Session 1: Introduction to data analysis in R - a practical example

Christoph Schürz *christoph.schuerz@boku.ac.at*

The goal of the first session is that you get familiar with simple data analysis tasks in *R*. We learn basic tasks such as loading data, working with tables and visualizing analysis results, using Riesel climatic data.

Contents

Required packages

```
library(tidyverse)
library(lubridate)
library(pasta)
library(here)
```

Data acquisition

Loading climatic data directly from a webpage

I added this section just for the sake of completeness. It is a great example for all the challanges "real" data pose when working with them. Below I provide some ideas about working with "real and messy" data and how we can do better. I think this is important for working with data and it might spare you some hassle in the future. It is not essential to be able to follow every step of this section. You can download the resulting cleaned up dataset from the github repository. We will use that dataset to learn how to load data from your hard drive in the following section.

The Riesel climatic data is available from the ARS webpage. In general, there is not much of a difference between a path to a file on your comuter and a URL. Therefore, we can directly load the data from the internet.

```
# The URL where the weather files on the ARS webpage are stored
ars_url <- "https://www.ars.usda.gov/ARSUserFiles/30980000/riesel/climate"
# A verctor of the names of the weather files
weather_files <- 2010:2012%&%"hrly.txt"

# Loop over all weather file names, load the files from the web page and
# merge them to a tibble (data.frame)
ars_clim <- map_dfr(weather_files, ~read_table(file = url(ars_url%//%.x),
                                               guess_max =  10000,
                                               skip = 3, col_names = F))
```

The header of the tables provided online are (no offense) a bad example of storing data. Therefore I skipped the header line `skip = 3` when loading the data. In order to give the variables good names (one of the hardest tasks in programming) to work with in the following I created a name vector with "good" variable names. We will discuss the naming of variables and I will share some of my ideas to this topic at this point.

```
# The variable names we will assign to the variables in the weather data set.
tbl_header <- c("year",         # yyyy
                "day",          # jdn
                "hour",         # (h)hmm
                "t_air_ave",    # degC
                "t_air_max",    # degC
                "t_air_min",    # degC
                "rh_max",       # %
                "rh_min",       # %
                "p_vap_ave",    # kPa
                "sr_tot",       # kJ m^-2
                "wnd_v_ave",    # m s^-1
                "wnd_v_max",    # m s^-1
                "wnd_dir_ave", # deg
                "pr_tot",       # mm
                "t_sol_ave",    # degC
                "t_sol_max",    # degC
                "t_sol_min"     # degC
                )

names(ars_clim) <- tbl_header

ars_clim
```

```
## # A tibble: 29,679 x 17
##     year   day  hour t_air_ave t_air_max t_air_min rh_max rh_min p_vap_ave
##    <dbl> <dbl> <dbl>     <dbl>     <dbl>     <dbl>  <dbl>  <dbl>     <dbl>
##  1  2010     1   100      2.08       2.75      1.62   81.0   78.5     0.570
##  2  2010     1   200      1.28       1.68      0.88   83.8   80.8     0.55
##  3  2010     1   300      0.67       1.01      0.34   85.4   83.6     0.54
##  4  2010     1   400      0.08       0.48     -0.25   86.9   85.2     0.53
##  5  2010     1   500    -0.580     -0.18     -0.99   88.8   86.6     0.51
##  6  2010     1   600    -1.11      -0.78     -1.45   89.1   88.0     0.5
##  7  2010     1   700    -1.32      -1.18     -1.45   89.4   87.8     0.49
##  8  2010     1   800    -1.36      -1.24     -1.51   88.0   86.0     0.48
##  9  2010     1   900    -0.7        0.16     -1.31   86.2   80.5     0.49
## 10  2010     1  1000     1.03       1.9      -0.04   80.7   70.9     0.5
## # ... with 29,669 more rows, and 8 more variables: sr_tot <dbl>,
## #   wnd_v_ave <dbl>, wnd_v_max <dbl>, wnd_dir_ave <dbl>, pr_tot <dbl>,
## #   t_sol_ave <dbl>, t_sol_max <dbl>, t_sol_min <dbl>
```

Saving data

There are many ways and data formates to save your data. If I do not intend to use the data with any other software (e.g. Excel) or share the dataset with colleagues, my preferred way to store data is as ***.rds** files (short for R Data Single). The advantage of this format is, that it holds exactly one variable that I can assign to a specific variable name when loading it. This makes code easier to read.

If the data should be accessible by other software (and the data is a spreadsheet) I suggest to save in the **.csv** format. This is a format that can be interpreted by a wide range of software (in contrast to ***.xslx** and others).

Here I show you different ways to save your data. We will briefly talk about the pros and cons of all the methods. We will come back to saving data at a later point.

```
# Saving as a single data object
saveRDS(object = ars_clim, file = here("Session_1/ars_clim.rds"))

# Saving the data with the default save command
# you will also frequently see *.RData as a file suffix. .rda is just
# an abbreviation for that
save(ars_clim, file = here("Session_1/ars_clim.rda"))

# Writing the table in a csv file
## Using the readr package
write_csv(x = ars_clim, path = here("Session_1/ars_clim1.csv"))
## Using the base R funciton
write.csv(x = ars_clim, file = here("Session_1/ars_clim2.csv"))
```

First analysis and tidying of the data

Loading the data

The syntax for loading your data is equivalent to saving the data. The main difference is that instead of telling the function which variable you want to save/write, you assign the data you load to a variable. Here the difference between ***.rds*** *and* **.rda** becomes clear. When loading an RData file you cannot assign it to a variable, but the loaded data keeps the name it had when saving. The ***.rds*** *file must be assigned to a variable instead (same as the* **.csv**). To follow these lines of code download the files (or the entire *R* project) from the course github repository.

```
# Loading the rds file and assigning it to a new variable
ars_clim_rds <- readRDS(file = here("Session_1/ars_clim.rds"))

# Loading the content of the rda file into your working space
load(file = here("Session_1/ars_clim.rda"))

# Read a csv file into your working space
# Using the readr package
ars_clim_csv1 <- read_csv(file = here("Session_1/ars_clim1.csv"))
#Using the base R function
ars_clim_csv2 <- read.csv(file = here("Session_1/ars_clim2.csv"))
```

Exploring the data

I always suggest to do some checks on the data before using it. Below you find some ways to perform initial checks. A very nice *R* package to explore your data is the `DataExplorer`. I usually also check summary statistics of the data. These provide a good first insight for your data.

```
# Using the DataExplorer (I assume it is not installed on your computer)
# If you want to use it install it as follows (without the '#'):
# install.packages("DataExplorer")
# Exploring the data
# DataExplorer::create_report(ars_clim)

# Summary statistics
summary(ars_clim)
```

```
##       year                day              hour         t_air_ave
##  Min.   :   0.618   Min.   :  1.0   Min.   : 100   Min.   :-9.64
##  1st Qu.:2010.000   1st Qu.:102.0   1st Qu.: 700   1st Qu.:13.93
##  Median :2011.000   Median :190.0   Median :1300   Median :21.43
##  Mean   :2010.950   Mean   :188.4   Mean   :1250   Mean   :20.25
##  3rd Qu.:2012.000   3rd Qu.:278.0   3rd Qu.:1900   3rd Qu.:26.99
##  Max.   :2012.000   Max.   :366.0   Max.   :2400   Max.   :41.34
##  NA's   :4342       NA's   :4343    NA's   :4343   NA's   :4343
##    t_air_max       t_air_min         rh_max           rh_min
##  Min.   :-9.54   Min.   :-9.80   Min.   : 12.03   Min.   :  9.10
##  1st Qu.:14.70   1st Qu.:13.25   1st Qu.: 50.40   1st Qu.: 42.47
##  Median :22.16   Median :20.82   Median : 68.05   Median : 60.80
##  Mean   :21.06   Mean   :19.61   Mean   : 65.63   Mean   : 59.78
##  3rd Qu.:27.88   3rd Qu.:26.30   3rd Qu.: 83.31   3rd Qu.: 78.34
##  Max.   :47.80   Max.   :40.93   Max.   :100.60   Max.   :100.00
##  NA's   :4343    NA's   :4343    NA's   :4343     NA's   :4343
##    p_vap_ave          sr_tot           wnd_v_ave        wnd_v_max
##  Min.   : 0.160   Min.   :-99999.0   Min.   : 0.000   Min.   :  0.0
##  1st Qu.: 0.900   1st Qu.:     0.0   1st Qu.: 2.050   1st Qu.:131.4
##  Median : 1.650   Median :    41.0   Median : 3.250   Median :164.6
##  Mean   : 1.568   Mean   :   788.8   Mean   : 3.503   Mean   :170.9
##  3rd Qu.: 2.170   3rd Qu.:  1506.0   3rd Qu.: 4.730   3rd Qu.:196.9
##  Max.   :13.470   Max.   :  3948.0   Max.   :11.770   Max.   :360.0
##  NA's   :4343     NA's   :4344       NA's   :4344     NA's   :4344
##   wnd_dir_ave        pr_tot         t_sol_ave       t_sol_max
##  Min.   : 0.00   Min.   : 0.000   Min.   :-1.68   Min.   :-1.63
##  1st Qu.: 3.50   1st Qu.: 0.000   1st Qu.:14.36   1st Qu.:14.80
##  Median : 5.53   Median : 0.000   Median :22.22   Median :22.77
##  Mean   : 5.77   Mean   : 0.067   Mean   :22.10   Mean   :22.66
##  3rd Qu.: 7.63   3rd Qu.: 0.000   3rd Qu.:28.94   3rd Qu.:29.59
##  Max.   :21.20   Max.   :70.610   Max.   :55.41   Max.   :55.94
##  NA's   :4344    NA's   :4344     NA's   :4344    NA's   :4345
##    t_sol_min
```

```
##  Min.   :-1.71
##  1st Qu.:13.93
##  Median :21.70
##  Mean   :21.53
##  3rd Qu.:28.37
##  Max.   :54.59
##  NA's   :4345
```

You can see that all columns have almost the same number of `NA` values. The can indicate that there was an issue with reading the data. In this case I think the reason was that one of the tables from the web page had a lot of blank lines at the end. The `read_table()` function interpreted these as rows with no values. We will remove these lines from our data later.

Most of the variables seem to have plausible values according to the summary statistics. Only the variable `sr_tot` shows large negative values of `-99999.0`. Typically such values are used as `NoData` values. In the following step we will change these values to `NA`. The hours are provided in intervals of hundreds (probably to give the option to add minutes). To use the variable `hour` below, we have to devide it by `100`.

Tidying up the data, first mutations of the data

Working with dates

The date in our data is provided in three columns (year, julian day, and hour). In the following we want to create one date vector from these columns. Working with dates can cause a lot of trouble in *R* (e.g. with time zones, daylight saving time, leap years). Further, there are many ways to work with dates in *R*. I prefer to work with the `lubridate` package which is also from the `tidyverse`. As with the other `tidyverse` packages it has a very intuitive syntax. Unfortunately I cannot find a direct function to create dates from julian days. Therefore, I wrote short functions to do the trick. Whenever you stumble upon julian day format and you want to convert to date format you can use these functions:

```
# Function to convert from year and julian day to a date
yj <- function(y, jdn) {
   date <- ymd(y%//%"01"%//%"01")
   yday(date) <- jdn
   return(date)
 }

# Function to convert from year, julian day, and hour to a date
 yjh <- function(y, jdn, h) {
   date <- ymd_h(y%//%"01"%//%"01"%&&%h)
   yday(date) <- jdn
   return(date)
 }
```

Now we can use these functions to calculate dates from year, julian day and hour. To calculate new variables the `dplyr` package provides the function `mutate`.

```
ars_clim <- mutate(ars_clim, date = yjh(year, day, hour/100))
```

```
## Warning: 4343 failed to parse.
```

Cleanig the data, selecting variables

We identified some issues with the data in our data exploration. In the following we remove the rows that hold only `NA` values, change the `NoData` value for the solar radiation, and select the columns we are interested in.

Here you learn a bunch of new functions. First you see here a more 'modern' way of *R* programming using the pipe operator `%>%`. It litterally pipes the result from the previous function to the next function and puts it at the position of the `.` (e.g. `function(a, ., b)`). This might be a little unconventional to read in the beginning. It makes the code however very tidy and easy to read when used to it.

To filter specific rows from a table you can use the function `filter()`. It filters the rows where a logical expression is true. The selection of columns is done with the funcion `select()`.

```
ars_clim <- ars_clim %>% #This is the pipe operator
  filter(., !is.na(date)) %>%
  mutate(., sr_tot = ifelse(sr_tot < 0, NA, sr_tot)) %>%
  select(., date, t_air_ave, pr_tot, t_sol_ave)

ars_clim
```

```
## # A tibble: 25,336 x 4
##    date                t_air_ave pr_tot t_sol_ave
##    <dttm>                  <dbl>  <dbl>     <dbl>
##  1 2010-01-01 01:00:00     2.08       0      5.74
##  2 2010-01-01 02:00:00     1.28       0      5.08
##  3 2010-01-01 03:00:00     0.67       0      4.55
##  4 2010-01-01 04:00:00     0.08       0      4.2
##  5 2010-01-01 05:00:00    -0.580      0      3.98
##  6 2010-01-01 06:00:00    -1.11       0      3.73
##  7 2010-01-01 07:00:00    -1.32       0      3.51
##  8 2010-01-01 08:00:00    -1.36       0      3.28
##  9 2010-01-01 09:00:00    -0.7        0      3.1
## 10 2010-01-01 10:00:00     1.03       0      3.09
## # ... with 25,326 more rows
```

Visualizing daily mean, minimum, and maximum temperatures, and precipitation

Aggregation of the data to daily values

When aggregating temporal data I always prefer the work around to calculate the year, month, day, etc. values from the date, group the data by these values and aggregate the observations. I assume there are more elegant ways. This is simply my routine. The code example below demonstrates how to aggregate data by specific time steps. The grouping variables can however be any other variables as well.

```
# If the data is piped to the first position in a function you can omit the "."
ars_clim_day <- ars_clim %>%
  mutate(year = year(date),      # Using lubridates year to calculate the year
         mon  = month(date),     # You got the idea... :)
         day  = day(date)) %>%   # Same here
  group_by(year, mon, day) %>%
  summarize(pr_sum = sum(pr_tot),          # Calculate the daily precip sums
            t_air_mean = mean(t_air_ave), # Calculate the daily mean air temp
            t_air_max  = max(t_air_ave),  # Again I hope you got the idea...
            t_air_min  = min(t_air_ave),
            t_sol_mean = mean(t_sol_ave),
            t_sol_max  = max(t_sol_ave),
            t_sol_min  = min(t_sol_ave)) %>%
  ungroup(.) %>%  # After the aggregation I always ungroup to avoid unwanted
                  # behavior in following steps
  mutate(date = ymd(year%-%mon%-%day)) %>%  # Get the date from year, month, day
  select(- year, - mon, - day)
ars_clim_day
```

```
## # A tibble: 1,058 x 8
##    pr_sum t_air_mean t_air_max t_air_min t_sol_mean t_sol_max t_sol_min
##     <dbl>      <dbl>     <dbl>     <dbl>      <dbl>     <dbl>     <dbl>
##  1   0         4.28      10.6      -1.36      6.22      11.1       3.09
##  2   0         5.27      11.2       0.8       5.99      10.9       2.6
##  3   0         3.99       6.73      1.26      5.97       8.23      3.88
##  4   0         0.644      4.75     -2.73      4.93       7.87      2.59
##  5   0         0.653      6.48     -4.18      4.22       9.23      1.07
##  6   1.01      5.35       9.37     -0.05      5.56       8.03      3.16
##  7   0.25     -0.220      8.12     -4.77      4.48       7.23      1.51
##  8   0        -5         -2.13     -6.88      1.09       2.58      0.43
##  9   0        -3.56       3.01     -9.1       0.887      4.25     -0.72
## 10   0        -0.891      6.09     -6.89      1.79       5.78     -0.25
## # ... with 1,048 more rows, and 1 more variable: date <date>
```

Visualization of the data

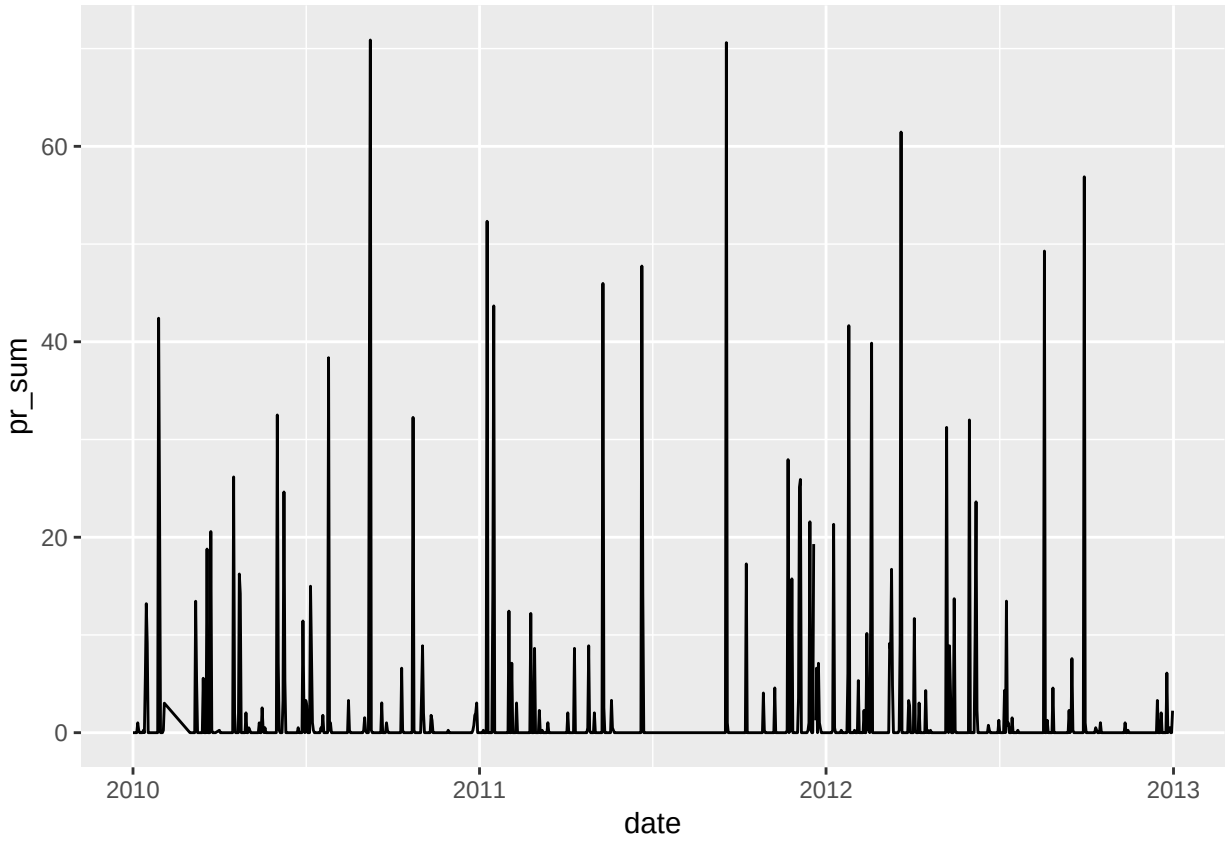
Again, there are many ways for visualization in *R*. My preferred way is to use the `ggplot2` package, because of its great flexibility and intuitive syntax. We will talk about the characteristics of `ggplot2` in this course. An excellent overview to use it in the future is provided by RStudio's data visualization cheatsheet (See all the other fantastic cheatsheets too!). It is essential to learn the basic concept of ggplot. The major benefit of ggplot is, when you got the concept you can apply it more or less to all of your visualization tasks. The code below applies ggplot to our climate data.

Prepare the precipitation data

```r
pr_data <- select(ars_clim_day, date, pr_sum)
```

Visual check of the precipitation data

```r
qplot(date, pr_sum, data = pr_data, geom = "line")
```



Prepare the temperature data

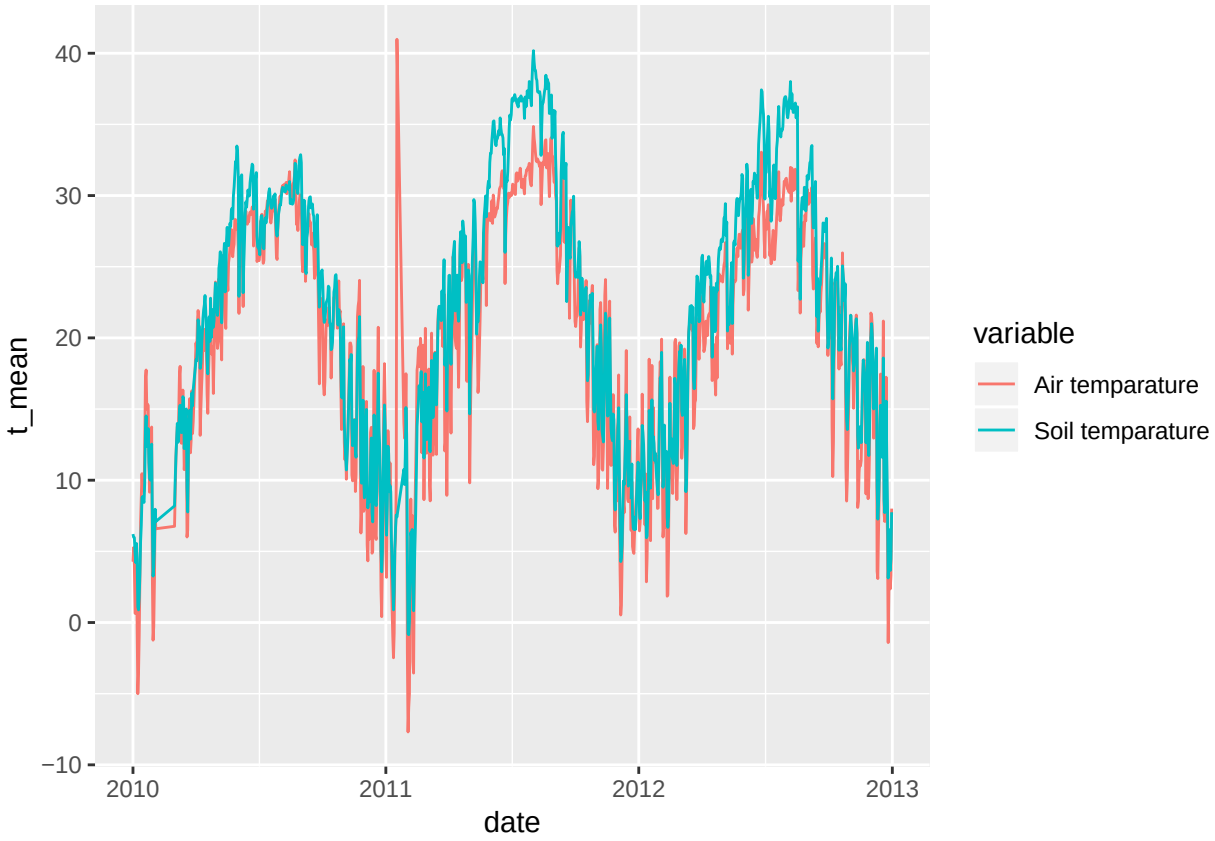
```r
# Prepare the air tempeartures
t_air <- ars_clim_day %>%
  select(date, starts_with("t_air")) %>%  # That is why good naming is important
  set_names(c("date", "t_mean", "t_max", "t_min")) %>%
  mutate(variable = "Air temparature")

# Prepare the soil tempeartures
t_sol <- ars_clim_day %>%
  select(date, starts_with("t_sol")) %>%
  set_names(c("date", "t_mean", "t_max", "t_min")) %>%
  mutate(variable = "Soil temparature")
```

```
# Combine the temperature data to a combined table
t_data <- bind_rows(t_air, t_sol)
```

Visual check of the temperature data

```
qplot(date, t_mean, data = t_data, col = variable, geom = "line")
```



Creating the final plot

Preparing the temperature plot

```
t_plot <- ggplot(data = t_data) +
  geom_ribbon(aes(x = date, ymin = t_min, ymax = t_max), alpha = 0.5) +
  geom_line(aes(x = date, y = t_mean)) +
  facet_wrap(variable ~ ., ncol = 1) +
  xlab("Date / yyyy") +
  ylab("Temperature / degC") +
  theme_bw()
```

Preparing the precipitation plot

```
pr_data <- mutate(pr_data, pr_sum = ifelse(pr_sum == 0, NA, pr_sum))

pr_plot <- ggplot(data = pr_data) +
  geom_bar(aes(x = date, y = pr_sum), col = "royalblue4", stat = "identity") +
  xlab("Date / yyyy") +
  ylab("Precipitation / mm") +
  theme_bw()
```

Combining both plots

A great way to arrange ggplots is provided by the `patchwork` package. You install it from the following github repository:

```
devtools::install_github("thomasp85/patchwork")
```

Combining plots with `patchwork` is very intuitive. You simply **add** them together.

```
library(patchwork)

pr_plot + t_plot + plot_layout(ncol = 1, heights = c(1, 2))
```