function, as in -

```
f(arg1, arg2, ...)
```

- A **function** is an object containing code, which is loaded into the RAM and can be executed with specific parameters
- So far we met function defined in the **default** R packages (e.g. `seq`)
- Later on we will use functions from **external** packages (e.g. `left_join`)
- Now we learn how to define our own **custom** functions

Function definition

```
add_five = function(x) {
  x_plus_five = x + 5
  return(x_plus_five)
}
```

- Function name (`add_five`)
- Assignment operator (`=`)
- Function keyword (`function`)
- Parameter(s) (`(x)`)
- Brackets (`{`)
- Code (`x_plus_five = x + 5`)
- Returned value (`return(x_plus_five)`)
- Brackets (`}`)

Function definition

- The idea is that the code inside the function gets **executed** each time the function is called -

```
# Function definition
add_five = function(x) {
  x_plus_five = x + 5
  return(x_plus_five)
}
```

```
# Function call, with argument 5
add_five(5)
## [1] 10
```

```
# Function call, with argument 7
add_five(7)
## [1] 12
```

Function definition

- When we make a function call, the values we pass as function arguments are assigned to **local variables** which the function code can use
- The local variables are **not accessible** in the global environment -

```
x_plus_five
## Error in eval(expr, envir, enclos): object 'x_plus_five
```

Function definition

- Every function **returns** a value
- We can **assign** the returned value to a variable to keep it in memory for later use -

```
result = add_five(3)
result
## [1] 8
```

Function definition

- We can **omit** the `return` expression -

```
return(x_plus_five)
```

- In which case, the returned value is the **last expression** in the function body
- We can also omit the `{` and `}` parentheses in case the code consists of a **single expression**
- Therefore we could define the `add_five` function using **shorter** code -

```
add_five = function(x) x + 5
```

Function definition

- **Default** parameter values can be given in the function definition
- In case there is a default value, we can **skip** that parameter in function calls -

```
add_five = function(x) x + 5
add_five()
## Error in add_five(): argument "x" is missing, with no d
```

```
add_five = function(x = 1) x + 5
add_five()
## [1] 6
```

Function definition

- There are **no restrictions** for the classes of objects a function can accept, as long as we did not set such restrictions ourselves
- However, we get an **error** if one of the expressions in the function code is illegal given the arguments -

```
add_five(1:3)
## [1] 6 7 8
```

```
add_five("one")
## Error in x + 5: non-numeric argument to binary operator
```

Function definition

- As another example, let's define a function named `first_last` which accepts a vector and returns the **difference** between the **last** and the **first** elements -

```
first_last = function(x) {
  x[length(x)] - x[1]
}
```

```
first_last(1:3)
## [1] 2
```

```
first_last(nov)
## [1] -0.89
```

Function definition

- Question: write a function named `modify` that accepts three arguments -
  - `x`
  - `index`
  - `value`
- The function assigns `value` into the element at `index` index of vector `x`
- The function returns the modified vector `x`
- For example -

```
v = 1:10
v
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
modify(x = v, index = 3, value = 99)
##  [1]  1  2 99  4  5  6  7  8  9 10
```