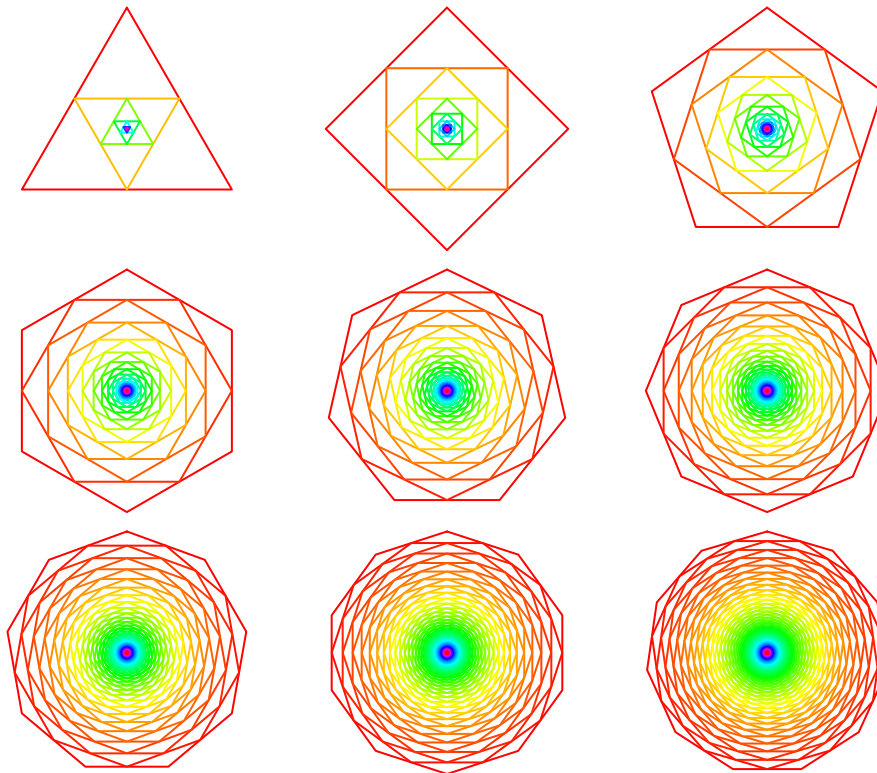


A Brief Introduction to Plotting in

by Roger Mundry

September 26, 2019



Contents

1	Before you begin	4
2	Introduction	4
2.1	Low and High Level Plotting Functions	4
3	The Basics	5
3.1	The Function <code>plot</code> (the Workhorse Function)	5
3.2	Graphical Parameters (selection)	6
3.2.1	Changing the Type of Plot (argument <code>type</code>)	6
3.2.2	Changing the Overall Appearance of a Plot Window	7
3.2.3	Graphical Parameters for Axes	8
3.2.4	Changing the Appearance of Points	9
3.2.5	Further graphical Parameters	10
4	Getting Several Plots into a Single Plotting Window	11
5	Adding Elements to a Plot	13
5.1	Adding Lines	13
5.1.1	Function <code>abline</code>	13
5.1.2	Function <code>lines</code>	15
5.1.3	Function <code>segments</code>	16
5.1.4	Function <code>arrows</code>	17
5.1.5	Defining the Appearance of Lines	18
5.1.6	Defining the Style of Line Ends (Argument <code>lend</code>)	18
5.2	Adding Points (Function <code>points</code>)	19
5.3	Adding Rectangles (Function <code>rect</code>)	20
5.4	Adding Polygons (Function <code>poly</code>)	21
5.5	Adding an Axis (Function <code>axis</code>)	22
5.6	Adding a Legend (Function <code>legend</code>)	24
5.7	Adding Text	25
5.7.1	The Function <code>text</code>	25
5.7.2	The Function <code>mtext</code>	27
5.7.3	Displaying long axis labels	28
5.8	Mathematical Symbols, Equations, Annotations, etc.	29
5.8.1	Greek Symbols	30
5.8.2	Mathematical Symbols and Expressions	30
5.9	Getting Entries out of R Objects into Plots	31
6	Colors	34
6.1	Colors by Name or Number	34
6.2	The Function <code>rainbow</code>	35
6.3	Colors in red-green-blue Space (Function <code>rgb</code>)	35
6.4	White-to-Black gradient (Function <code>grey</code>)	37
6.5	Custom Color Gradients (Function <code>colorRamp</code>)	38
6.6	Transparent Colors (Arguments <code>alpha.f</code> or <code>alpha</code>)	39
7	Customized Legends	41
8	Fonts	42
9	3D-plots (Function <code>persp</code>)	43

10 Complex Plots (getting "high-level" Plots using low-level Functions)	48
10.1 A Bubbleplot indicating Sample Size	48
10.2 A Boxplot for a Two-Factor Design	50
10.3 Combining Bubbleplots, Boxplots and Models	52
10.4 Two y-axes in one Plot	55
10.5 Data availability per Individual	57
10.6 Getting started with your own customized Plots	58
11 Writing your own plotting Functions	59
12 Defining the Size of Plotting Windows	62
13 Saving Plots	62
13.1 File Types	62
13.2 Saving Plots using <code>savePlot</code> or <code>dev.copy2pdf</code>	63
13.3 Saving Plots using the Function <code>tiff</code>	63
13.4 Saving a Plot using the CMYK Color Model	64
14 Plots for Papers and Presentations	64
15 and finally...	66
16 Acknowledgements	67
17 References	67

1 Before you begin

Please note that this is pretty much a first draft of a first draft. Practically this means a couple of things:

- please print this document only in case you are going to read it right away (and entirely) since most likely it will be updated changed/complemented very soon;
- please let me know about all typos, errors, etc. that you found (including grammar, punctuation, wording);
- please let me know in case you miss something; and
- please don't distribute it to others without my permission (I don't want this draft-of-a-draft to float around in the world).

Thanks for your understanding.

2 Introduction

Before I began using R (R Core Team 2017) I usually created plots by thinking about how the standard plots available in normal spreadsheet software could be used to somehow represent the data and models to be shown (well, at that time I rarely showed models since I didn't know how to do it). By this I limited my thinking and creativity up from the very beginning. Since I use R this is very different: whenever I need a plot I begin with trying to imagine what the best representation (I could think of) of the data and the model would be - and then create the plot. Since R is simply amazing with regard to its graphical capabilities, whatever I envisioned, it will surely be possible. These graphical capabilities alone would make a good reason to learn how to use R. Another advantage of R is that once the script creating a certain plot is created and saved (potentially together with the workspace), one can easily, and at any point later in time, create the plot again (e.g., when data were needed to be corrected) or modify it just a little (e.g., when the manuscript should be submitted to another journal that has slightly different conventions) - a potentially very time saving option. Here I'd like to explain a bit about how I create plots in R.

The way I use R for plotting is admittedly peculiar in the sense that I create most plots basically from scratch rather than using so-called 'high level' plotting functions (see below). However, it gives me more freedom and the power to achieve pretty much everything and exactly what I want (more on that below).

My motivation for writing this piece is actually many-fold: first, I began to produce it to get a little familiar with LaTeX and knitr (Xie 2013, 2014, 2016). In fact, this was my first serious attempt to create a document in LaTeX (here used in combination with knitr, an R-package that allows to integrate LaTeX and R). Secondly, although I really create plots in R all the time there are things I need only very rarely, and whenever I need them I might struggle with them quite bit. This document includes some of the stuff I use rarely in the hope that next time I'll need it I don't need to dig for a solution again but can look it up here. I also give this tutorial to the participants of my courses to save some time when it comes to plotting. Finally, I hope it will be helpful for others.

2.1 Low and High Level Plotting Functions

R provides so called 'low level' and 'high level' plotting functions. High level plotting functions create different kinds of plots (e.g., bar charts, boxplots, etc.). They usually create quite okay looking plots in a fairly simple way. However, they also limit us in a somewhat similar way as do the standard plots in spreadsheet programs (i.e., it might be tricky or even not possible at all to modify the plots as desired). Low level plotting functions are much more 'basic'; that is, they don't create great looking plots automatically. At the same time, though, they allow the user to really create the plot desired and do not really impose any limitations.

When I began using R I used high level plotting functions to some extent, but I soon felt that these limited me too much. Since then I mainly create plots using the function `plot` and in case I need lines, boxes, arrows, or whatever added to it, I use other functions to do this. This gives me the freedom to really create the plots I want. In the following I shall elaborate on this way of creating plots a little further.

3 The Basics

3.1 The Function `plot` (the Workhorse Function)

The function `plot`¹ is my 'workhorse' function to create plots in R, and I base pretty much all kinds of figures on it (with the exception of 3D-plots). The reason for this is that it provides unsurpassed flexibility and ease of use. Basically, one needs to know some functions allowing to add graphical objects (like points, lines, rectangles, etc.) to an existing plot, and that's it. In fact, I always create plots using the same few function(s) and don't have to learn much. The alternative would be to use many different plotting functions which might have their special peculiarities and limitations (for instance, it's of course possible to add error bars to a bar chart created using the function `barplot`, but this is as complicated and specific as it is to just do everything from scratch using the functions `plot`, `rect`, and `segments`; but I hardly ever use bar charts anyways).

The function `plot` can be used to get scatter plots, histograms and barcharts (of discrete variables), line plots, mosaic plots, and boxplots (to mention just a few). The details of how to get them are explained further below. Most usually one will call this function with two vectors representing the x- and y-values to be depicted. If both are numeric or integer the resulting plot will be a scatter plot (Fig. 1a); if `x` is a factor and `y` numeric or integer, you'll get a boxplot (Fig. 1b); and if both are factors you'll get a 'mosaic plot' (Fig. 1c). A few examples might illustrate this (in the following code chunk worry only about how `x` and `y` are created and how the function `plot` is called, but not about the functions `par`, `set.seed`² and `mtext`; these will be explained later):

```
par(mfrow=c(1, 5), mar=c(3, 3, 1.2, 0.2), mgp=c(1.7, 0.3, 0), tcl=-0.15)
set.seed(1)
x=rnorm(n=100)
y=runif(n=100)
plot(x=x, y=y)
mtext(text="(a)", side=3, line=0.1, cex=0.8)
x=as.factor(sample(x=letters[1:4], size=100, replace=T))
plot(x=x, y=y)
mtext(text="(b)", side=3, line=0.1, cex=0.8)
y=as.factor(sample(x=letters[1:4], size=100, replace=T))
plot(x=x, y=y)
mtext(text="(c)", side=3, line=0.1, cex=0.8)
plot(x=x)
mtext(text="(d)", side=3, line=0.1, cex=0.8)
y=runif(n=100)
plot(x=y)
mtext(text="(e)", side=3, line=0.1, cex=0.8)
```

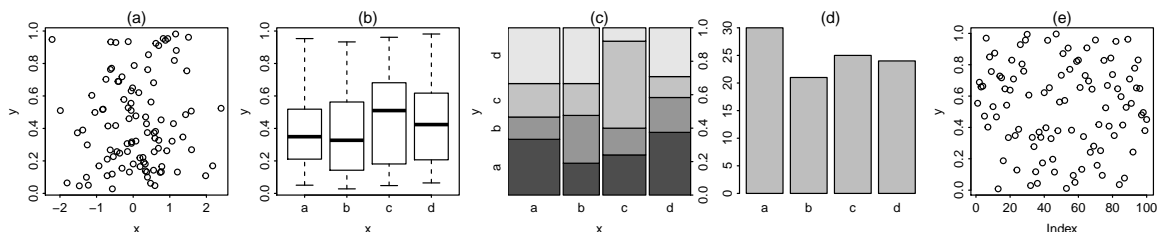


Figure 1: Different kinds of plots one can create using `plot`.

Note that you can also call the function `plot` with only one vector handed over to just the argument `x` (see

¹throughout the text I use text in *courier* to indicate the names of R-functions, arguments of such functions and also objects available in the current working environment; words in carets `<...>` represent placeholders in pieces of R code which need to be replaced before the respective instruction could be run (e.g., `<vector>` might need to be replaced by `age`); also note that this tutorial requires a recent R version (3.x).

²The function `set.seed` called in the code deserves some explanation: the function `runif` draws pseudo random numbers from a uniform distribution. Setting a certain 'seed' has the consequence that the sequence of random numbers generated following such a call of `set.seed` is completely determined, which has the effect that if you try the same code, you'll get the same figure (which might avoid confusion).

the above code chunk). If this vector is a factor you'll get a bar plot showing the relative frequencies with which the different levels do occur (Fig. 1d); and if it is numeric or integer, you'll get just its values plotted against their index (position in the vector; Fig. 1e).

3.2 Graphical Parameters (selection)

Most plotting functions will produce a reasonably looking plot with just a few arguments handed over. However, they might not look very much as desired. Apart from using different plotting functions there are largely two ways of changing the appearance of individual plots: adding elements to a plot (see section 5) and changing graphical parameters.

Graphical parameters allow to manipulate the appearance of plots. They allow to modify such things as the color of the fore- and background, the width of lines, the appearance of the box around the plot, the widths of the margins of the plotting region, the size of text and symbols, or the orientation of axis labels, to mention just a few. Some of them can only be handed over to the function `par`, but others can be handed over to many plotting functions. For instance, the argument `lwd` which sets the width of lines can be handed over to all plotting functions that add elements comprising lines to a plot, for instance, the functions `plot`, `points`, `rect`, `polygon`, `lines`, `segments`, or `arrows`.

In this section I'll give an overview about some the graphical parameters I use frequently (others will be explained in the sections where they are most relevant; in a way, probably half this tutorial is about graphical parameters). You can learn about all of them by looking at the help of the function `par` (use `?par` to get to see this³). However, I'll not discuss all of them (quite a few I don't even know and have never used until now) but focus on those I use quite commonly. Throughout, I shall use the function `plot` to show how to use them, but most of them will work with other graphical functions as well.

3.2.1 Changing the Type of Plot (argument type)

Some examples for different types of plots one can get using the function `plot` you have seen already in Fig. 1. But also when both arguments `x` and `y` get handed over vectors being numeric or integer one can get quite different looking plots. The default plot for such data is a scatter plot, but the argument `type` of the function `plot` allows to get quite different plots, too.

First, it is possible to get a line graph connecting the points to be depicted in the order in which they appear in the two vectors handed over. Try

```
set.seed(1)
days.using.R=1:10
nr.plots=rpois(n=10, lambda=days.using.R/2)
par(mar=c(3.2, 3.2, 1.2, 0.5), mgp=c(2, 0.8, 0))
plot(x=days.using.R, y=nr.plots, las=1, xlab="days since using R", ylab="number plots created",
     type="l")
```

to see that (Fig. 2a). As is obvious, such a plot usually only makes sense when the data are ordered according to the values to be plotted at the x-axis and when they represent some kind of time series or something similar (in the example of the plot this could, for instance, be the number of days since starting using R and the number of successfully produced graphs). By the way, the argument `type` is an argument of the function `plot` and its default is `"p"` which means to depict *points*. If you want to show the points and a line connecting them use `type="b"` (to get 'both' points and lines), e.g., `plot(x=days.using.R, y=nr.plots, las=1, xlab="days since using R", ylab="number plots created", type="b")` (Fig. 2b). Another option is a histogram which can be got using `plot(x=days.using.R, y=nr.plots, las=1, xlab="days since using R", ylab="number plots created", type="h")` (Fig. 2c). By the way if you have a table showing the frequency of something this can also be plotted and will reveal a histogram. Use `plot(table(nr.plots), las=1, xlab="number plots", ylab="number days")` to see that (Fig. 2d). I frequently use this option when

³probably you'll be quite overwhelmed when looking at this for the first time (as I was); but with time you'll familiarize yourself with it and find it more and more useful

I need a histogram of a count variable.

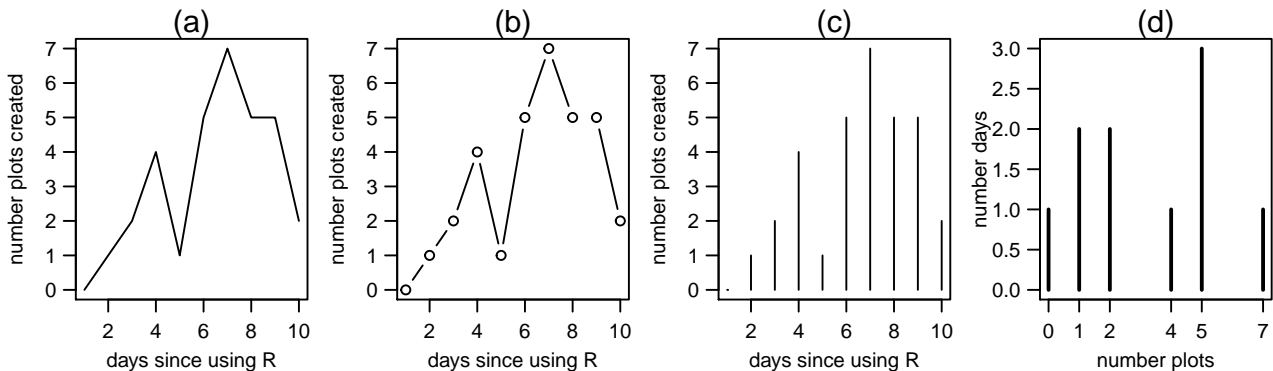


Figure 2: Examples of different types of plots created using the argument `type` of the function `plot`. (a), (b), and (c) are different visualizations of the same process (e.g., number figures successfully produced with increasing number of days since being an R user); (d) shows a histogram of the frequencies of a variable (here number plots per day).

Perhaps the most useful `type` (at least the one I use most frequently) is `"n"`. This means that a plotting window is created but no data are shown. Try `plot(x=1, y=1, type="n")` or to see what happens. Further down I'll show how to also suppress axes, axis-labels, and tick labels. This option is extremely useful when the plot is more complicated and data, summaries of the data and/or models should be added 'manually'. Further down I will show some examples.

3.2.2 Changing the Overall Appearance of a Plot Window

Several graphical parameters allow for defining the overall appearance of a plotting window. For instance, I'm frequently unhappy with the wide margins around the plotting region. These can be changed by calling the function `par` with the argument `mar` ('margins'). The defaults can be found by calling the function `par` and naming the part of the output desired, e.g., `par()$mar` which reveals 5.1 4.1 4.1 2.1. These are the widths of the margins around the plot, counted beginning with the margin of the x-axis and in clockwise direction. The numbers indicate the widths of the margins in lines (of text); that is, in a margin of width 4, four lines of text could be displayed below/besides one another (see function `mtext` below for more). I frequently change the margins such that they are narrower at the top and right side of the figure. This is done by calling the function `mar` and handing over the new margins, e.g., `par(mar=c(4.0, 4.0, 0.5, 0.5))`. If we now create the plot again this will appear larger in the plotting window (because of the narrower margins around the plot). Use `plot(x=rnorm(10), y=rnorm(10))` to see that. Note that the parameter `mar` can only be changed by calling the function `par` (i.e., it doesn't have any effect when being handed over to the function `plot`).

Another useful graphical parameter is `mgp`. It allows to set the distance from the axes at which axis (argument 1) and tick labels (argument 2) are displayed (and also the distance of the axes from the box around the figure). To move axis and tick labels closer to the axes use, e.g., `par(mgp=c(2.5, 0.5, 0))`. Usually after having changed the parameter `mgp` to this new settings one will also need to create the plot with shorter ticks which can be achieved by calling `plot` or `par` with the argument `tcl` set to, e.g., `-0.25` (`-0.5` is the default).

The function `par` can be used to define many further aspects of the appearance of a plot. Here I want to show a few more that allow to change the color of the background, the figure, the axes, etc. itself. For instance, to change the color of the background one would use the argument `bg` ('background'). If this is changed one will probably also need to change the color with which the data are shown and the axes and their labels will be displayed. This can be done using the arguments `fg` ('foreground'), `col.axis` (color of the axes), and `col.lab` (color of the labels printed at the axes). An example of the use all the arguments mentioned in this section could look as shown in Fig. 3:

```
set.seed(1)
par(bg="blue", fg="yellow", col.axis="yellow", col.lab="yellow",
```

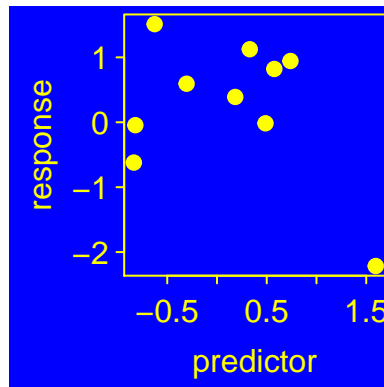


Figure 3: Illustration of the effects of the arguments `bg`, `fg`, `col.axis`, `col.lab`, `mar` and `mgp`.

```
mar=c(3, 3, 0.2, 0.2), mgp=c(1.7, 0.4, 0), tcl=-0.2)
plot(x=rnorm(10), y=rnorm(10), xlab="predictor", ylab="response", las=1, pch=19)
```

Note that the arguments `bg`, `fg`, `col.axis`, and `col.lab` can only be set by calling the function `par` but do not work when handed over to the function `plot`.

3.2.3 Graphical Parameters for Axes

In the previous section you have already seen the use of arguments `col.lab` and `col.axis`, allowing to change the color of the axes and axis labels as well as `tcl` allowing to define the tick length. Besides these there are several other graphical parameters for axes. Two of these are `xlab` and `ylab` (which I use every day). These allow to define the labels of the x- and y-axis, respectively. They get handed over texts indicated in quotes. If you don't want labels at neither the x- nor the y-axis, call the plot with `ann=F`; and if you don't want a label at only one of the axes give it an empty text (e.g., `xlab=""`).

Two further arguments I use every day are `xlim` and `ylim` which define the limits of the x- and y-axis, respectively. Both can be handed over when the function `plot` is called, and they both need to get handed over a vector with two numbers, defining the left and right limit of the x-axis and the lower and upper limit of the y-axis, respectively. Note that both, `xlim` and `ylim`, do not only define the range of the respective axis but also its values at either edge; that is, if you hand over a vector with two numbers to `xlim`, the first will define the left edge of the plot and the second the right (applies correspondingly to the y-axis). This can be useful when, for whatever reasons, one wants reverse the 'natural' order of values along axes (see Fig. 4 for an example).

```
par(mar=c(2.5, 2.5, 0.2, 0.2), mgp=c(1.7, 0.4, 0), tcl=-0.2, las=1, cex.axis=0.6, cex.lab=0.8)
x=0:25
y=dnorm(x=seq(from=-3, to=0, length.out=length(x)))
plot(x=x, y=y, ylim=rev(range(y)), type="l", xlab="distance from shore", ylab="water depth")
```

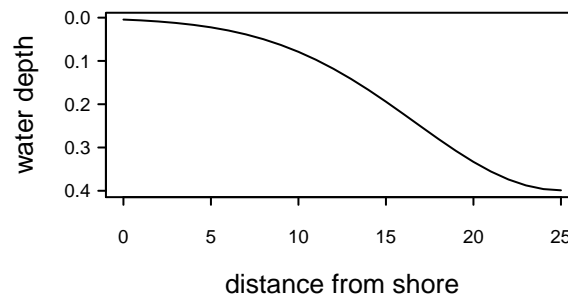


Figure 4: Example for a plot showing the y-axis with large values at the bottom (achived by use of `ylim` getting handed the larger number first)

Two further arguments dealing with the range of the axes are `xaxs` and `yaxs`. These allow to determine how much the plotting region extends beyond the data handed over to be plotted (or the limits set by `xlim` or `ylim`, respectively). The default ("`r`") adds some space on either side of the plot which aims at points not falling directly on the edge of the plotting region. The other possible value these arguments can take ("`i`") leads to the plotting region being limited to the range of the data handed over to `plot` (or what is specified with the arguments `xlim` or `ylim`). I use this latter option quite frequently, for instance, when plotting maps. Its also possible to entirely suppress the appearance of axis ticks and tick labels which can be achieved by setting the arguments `yaxt` and/or `xaxt` to "`n`". If you don't want to show neither of the two axes you could also use `axes=F`⁴.

The argument `tcl` ('tcl' abbreviates 'tick length') which defines the length of the ticks attached to axes I mentioned already. However, it might be worth mentioning how to get ticks being inside the plotting region. In fact, this can be very easily achieved giving `tcl` a small *positive* value (the default, -0.5, leads to ticks being displayed outside the plotting region which is okay for most journals, but some want the ticks to be displayed inside the plotting region).

Two further arguments might be worth mentioning, namely `cex.axis` and `cex.lab` which allow to adjust the size with which axis tick labels and axis labels, respectively, do appear ('cex' abbreviates 'character expansion'). The default for both is 1; so when you want labels to be larger indicate a larger value, and you want them to be smaller indicate a smaller value. Note the effects of these two arguments are quite 'drastic'; that is values differing only slightly from 1 (such as 1.2 or 0.8) already have quite an effect on the size of the labels (in fact, 2 doubles and 0.5 halves the size of them).

Another argument I use very frequently is `las`, ('label axis style') which defines the orientation of the tick labels. The values one can hand over to `las` are the integer numbers from 0 to 3, and `las=1` leads to tick labels displayed all upright (the default, 0, leads to the reading direction of the tick labels being parallel to the respective axis).

3.2.4 Changing the Appearance of Points

The default symbol when creating a scatter plot is the open circle. However, one might wish to use other symbols (for instance, filled symbols for presentations or different symbols for different levels of a factor). This can be achieved by use of the argument `pch` ('point character') which defines the points' appearance. This argument accepts integer numbers from 1 to 25 (well, it accepts also certain others, but these either don't lead to symbols shown or show characters). To get to see the available symbols (Fig. 5) use

```
set.seed(1)
par(mar=rep(0.1, 4))#set margin widths (see later for an explanation)
plot(x=1:25, y=rep(1, 25), pch=1:25, ann=F, axes=F, ylim=c(0.8, 1.2))
#(in the preceding line ann=F leads to no axis titles shown
#and axes=F suppresses depicting axes and the box around the figure)
text(x=1:25, y=1, labels=1:25, pos=3)#see below for more about the function text
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
○	△	+	×	◇	▽	⊠	*	⊕	⊗	⊞	⊟	⊠	⊡	■	●	▲	◆	●	●	○	□	◇	△	▽

Figure 5: Plotting symbols available with the argument `pch` getting handed over the integers 1 to 25.

Now lets assume we have data grouped by a factor and want to show the grouping of the data by depicting each level of it by another symbol. This can be achieved creating a vector indicating the point characters (i.e., the numbers associated with the different symbols) to be used (one for each level of the factor) and then indexing this vector by the factor turned into integer numbers from 1 to n with n being the number of levels the factor has. Assume the factor has two levels and we want to depict them using the open square (`pch=22`) for one

⁴note that when you create a plot with `axes=F` this will also have the effect that no box will appear around the figure; to subsequently add this use the function `box`

level and the filled diamond (`pch=18`) for the other. Fortunately, the cases of a factor can be easily turned into integer numbers using the function `as.numeric` with the factor handed over to it. This will turn the individual entries of a factor into integer numbers with the smallest integer being 1 and the largest being equal to the number of levels the factor has, whereby the individual cases are numbered according to the order with which the levels do appear when the function `levels` is applied to the factor. This is how it works in practice:

```
set.seed(1)
n=20
group=as.factor(sample(x=LETTERS[1:2], size=n, replace=T))#create grouping factor
var1=runif(n=n, min=0, max=10)
var2=runif(n=n, min=0, max=10)
pch.nr=c(22, 18)#assign symbols to the levels of the factor
par(mar=c(3, 3, 0.2, 0.2), mgp=c(1.7, 0.5, 0))#set margins etc. (see below)
plot(x=var1, y=var2, xlab="predictor", ylab="response", las=1, pch=pch.nr[as.numeric(group)], tcl=-0.25)
```

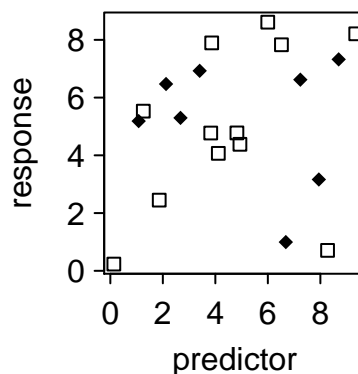


Figure 6: Simple scatter plot with different groups indicated by different symbols (open squares and filled diamonds).

In case you want to explore the above code producing Fig. 6 a little further, you probably should have a look at what `levels(group)` and `table(group, as.numeric(group))` reveal. Further down I'll show how to add a legend to such a plot.

Another argument I very frequently use when depicting points is `cex` ('character expansion') which allows to define the size by which the symbols are depicted. The default of it is 1, so if the points are too small choose a larger values, and when they are too large choose a smaller value (`cex=0` means that no point is shown). Below I'll show how the argument `cex` can be used to indicate sample sizes (see section 10.1).

There are plenty of other arguments one can use to manipulate the appearance of points. For instance, one can change the thickness of the lines surrounding points (see section 5.2) or their colors, and one can also use transparent colors (see section 6).

3.2.5 Further graphical Parameters

There are plenty of other graphical parameters, some of which are treated in this section. First, an argument which I use rarely but which nevertheless can be very useful is `main`. This is defining the *title* of the plot (I use it rarely because most journals I know do not accept plots with titles, but it can be very useful when many plots are created and one wants to easily see the name of the variable or data set depicted in the figure, though `mtext` does pretty much the same job; see below). The argument `main` gets handed over a text in quotes, e.g., `plot(x=rnorm(10), y=rnorm(10), main="some data")`, and it is handed over to the function `plot`.

Many journals have their particular 'house style' wrt to figures and, hence, when it comes to preparing figures for manuscripts to be submitted, particularly two arguments might play a role: First, the argument `bty` (which can be handed over to the function `plot`) allows to set the appearance of the *box* drawn around the plotting region. This argument takes one of "o", "l", "7", "c", "u", "]", or "n" where the first is the default, and the last leads to no box displayed. All but the last have names representing where around the plotting region

margins do appear (e.g., `bty="l"` creates margins to the left and below the plotting region). The other is `tcl` which allows to determine whether ticks are shown outside the plotting region or within it. This was already explained above (section 3.2.3).

Certainly, this overview of graphical parameters isn't complete, but just a starter. In fact, there are many more (some of which I rarely use or might have never used so far). For more about them see the help of the function `par` (which explains them in quite some detail and can be got by calling `?par`). Also, as already said above, probably at least half this tutorial is about graphical parameters, and throughout you'll encounter more of them and also those that I already introduced but with different values handed over.

4 Getting Several Plots into a Single Plotting Window

A variety of options does exist to get several plots into a single window. The most basic and simple is to use the arguments `mfrow` or `mfcoll` of the function `par`. These arguments allow to split the plotting region into columns and rows. Both get handed over a vector with two numbers indicating the number of rows and columns (in this order), respectively, into which the plotting window should be split. For instance, to get two plots besides one another, use `par(mfrow=c(1, 2))` which means to create a plotting window with one row and two columns. The resulting window now allows to comprise two plots which are produced from left to right. When using the function `mfrow`, the plots are created beginning in the top left corner and then added row by row whereby each time the function `plot` (or `hist` etc.) is called a new plot is created in the next of the available regions (when all are containing a plot already, the plotting will begin in the top left corner again). The difference between `mfrow` and `mfcoll` is that when using `mfrow` the plots are added row by row, whereas when `mfcoll` was used the plots are added column by column. An example would be Fig. 7, which I created using the following code (other examples are Fig. 1 and Fig. 2):

```
set.seed(1)
var1=rnorm(n=100)
var2=runif(n=100)
par(mfrow=c(1, 2), mar=c(3, 3, 1.5, 0.2), mgp=c(1.7, 0.3, 0), tcl=-0.15, las=1)
hist(var1, las=1, cex.main=0.7, cex.lab=0.7, cex.axis=0.5)
hist(var2, las=1, cex.main=0.7, cex.lab=0.7, cex.axis=0.5)
```

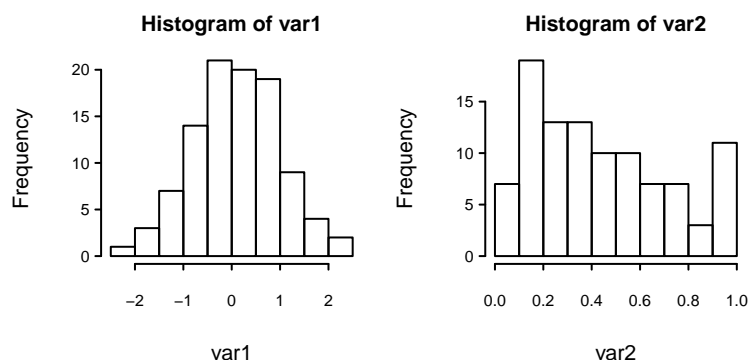


Figure 7: Two histograms displayed besides one another.

A more sophisticated (and more flexible) way of getting many plots into a single window is by using the function `layout`. This gets handed over a matrix with numbers defining how the plotting region should be split up and potentially two vectors defining the heights and widths of the different rows and columns in the plotting window. Assume, for instance, I'd want to plot the relation between two variables using scatter plots, separately for two species and both sexes. On top of the figures I'd like to depict the species name and to the left of it the sex; the axes labels should be shown centered across both figures. I'd begin with creating the matrix with numbers defining where each of the scatter plots should be depicted using `plot.mat=matrix(1:4, ncol=2, byrow=T)`. This creates a matrix with two rows and columns, comprising the numbers 1 to 4, and I shall use it to depict the respective scatter plots with the relation between the two variables, separately for the

four combinations of species and sex. Use `layout(plot.mat)` and then `layout.show(max(plot.mat))` to see what I have achieved so far. What is missing are two further plotting regions below and to the left of these plots into which I can put the axes' labels (centered across both figures in the respective direction). Furthermore, I need plotting regions above and to the left of the four plots to depict the names of the two species and the two sexes. This can be achieved by adding rows and columns to the matrix `plot.mat`. Let's begin with adding two rows, one on the top to finally show the names of the two species (which needs two plotting regions) and one at the bottom to show the common x-axis label (which needs one plotting region). I'd use `plot.mat=rbind(c(5, 6), plot.mat, 7)` to achieve this. Now `layout(plot.mat)` and `layout.show(max(plot.mat))` reveal what I have achieved so far. Notice that having a rectangular block of adjacent cells in the matrix which all contain the same number leads to all of them being assigned the same plotting region (here I have one plotting region (number 7) spanning the entire bottom of the plotting window horizontally to eventually depict the common x-axis label). However, now all the four rows are of the same height whereas one would probably like to show the data in higher plots as compared to annotations (of species) and the x-axis label. This can be achieved by using the argument `heights` of the function `layout`. This needs to be a vector with numbers indicating the relative heights of the (here) four rows (from top to bottom). Here I want the scatterplots to be depicted with five times the height of the annotations and the axis label, so I can use `layout(plot.mat, heights=c(1, 5, 5, 1))`. Use `layout.show(max(plot.mat))` to see how it looks like now. Now I have to add two columns to the left that should show the common y-axis label (right next to the scatterplots) and to the left of it the two sexes. This leaves two corners (the top left and the bottom left corner) into which nothing needs to be drawn, but that need to get assigned numbers as well. I usually give these the largest numbers in the matrix. Since so far the largest number in the matrix is 7 I'd add a column with two 8s (to depict the y-axis label) and a column with a 9 and a 10 (to depict the sexes) to the left of `plot.mat`; but to make the matrix `plot.mat` rectangular I need to add some additional (larger) numbers at the top and bottom of the vectors to be added, i.e., `to.add=cbind(c(11, 9, 10, 12), c(11, 8, 8, 12))`. Then I can use `plot.mat=cbind(to.add, plot.mat)`, `layout(plot.mat, heights=c(1, 5, 5, 1), widths=c(1, 1, 5, 5))`, and `layout.show(max(plot.mat))` to see what I produced. Here the additional argument `widths` defines the widths of the plotting regions (as shown above for the heights). The entire code looks like that (see Fig. 8 for the result):

```
#<<echo=TRUE, fig.height=2, fig.width=2, fig.align='center', background='white', comment=''>>=
plot.mat=matrix(1:4, ncol=2, byrow=T)#begin with four plots for the correlations
#add rows for species names and x-axis label, respectively:
plot.mat=rbind(c(5, 6), plot.mat, c(7, 7))
#create matrix to add for sex and y-axis label:
to.add=cbind(c(11, 9, 10, 12), c(11, 8, 8, 12))
plot.mat=cbind(to.add, plot.mat)#append it to plot.mat
plot.mat#have a look at plot.mat
```

	[,1]	[,2]	[,3]	[,4]
[1,]	11	11	5	6
[2,]	9	8	1	2
[3,]	10	8	3	4
[4,]	12	12	7	7

```
layout(plot.mat, heights=c(1, 5, 5, 1), widths=c(1, 1, 5, 5))#create plotting window
layout.show(max(plot.mat))#and show it
```

11	5	6
9	1	2
8		
10	3	4
12	7	

Figure 8: Result of the function `layout` used to set the plotting region such that several figures (regions 1 to 4) with common axis labels (regions 7 and 8) and identifiers of rows and columns (regions 5, 6, 8, and 9) can be plotted into a single plotting window.

Now I could 'fill' this plotting window with figures one by one (in the order indicated by the numbers depicted in Fig. 8). Below you will see an example of how to do that (section 10.3 and Fig. 51).

5 Adding Elements to a Plot

One of the cool things of R is that it is possible to 'manually' add elements to a an already existing plot. This has the great advantage that one isn't confined to, say, *either* create a boxplot *or* a scatter plot, but can flexibly combine all sorts of graphical elements, and there are really no limits to this. This section is about a whole bunch of functions allowing to add graphical elements such as rectangles, polygons, lines, arrows, points, axes, texts, etc. and how these can be used. Some of them will be shown in the following (there are others). Many of the functions shown in the following have some arguments in common. For instance, the functions `segments`, `lines`, `arrows`, `rect`, and `polygon` share the arguments `lty` defining the 'line type' (e.g., positive integers) and `lwd` defining the 'line width' (positive numbers; with both referring to the border of rectangles and polygons). In fact, this is one of the other cool features of R that it is so 'unified' or consistent; that is, all low level plotting functions for which it makes sense have, for instance, an argument `lwd` which is interpreted by all of them in exactly the same way.

5.1 Adding Lines

There are several ways of adding a line to a plot. Straight lines which extend over the entire plot can be added using the function `abline`; straight line segments connecting two points are got into a figure using the function `segments`; and lines connecting several points are conveniently drawn using the function `lines`.

5.1.1 Function `abline`

This function has several parameterizations. The first I show here is the one that can be used to add a regression line. For instance, in case I know the intercept and the slope of a regression I can call `abline`⁵ with the two arguments `a` (for the intercept) and `b` (for the slope). Assume, for instance, a data set comprising a number of known true values (thereafter 'truth') and an actual measure of the same thing ('measure'). Such a data set could be generated using

```
set.seed(1)
truth=runif(n=15, min=0, max=10)
measure=truth+rnorm(n=15, mean=0, sd=3)
```

Now the data can be plotted, and the line can be added. The line should show what the measured values would be if there were no measurement error (in which case the regression would have an intercept of 0 and a

⁵maybe you remember from school the equation $y = a + b \times x$; I guess this gave the function `abline` its name

slope of 1 and all points would lay exactly on this regression line). One can use `abline(a=0, b=1)` to get this line (Fig. 9a). Here's the code producing it:

```
par(mar=c(3, 3, 0.2, 0.2), mgp=c(1.5, 0.4, 0))
plot(x=truth, y=measure, pch=1, las=1, tcl=-0.25, xlim=range(c(truth, measure)), ylim=range(c(truth, measure)))
abline(a=0, b=1)
```

Note in the above code how the arguments `xlim` and `ylim` are used to give both axes the exact same scale. The function `abline` can also be used to add a regression line directly from the output of a simple linear regression. Let's add the relation between the measure and truth as estimated from the data using this parameterization (Fig. 9b):

```
par(mar=c(3, 3, 0.2, 0.2), mgp=c(1.5, 0.4, 0))
plot(x=truth, y=measure, pch=1, las=1, tcl=-0.25,
     xlim=range(c(truth, measure)), ylim=range(c(truth, measure)))
model=lm(measure~truth)#run regression
abline(model, lty=2)#add regression line to plot
```

Here I fit a linear model (here simple linear regression) and then hand over the result to the function `abline`. The additional argument `lty` defines the 'line type' and accepts, e.g., integer numbers (see below).

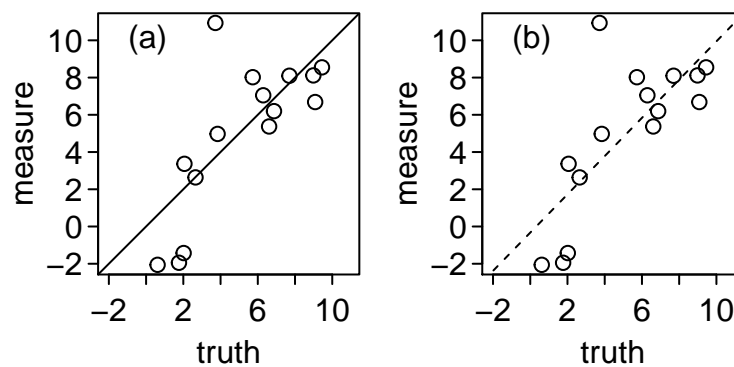


Figure 9: Illustration of the use of the function `abline` (and its argument `lty`). Depicted are two variables which should be identical (as indicated by the straight line in (a)) but are not. In (b) the actual relation between the two variables is shown.

Another parameterization of `abline` is one using the arguments `h` or `v` (to hand over the values at which horizontal and vertical lines should appear). For instance, one could get a grid into a plot using the following code (Fig. 10):

```
set.seed(seed=1)
var1=runif(n=100)
var2=runif(n=100)
par(mar=c(3.5, 3.5, 0.2, 0.2), mgp=c(2, 0.4, 0))
plot(x=var1, y=var2, tcl=-0.25, xlim=c(0,1), ylim=c(0,1), pch=19, col=grey(level=0.25, alpha=0.5))
abline(h=seq(from=0, to=1, by=0.2), v=seq(from=0, to=1, by=0.2), lty=3)
```

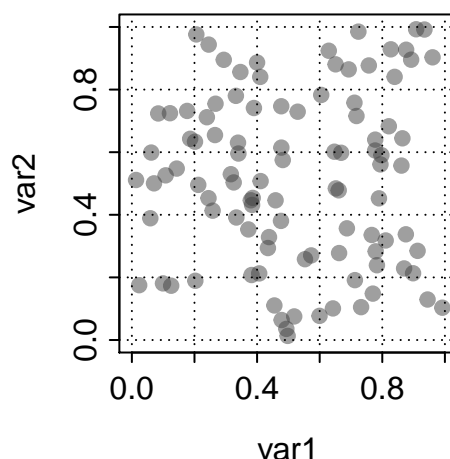


Figure 10: Example of a figure with grid lines added using the function `abline` using the arguments `h`, `v`, and `lty`.

Note that in Fig. 10 I use transparent points which is achieved by `col=grey(level=0.25, alpha=0.5)`. This has the effect that those parts of the points laying on top of others get darker (the argument `alpha` of the function `grey` allows to determine the level of opacity of the color; that is, the larger the value handed over to `level` the more opaque the point becomes; section 6.6 explains this in more detail).

5.1.2 Function lines

Another option to get lines into a plot is the function `lines`. This gets handed over a vector with x- and y-values and adds a line connecting the points defined by the x- and y-values in the order in which they appear in the two vectors. Let me use the results of a Poisson regression as an example. First, I need to generate some data and fit a model to them:

```
set.seed(1)
days.since.using.r=1:20
number.plots=rpois(n=length(days.since.using.r), lambda=0.2*days.since.using.r)
model.res=glm(number.plots~days.since.using.r, family=poisson)
```

Here the data are supposed to represent the number of plots created since one began using R. The model fitted is a Poisson regression since the response (`number.plots`) is a count. To plot the data one could just use

```
par(mar=c(4.5, 4.5, 0.5, 0.5))
plot(days.since.using.r, number.plots, pch=19, las=1, xlab="days since using R",
     ylab="number plots created")
```

Since a Poisson regression doesn't reveal a straight line I can't use the function `abline` to depict the model. Instead I create a vector with increasing x-values, calculate the fitted probabilities for them and then add the line using the function `lines`. To get the x-values from the smallest to the largest value of `days.since.using.r` I use the function `seq` ('sequence'), i.e., `x.values=seq(from=min(days.since.using.r), to=max(days.since.using.r), by=1)`. Here I create a vector with an increment of 1 because the predictor is discrete (otherwise I'd replace the argument `by` by `length.out` and indicate it to be, e.g., 100 to get a smooth line). In the next step I calculate the fitted values. Note that since the model is a Poisson regression I first need to derive the linear predictor and then turn it into fitted values by exponentiating them. Hence, I use `y.values=coefficients(model)["(Intercept)"+coefficients(model)["days.since.using.r"]*x.values` and then `y.values=exp(y.values)`. Now I can add the line using `lines(x.values, y.values, lty=2, lwd=2)` (Fig. 11). Here it is all the way through:


```

par(mar=c(3, 3, 0.2, 0.2), mgp=c(1.5, 0.4, 0))
plot(days.since.using.r, number.plots, pch=19, las=1, xlab="days since using R", tcl=-0.25,
      ylab="number plots created")
x.values=seq(from=min(days.since.using.r), to=max(days.since.using.r), by=1)
y.values=coefficients(model.res)["(Intercept)"+
  coefficients(model.res)["days.since.using.r"]*x.values
y.values=exp(y.values)
lines(x.values, y.values, lty=2, lwd=2)

```

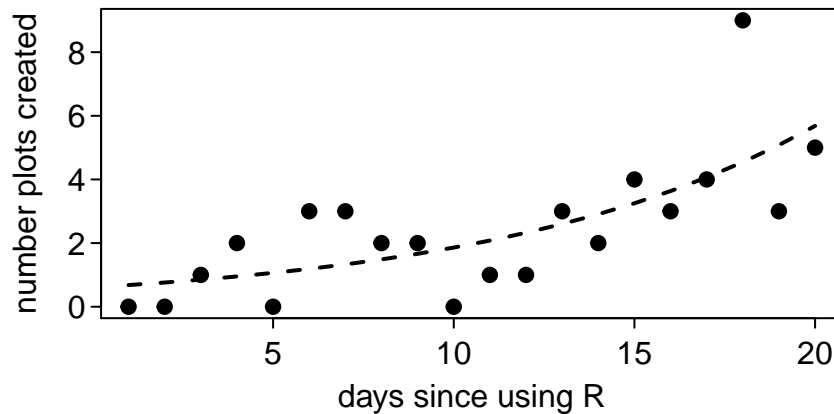


Figure 11: Example showing the use of the function `lines` to depict a non-linear relation between a predictor and a response variable.

5.1.3 Function segments

Yet another option to add lines to an existing plot is the function `segments`. This adds lines being defined by the coordinates of their start- (arguments `x0`, `y0`) and endpoints (arguments `x1`, `y1`). The function then adds a straight line connecting the start- and endpoint. As an example, lets add the median response as a line to a plot which depicts the response, separately for females and males. First, I need to generate some data and calculate the median, separately for the two levels of a factor 'sex':

```

set.seed(1)
sex=as.factor(sample(x=c("F", "M"), size=25, replace=T))
response=rnorm(n=25, mean=5, sd=1)
medians=tapply(X=response, INDEX=sex, FUN=median)

```

Note that the function `tapply` returns a named vector in which the values appear in the order in which the levels of the factor `sex` are indicated when calling the function `levels` (just try `medians` and `levels(sex)` to see that). Now the plot can be created and the lines added using

```

par(mar=c(2, 3.5, 0.5, 0.5), mgp=c(2.2, 0.5, 0), cex.lab=0.8, cex.axis=0.6)
plot(x=as.numeric(sex), y=response, xlim=c(0.5, 2.5), las=1, pch=19, xaxt="n", tcl=-0.2,
      xlab="", col=grey(level=0.25, alpha=0.5))
hll=0.25#create scalar indicating half the line length
segments(x0=(1:2)-hll, x1=(1:2)+hll, y0=medians, y1=medians, lwd=2)
mtext(text=levels(sex), at=1:2, line=0.2, side=1, cex=0.8)

```

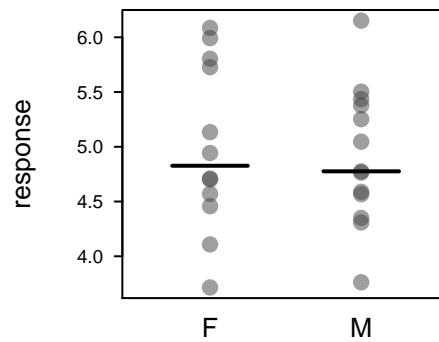



Figure 12: Example showing the use of the function `segments` to add lines to a plot.

Here (Fig. 12) I turn the factor `sex` into integer values using the function `as.numeric` which consecutively numbers (beginning with 1) the levels of the factor according to the order in which they are indicated by the function `levels`; hence, females will be displayed at $x=1$ and males at $x=2$ (see also above). Note that in order to the points not being displayed close to the left and right edge of the figure but closer to its center, I call the function `plot` with the argument `xlim` given the vector `c(0.5, 2.5)`. I also create an object `h11` which defines half the length of the lines to be added. The lines extend from 1 and 2 by `h11` in either direction. The argument `lwd` sets the 'line width' (with 1 being the default). Finally, I use the function `mtext` to add labels to the x-axis (see section 5.7.2) which here only makes sense after putting ticks along the x-axis had been suppressed (argument `xaxt` of the function `plot` set to `"n"`). Also note the use of the function `grey` which is explained in some detail in section 6.6. Finally, note that the function `segments` can be vectorized; that is, it can add as many segments as indicated by the values handed over to its arguments.

5.1.4 Function arrows

There is another function, `arrows`, which works very similar to `segments`. As its name suggests it draws arrows. The starts and ends the arrows are specified as for `segments` (i.e., the arguments are named `x0`, `x1`, `y0`, and `y1` for the x- and y-coordinates of the arrow's starts and ends, respectively). The length of the arrow heads (in inches) is specified using the argument `length`, and the angle (in degrees) is specified using the argument `angle` (Fig. 13). The option to have an `angle=90` can be used to display error bars. Finally, the argument `code` determines at which end of the line the arrow head is displayed. With `code=2` (the default) its shown at the end of the line (as specified by `x1` and `y1`); when `code=1` its shown at the line's start (`x0`, `y0`), and when `code=3` an arrow head is shown at both ends of the line (useful, for instance, in case one wants to show error bars).

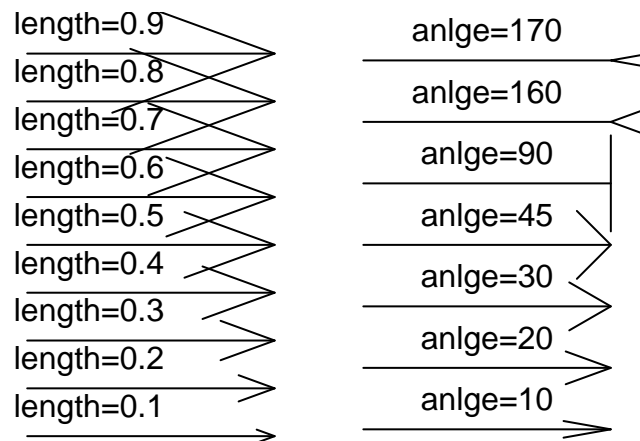


Figure 13: Parameters defining the appearance of arrow heads in the function `arrows`.

5.1.5 Defining the Appearance of Lines

Several arguments allow to define the appearance of lines (created, e.g., using the functions, `segments`, `lines`, `abline`, or `arrows`) and the borders of rectangles and polygons created using `rect` and `polygon`, respectively. The argument `lwd` (default: 1) sets the 'line width'; it accepts non negative numbers (depending on the operating system and the graphics device values between 0 and 1 might or might not work; and a `lwd=NA` will lead to no line visible). The argument `lty` defines the 'line type'. First, it can get handed over either integer numbers from 1 to 6 which lead to the line being solid (1), dashed (2), dotted (3), dotdash (4), longdash (5), or twodash (6) (Fig. 14a); alternatively, one can also hand over the name just indicated (i.e., `lty=3` and `lty="dotted"` both work and reveal the same). A peculiar (but very useful) way of defining the linetype is to indicate a string of up to eight characters being from 1 to 9 and "A" to "F" (e.g., `lty="1414"`). These values define the length of the 'on' and 'off' sections of the line, where "1" means shortest and "F" means longest. For instance, `lty="1614"` means one short 'on' section, a long 'off' section, a short 'on' section and an intermediate 'off' section (see Fig. 14b for a couple of examples). Note that when this form of definition of the line type is used the character string handed over *must* consist of 2, 4, 6, or 8 characters.

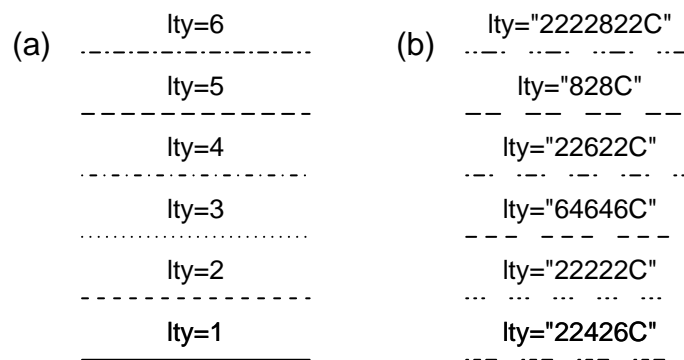


Figure 14: Examples of the use of the argument `lty`. Note that the three upper most in (b) depict the Morse code for F, M, and R, respectively.

5.1.6 Defining the Style of Line Ends (Argument `lend`)

That looks weird, I guess, 'defining the style of line ends'... However, when using R for plotting one might become picky, and then one might even bother about things like the style of line ends. For instance, when

'manually' creating boxplots for presentations in which the median is shown with a thick line, or when putting arrows to figures to be part of a presentation, the style of the line end might have some relevance. In this tutorial I use the argument in Fig. 42 and Fig. 57, and Fig. 15 shows what can be indicated to the argument `lend` and the values handed over to it reveal. I myself like the arrow with rounded ends (`lend=0`) more, and frequently use the function `segments` with `lend=1` to make sure that line ends exactly aligned with the boxes to which I add them (see Fig. 57 for an example, and try it with `lend=0`, `lend=1`, or `lend=2` to see that it matters).

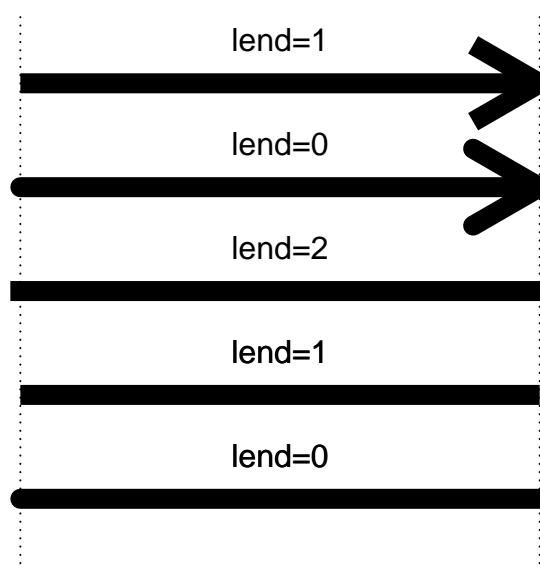


Figure 15: Illustration of the use of the argument `lend` handed over to the functions `segments` and `arrows`, respectively. The dotted vertical lines depict the lines' starts and ends; note that `lend=1` leads to line tips being exactly at their start and end, respectively, whereas with `lend=2` line tips extend a little beyond line starts and ends.

5.2 Adding Points (Function points)

Not really surprising, it is also possible to add points to a plot, and this can be achieved using the function `points` with the key arguments being `x` and `y` indicating the x- and y-coordinates of the points, respectively. Besides these, it accepts quite a few further arguments among which are `pch` and `cex` (to mention just two). A useful argument of the function `points` is actually `lwd` as it allows to adjust the thickness of the lines. For instance, when showing open circles and laying crosses, these might be easier to discern when being depicted with thicker lines. Fig. 16 and the code chunk below show an example. Beyond this I don't consider the function here any further because below there are several examples of its use.

```
set.seed(1)
n=40
xcov=runif(n=n, min=0, max=10)
xfac=as.factor(x=sample(letters[1:2], size=n, replace=T))
rv=xcov*c(0, 1)*as.numeric(xfac)+rnorm(n=n)
par(mar=c(3, 3, 0.2, 0.2), mgp=c(1.7, 0.3, 0), las=1, tcl=-0.15, mfrow=c(1, 2))
plot(x=xcov, y=rv, pch=c(1, 4)[as.numeric(xfac)], xlab="predictor", ylab="response", cex.axis=0.8)
legend("topleft", legend="(a)", bty="n")
plot(x=xcov, y=rv, pch=c(1, 4)[as.numeric(xfac)], xlab="predictor", ylab="response", lwd=2, cex.axis=0.8)
legend("topleft", legend="(b)", bty="n")
```

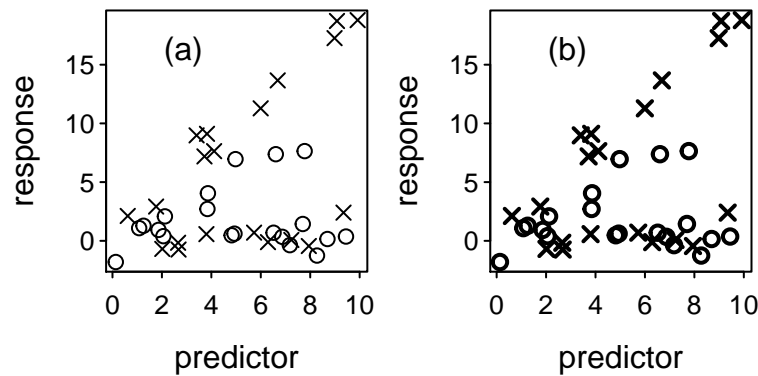


Figure 16: Illustration of the use of the argument `lwd` in combination with the function `points`. In (a) `lwd` is left at its default (1), whereas in (b) it is set to 2.

5.3 Adding Rectangles (Function `rect`)

Adding rectangles to a plot (or even creating a plot that doesn't show much else than rectangles) is something I do very frequently, for instance, when creating maps or when showing quartiles and medians in addition to the data. Here, I show an example for both applications.

Rectangles can be added using the function `rect` of which the key arguments are `xleft`, `xright`, `ybottom`, and `ytot`. These define the vertical and horizontal locations of the edges of the rectangle (note that rectangles added using the function `rect` always have edges being parallel to the x- and y-axes of the figure). Other arguments control how the rectangle is filled, with `col` defining the color, and `angle` and `density` defining the angle and density of shading lines filling the rectangle (the default is no shading lines), and `border` defining the color of the border of the rectangle. The following example shows how to draw a *map* where the magnitude of some value (e.g., altitude) is shown in grey scale and missing values are depicted using rectangles with shading lines (see Fig. 17):

```
set.seed(1)
locs=expand.grid(1:10, 1:10)#create grid with cell mid points
names(locs)=c("x", "y")#give locs nice names
#create altitude values including some NAs:
altitude=sample(c(5+rnorm(90), rep(NA, 10)), 100, replace=F)
par(mar=c(2, 2, 0.5, 0.5))#set margin widths for figure
par(mgp=c(0.7, 0, 0))#and where axis labels are displayed (first value)
hrw=0.5#define scalar with half the width of the rectangles
#create plot (note that type="n")
plot(x=1, y=1, axes=F, xlab="longitude", ylab="latitude", type="n", asp=1,
     xlim=range(locs$x)+c(-hrw, hrw), ylim=range(locs$y)+c(-hrw, hrw))
#standardize altitude to range 0 to 1:
alt.s=(altitude-min(altitude, na.rm=T))/(diff(range(altitude, na.rm=T)))
#add rectangles for altitude not being NA:
rect(xleft=locs$x[!is.na(alt.s)]-hrw, xright=locs$x[!is.na(alt.s)]+hrw,
     ybottom=locs$y[!is.na(alt.s)]-hrw, ytop=locs$y[!is.na(alt.s)]+hrw,
     border=grey(1-alt.s[!is.na(alt.s)]), col=grey(1-alt.s[!is.na(alt.s)]))
#depict cells with altitude being NA as shaded rectangles
rect(xleft=locs$x[is.na(alt.s)]-hrw, xright=locs$x[is.na(alt.s)]+hrw,
     ybottom=locs$y[is.na(alt.s)]-hrw, ytop=locs$y[is.na(alt.s)]+hrw,
     border=NA, col="black", angle=45, density=20)
```

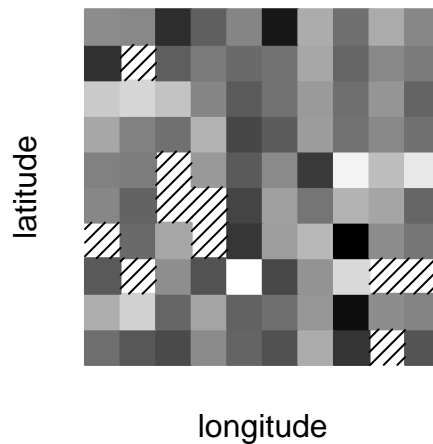


Figure 17: Illustration of the use of the function `rect` to display a surface of different values and NAs.

Some comments about this code might be helpful: the function `expand.grid` creates a data frame with one column for each of the vectors handed over and such that each of their combinations is included. The argument `mgp` handed over to the function `par` defines the margin lines at which axis labels, axis tick labels, and the axes itself (in that order) show up; here the first value is the only important one: it defines the axis labels to be shown close to the axis (the default is 3). The argument `asp` ('aspect ratio') handed over to the function `plot` ensures that the size of one unit is the same along the x- and y-axis, something very useful for displaying spatial data; and handing over `type="n"` leads to no points showing up in the plot. What is handed over to the arguments `xlim` and `ylim` is a cheap trick to ensure enough space being available for the rectangles to be displayed completely (i.e., the range of the locations \pm half the size of the rectangles to be added later, i.e. `hbw`). The standardization of altitude to a range from zero to one (`alt.s=(altitude-min(alitude, na.rm=T))/(diff(range(alitude, na.rm=T)))`) is needed since the function `grey` which I use below to color the rectangles only accepts values between 0 and 1 as input (the standardization is achieved by subtracting from each altitude value the minimum altitude and then dividing by the difference between the maximum and the minimum of altitude). The rectangles showing actually existing altitude values are finally added by the first call of the function `rect` which gets handed over only those data from the map/grid where altitude is not NA (everything indexed by `!is.na(alt.s)`). Particularly worth mentioning are the arguments `border` and `col` which set the color or the border and the area of the rectangle, respectively. Here I handed over `border` in addition to `col` to avoid small gaps between the rectangles which may otherwise show up (for details of color management see below). The final line adds rectangles to the plot where altitude values are missing. These don't get a border (`border=NA`) and are filled with diagonal lines (`angle=45`) and with 20 lines per inch `density=20`). Another example where I use rectangles a lot is when adding quartiles to scatterplots. For instance, to Fig. 12 I'd probably add quartiles as rectangles. I suggest to try that as an exercise.

5.4 Adding Polygons (Function `poly`)

Polygons are *shapes* more complicated than rectangles. They can be used to add complex figures to a plot (e.g., the circumference of a lake, a protected area, or a homerange) and are represented by two vectors indicating the x- and y-coordinates of the circumference of the object to be displayed (arguments `x` and `y`, respectively). Other than lines polygons always reveal a closed shape. To see the difference have a look at what the following lines of code reveal:

```
par(mar=rep(0.5, 4))
x=c(1, 3, 3)
y=c(1, 1, 2)
plot(x=x, y=y, axes=F, xlab="", ylab="", type="n", xlim=c(1, 6))
lines(x, y)
polygon(x+3, y)
```

Polygons can be created with various fillings and edges, and the arguments used for this have the same names and use as the respective arguments of `rect`, namely `angle` determining the angle of shading lines and `density` for their density. The color of the border and the area of the polygon are also defined as for the function `rect`, using the arguments `border` and `col`, respectively (see section 5.3 and the code below for details). Finally, the line type can be defined as usual, using the argument `lty` (see section 5.1.5). For a few examples see the following code chunk and Fig. 18.

```
#create two polygons:
nodes=seq(from=0, to=4*pi, length.out=6)+pi/2
poly1=cbind(x=cos(nodes), y=sin(nodes))
nodes=seq(from=0, to=2*pi, length.out=11)+pi/2
mult=sqrt((cos(pi/2-2*pi/10)*sin(pi/2-2*pi/5)/sin(pi/2-2*pi/10))^2+sin(pi/2-2*pi/5)^2)
mult=c(rep(c(1, mult), times=5), 1)
poly2=cbind(x=cos(nodes)*mult, y=sin(nodes)*mult)
#and plot them:
par(mar=rep(0.2, 4), mfrow=c(1, 5))
#poly 1:
plot(x=1, y=1, xlim=c(-1, 1), ylim=c(-1, 1), axes=F, xlab="", ylab="", type="n", asp=1)
polygon(x=poly1[, "x"], poly1[, "y"], border="blue", lwd=2)
#poly 2:
plot(x=1, y=1, xlim=c(-1, 1), ylim=c(-1, 1), axes=F, xlab="", ylab="", type="n", asp=1)
polygon(x=poly1[, "x"], poly1[, "y"], border="red", col="yellow", lwd=2)
#poly 3:
plot(x=1, y=1, xlim=c(-1, 1), ylim=c(-1, 1), axes=F, xlab="", ylab="", type="n", asp=1)
polygon(x=poly2[, "x"], poly2[, "y"], border="red", lwd=2, lty=3)
#poly 4:
plot(x=1, y=1, xlim=c(-1, 1), ylim=c(-1, 1), axes=F, xlab="", ylab="", type="n", asp=1)
polygon(x=poly2[, "x"], poly2[, "y"], border="red", lwd=2, density=10, angle=45, col="blue")
#poly 5:
plot(x=1, y=1, xlim=c(-1, 1), ylim=c(-1, 1), axes=F, xlab="", ylab="", type="n", asp=1)
polygon(x=poly2[, "x"], poly2[, "y"], border=NA, lwd=1, density=20, angle=90, col="blue")
polygon(x=poly2[, "x"], poly2[, "y"], border=NA, lwd=1, density=20, angle=0, col="blue")
```



Figure 18: Examples of polygons with various fillings and edges, created using the function `polygon` with the arguments `col`, `border`, `density`, `angle`, and `lty`.

5.5 Adding an Axis (Function axis)

Although perhaps not too commonly needed (well, I do that almost daily), it can be very useful to know how to manually add an axis to a figure. For instance, a model might have been fitted with the predictor being first log- and then z-transformed (to a mean of 0 and a standard deviation of 1) and one might want to show the impact of the predictor on the response (i.e., the data and the model), but with the x-axis showing tick labels at the original scale. In such a case one could, for instance, begin with creating the plot (including the data and the model) based on the first log- and then z-transformed predictor and finally add x-axis ticks and labels according to the original scale.

In the following example I take this approach. I begin with creating some data, running the model, and then showing it together with the data, whereby I suppress the x-axis to be shown (argument `xaxt="n"` in the call of the function `plot`; Fig. 19):

```

set.seed(1)
predictor=exp(rnorm(100))#create predictor
z.predictor=as.vector(scale(log(predictor)))#log and then z-transform it
response=log(predictor)+rnorm(100, sd=0.5)#create response
model.res=lm(response~z.predictor)#tun model
par(mar=c(3, 3, 0.5, 0.5), mgp=c(1.7, 0.4, 0), mfrow=c(1, 1))
#create the plot (note the argument xaxt="n")
plot(x=z.predictor, y=response, xlab="predictor", ylab="response", las=1, xaxt="n", tcl=-0.25)
abline(model.res, lty=2)#add regression line

```

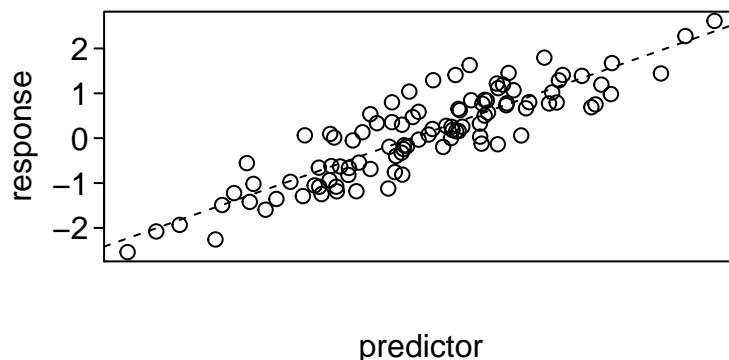


Figure 19: A simple data set and the respective model shown (used for illustration of the function `axis`).

To add the ticks to the x-axis at the original scale of the predictor one first needs to figure out the range of the original (i.e., untransformed) predictor using

```

range(predictor)

[1] 0.1091863 11.0410237

```

This suggests to show ticks at the values 0.25, 0.5, 1, 2, 4 etc. which can be achieved using `0.25*2^(0:5)`. A frequently very useful alternative is to use the function `pretty`. It suggests reasonable values at which to put a tick, based on a variable. For instance, with the predictor handed over it reveals

```

pretty(predictor)

[1] 0 2 4 6 8 10 12

```

Now one can add the axis. Here I'm not using what `pretty(predictor)` reveals because it wouldn't look very nice given the log-transformation of the predictor (see Fig. 20 for the result):

```

#note that the first 3 lines just create the plot again
par(mar=c(3, 3, 0.5, 0.5), mgp=c(1.7, 0.4, 0), tcl=-0.25)
plot(x=z.predictor, y=response, xlab="predictor", ylab="response", las=1, xaxt="n")
abline(model.res, lty=2)
xvals=0.25*2^(0:5)#create vector with values to be shown along the x-axis
#and add it
axis(at=(log(xvals)-mean(log(predictor)))/sd(log(predictor)), labels=xvals, side=1)

```

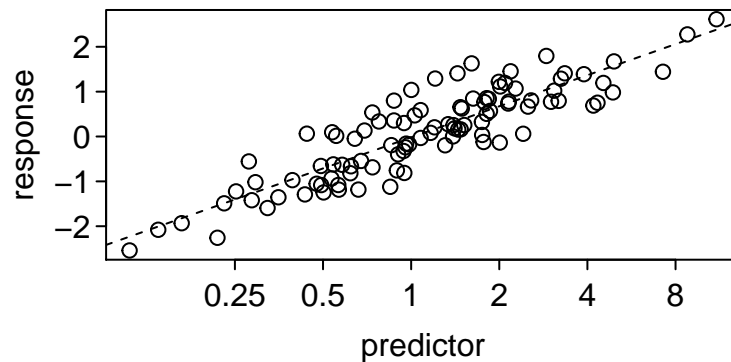


Figure 20: Illustration of an axis showing a variable (here the predictor) at its original scale although the model and the plot are based on it being log and then z-transformed (see text for more).

One piece of the code shown here is frequently puzzling beginners, namely what is handed over to the argument `at` of the function `axis`. So I should try to explain this: The vector `xvals` comprises a couple of values from the untransformed predictor for which we want to get ticks and labels at the x-axis. The plot itself, though, is based on the predictor first log- and then z-transformed. Hence, the same operations (a log- and then a z-transformation) need to be done with `xvals` to get the positions at which the ticks should be added. So what we need to do is to log-transform `xvals`, subtract from each of them the mean of the log of the predictor and finally divide the result by the standard deviation of the log of the predictor. This is what is done in the above code.

5.6 Adding a Legend (Function `legend`)

Quite frequently a legend might help getting the message accross. Of course, R has a function that adds a legend to an existing plot and the name of it is `legend` (surprised?). It has a whole bunch of arguments, and when it comes to more sophisticated legends one must be careful to not mess it up. For instance, it has an argument `cex` which defines the size of *texts* displayed and an argument `pt.cex` defining the size of the *symbols* (similarly it has two different arguments for defining the colors of texts and points, respectively). So when using this function to add a legend going beyond the basics, I frequently consult the help (`?legend` reveals that). Getting a simple legend, though, is totally easy. An example might illustrate the use of the function `legend`:

```
predictor=runif(n=100, min=0, max=10)#create predictor variable (covariate)
sex=as.factor(sample(x=c("F", "M"), size=100, replace=T))#create predictor variable (factor)
response=rep(x=NA, times=100)#create response
response[sex=="F"]=3+predictor[sex=="F"]/2+rnorm(n=sum(sex=="F"))#still creating the response
response[sex=="M"]=3+predictor[sex=="M"]/8+rnorm(n=sum(sex=="M"))#still creating the response
par(mar=c(3, 3, 0.5, 0.5), mgp=c(1.8, 0.5, 0))
plot(x=predictor, y=response, xlab="predictor", ylab="response", las=1, tcl=-0.25,
     pch=c(19, 1)[as.numeric(sex)])
#run a model and show it, separately for female and male data, respectively:
abline(lm(response[sex=="F"]~predictor[sex=="F"]), lty=1)
abline(lm(response[sex=="M"]~predictor[sex=="M"]), lty=2)
#add legend
legend(x="topleft", legend=levels(sex), pch=c(19, 1), lty=c(1, 2))
```

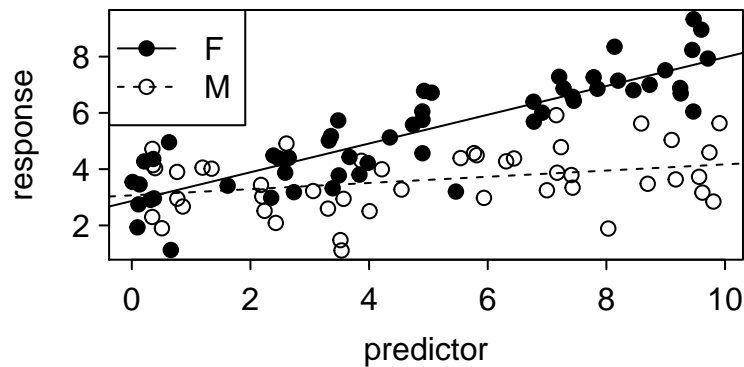



Figure 21: Figure with a simple legend created using the function `legend`.

A couple of comments might be helpful: the argument `x="topleft"` sets the position where the legend is placed. Of course other positions can be specified the same way (just check the help for `legend` for more such options). Another option to locate a legend is to hand over the x- and y-coordinates for the legend (arguments `x` and `y`, respectively). The next argument indicated (`legend`) specifies the text to be displayed in the legend. Here I used `levels(sex)` which reveals "F" "M" which appear in the legend in that order (from top to bottom). The argument `pch` defines the symbols to be displayed and `lty` the line types to be shown. In case no symbols or lines should be displayed just don't hand over anything to these arguments. One further argument I use quite frequently is `bty` which can be used to add a legend without a box around it (`bty="n"`). Usually I get along with such a basic legend, but, of course, there are occasions in which I don't. This is, for instance, the case when plotting colored gradients, and below I show how to get a legend for such plots as well (section 7, Fig. 42).

5.7 Adding Text

Frequently it is needed to add pieces of text to a plot, for instance, to add labels to individual observations or to display texts besides the axes. Fortunately, this is easy and a variety of functions for adding texts and options for adjusting their appearance does exist. Some of them are described in this section.

5.7.1 The Function `text`

The function `text` allows to add text to a plot. It gets handed over the x- and y-coordinates (arguments `x` and `y`) of the text to be added (argument `labels`). All these can be vectors, so its possible to add several text elements at a time. Let's begin with creating an empty plot using `plot(x=1, y=1, type="n")` and then adding text using `text(x=1, y=1, labels="hello")`. The text is centered and appearing with its middle at the specified position. Frequently, though, one doesn't want the labels to be shown right on top of position where they are to be added. This can be achieved using the argument `pos`. This allows to shift the text such that it is displayed besides, beneath or above the location (of the symbol). The argument `pos` takes the integer values from 1 to 4 which shift the text to below, the left, above and to the right of the specified point, respectively. Try the following code see what this reveals (Fig. 22):

```
par(mar=c(1.5, 1.5, 0.2, 0.2), mgp=c(1, 0.2, 0), tcl=-0.1)
plot(x=1, y=1, axes=T, xlab="", ylab="", ann=T, xaxt="n", yaxt="n")
axis(side=1, at=1)
axis(side=2, at=1, las=1)
text(x=1, y=1, labels=paste("pos", 1:4, sep=""), pos=1:4)
```

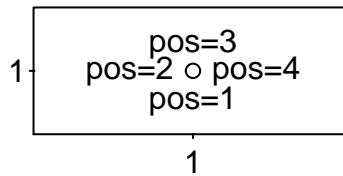


Figure 22: Illustration of the use of the argument `pos` of the function `text` to align labels around the position where they are printed.

Another (alternative) option is text alignment using the argument `adj`. This takes two values from the interval 0 to 1, specifying the adjustment parallel to the x- and y-axis, respectively, where 0 and 1 mean alignment to the left and right, respectively, (first value handed over to `adj`) and in vertical direction (second value). Both values need to be numbers between 0 and 1 (and a value of 0.5 means center alignment). The following two examples show left and right alignment (Fig. 23; see section 5.1.1 for more about the function `abline` which I use here to add a vertical line indicating the x-position of the texts):

```
par(mar=rep(0.2, 4), mfrow=c(1,2))
plot(x=1, y=1, axes=F, xlab="", ylab="", type="n")
text(x=0.8, y=c(0.9, 1), labels=c("hello", "world"), adj=c(0, 0.5))
abline(v=0.8, lty=2, col="grey")
mtext(text="(a)", side=3, line=-1) #see next section for an explanation of mtext
plot(x=1, y=1, axes=F, xlab="", ylab="", type="n")
text(x=0.8, y=c(0.9, 1), labels=c("hello", "world"), adj=c(1, 0.5))
abline(v=0.8, lty=2, col="grey")
mtext(text="(b)", side=3, line=-1)
```

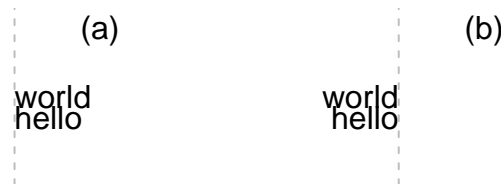


Figure 23: Illustration of the use of the argument `adj` (called with the function `text`) to align labels according to the position where they are printed. In (a) the text is left aligned (`adj=c(0, 0.5)`); in (b) it is right aligned (`adj=c(1, 0.5)`). The vertical lines indicate the vertical position handed over to the function `text`.

Another interesting option is to *rotate* the text using the argument `srt` ('string rotation'). This takes numbers representing the angle (in degrees) by which the text should be rotated, whereby positive numbers rotate the text to the left. Try the following code to see an example (Fig. 24):

```
par(mar=rep(0.2, 4), mfrow=c(1,2))
plot(x=1, y=1, axes=F, ann=F, type="n")
text(x=0.9, y=1, labels="srt=45", srt=45)
text(x=1.1, y=1, labels="srt=-45", srt=-45)
```

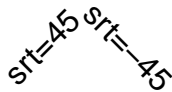


Figure 24: Illustration of the use of the argument `srt` (called with the function `text`) to rotate the text displayed.

5.7.2 The Function `mtext`

The function `mtext` allows to place text along the margins of a figure. The following are its most important arguments. The argument `side` specifies the side of the plot at which the text is added (with sides numbered from 1 to 4 indicating the sides beginning with the x-axis and counting in clockwise direction; Fig. 25). The argument `line` specifies the distance from the axis at which the text is displayed, with `line=0` meaning directly besides the axis (the maximum number of lines available is determined by the argument `mar` of the function `par`). Needless to say that the argument `line` accepts also non-integers and negative values. The argument `text` indicates the text to be displayed. The argument `at` specifies the position along the axis where the text is to be displayed (with the scale being the same as that of the respective axis); if it is not specified the text will be displayed in the middle of the axis.

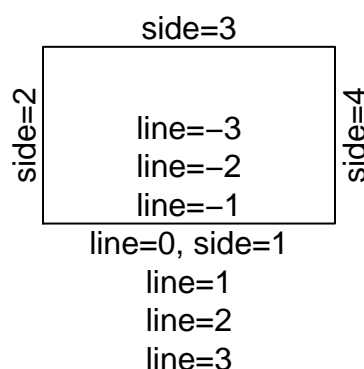
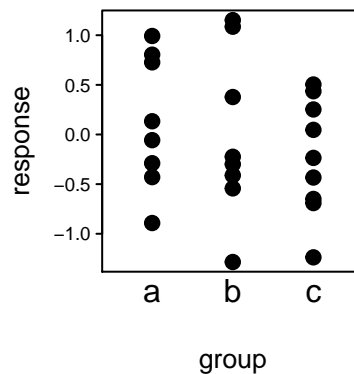


Figure 25: Illustration of the arguments `line` and `side` of the function `mtext`.

The following example (Fig. 26) shows how the function `mtext` could be used to get labels besides an axis:

```
set.seed(1)
group=as.factor(x=sample(x=letters[1:3], size=25, replace=T))
response=rnorm(n=25)
par(mar=c(3, 3, 0.2, 0.2), mgp=c(1.7, 0.3, 0), cex.lab=0.8, cex.axis=0.6, tcl=-0.15)
plot(x=as.numeric(group), y=response, xlab="group", ylab="response", xaxt="n", xlim=c(0.5, 3.5),
     las=1, pch=19)
mtext(text=levels(group), at=1:3, line=0, side=1)
```

Figure 26: Illustration of the use of the function `mtext`.

Note that I used `as.numeric(group)` as the x-values which reveals 1 for 'a', 2 for 'b', and 3 for 'c' (compare `levels(group)`). Furthermore, I set the limits of the x-axis manually to a range from 0.5 to 3.5 to achieve a plot in which the points are not too close to the left and right margin of the plot. Finally, I used `xaxt="n"` to suppress showing ticks at the x-axis (the numbers which otherwise would be shown are meaningless in case of a factor like here).

5.7.3 Displaying long axis labels

Sometimes you might struggle with very long axis labels. For instance, you might have some observations for a bunch of countries and want to plot them against country. Showing countries with longer names below the x-axis, however, requires to rotate them by 90 such that they don't overlap (this can be achieved setting the argument `las` to 1), and it also requires a very wide margin at the x-axis. The following code chunk shows an example:

```
#create some data:
countries=as.factor(rep(c("American Samoa", "Brunei Darussalam", "Burkina Faso", "Cayman Islands",
  "Cocos Islands", "Dominican Republic", "Equatorial Guinea", "Papua New Guinea"), each=8))
set.seed(1)
rv=rnorm(n=length(countries))
#set margins etc. (note the wide margin at the x-axis):
par(mar=c(8, 3, 0.2, 1.25), mgp=c(1.7, 0.3, 0), tcl=-0.15, las=1, mfrow=c(1, 3))
#begin with plot (x-axis) is suppressed using xaxt="n":
plot(x=as.numeric(countries), y=rv, xaxt="n", xlab="", pch=19, col=grey(level=0.5, alpha=0.5))
#add text to lower margin:
mtext(text=levels(countries), side=1, at=1:length(levels(countries)), las=2, line=0.2, cex=0.7)
```

The resulting Figure (Fig. 27a) isn't really too friendly as the names of the countries are oriented such that one has to turn one's head quite a bit to read them. So some rotation of the country names would be nice. Unfortunately, the function `mtext` doesn't evaluate the argument `srt` which allows to rotate text when using the function `text`. The function `text`, in turn, can't be used to add text outside the plotting region, you might think. But that isn't true! In fact, the function `par` has an argument named `xpd` which allows to set the range within which elements can be added to a plot. This argument has a default of `FALSE` in which case all plotting (e.g., use of functions like `points`, `lines`, `text`, etc.) is clipped to the plotting region. However, it accepts also the values `TRUE` in which case plotting is clipped to the figure region and `NA` which allows plotting in the entire device⁶. So what we can do is to set `xpd` to `T`. Following that we can use the function `text` to add text outside the plotting region below the x-axis (see Fig. 27b for the result):

```
plot(x=as.numeric(countries), y=rv, xaxt="n", xlab="", pch=19, col=grey(level=0.5, alpha=0.5))
par(xpd=T)
text(labels=levels(countries), x=1:length(levels(countries)), y=par()$usr[3]-(par()$usr[4]-par()$usr[3])*0.025,
  srt=45, cex=0.8, adj=1)
```

⁶Fig. 27 allows to explain the difference between the 'figure' and the 'device' region: here the device encompasses three Figures

The above code obviously needed some trial and error until I was satisfied with the result. But probably I should first explain what `par()$usr` is. This argument of `par` reveals the coordinates of the left, right, bottom, and top edge of the plotting region (in that order). Originally, I'd set `y=par()$usr[3]`, but then the labels ended to close to the x-axis. That's why I subtracted `(par()$usr[4]-par()$usr[3])*0.025` which corresponds to lowering the text by 2.5% of the height of the plotting region. The `adj=1` means that the text is right-adjusted (see also section 5.7.1).

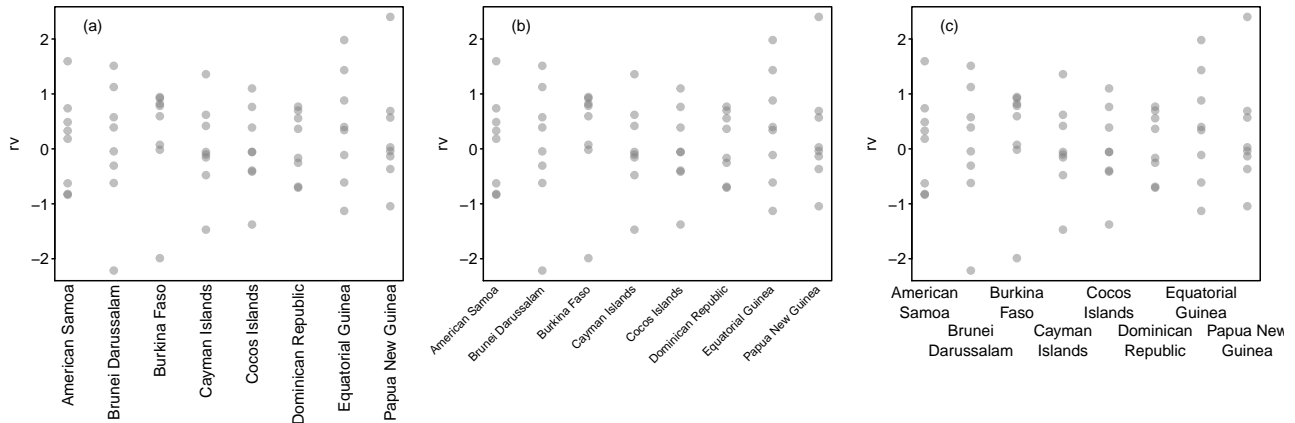


Figure 27: Options to show long names along an axis.

There are certainly other options to display long text along an axis, and the following example uses two of them at the same time. First, one could show the labels alternately at different distances from the axis. Second, it's also possible to have line breaks in labels. This latter can be achieved by use of the `\n` inserted into a text. `\n` is the R way of entering a line break into a piece of text. The following code chunk shows how this works:

```
plot(as.numeric(countries), rv, xaxt="n", xlab="", pch=19, col=grey(level=0.5, alpha=0.5))
#include line breaks:
countries2=c("American\nSamoa", "Brunei\nDarussalam", "Burkina\nFaso", "Cayman\nIslands",
             "Cocos\nIslands", "Dominican\nRepublic", "Equatorial\nGuinea", "Papua New\nGuinea")
mtext(text=countries2, side=1, at=1:length(levels(countries)), line=c(1.2, 3.2), cex=0.7)
```

Here the argument `line` of the function `mtext` leads to the alternating distances from the x-axis (see Fig. 27c for the result and section 5.7.2 for more about `mtext` and its arguments). Note that in the above call of `mtext` a vector of length 8 is handed over to the argument `text` but only a vector of length 2 to the argument `line`. This is possible because of the concept of 'recycling' (which basically means that the shorter vector gets repeated (or 'recycled') as many times as needed until the length of the longer vector is reached). I feel, of the three options shown here, I like this last version most.

5.8 Mathematical Symbols, Equations, Annotations, etc.

R wouldn't be R if it were not possible to get all sorts of cool symbols, mathematical expressions and the like into plots. Getting such things into plots is not always totally straightforward, I'd say, and one needs to get acquainted to it (but I don't have too much experience with this topic). The perhaps most important function in this context is `bquote`, and also very important is the help that `?plotmath` reveals. Getting mathematical expressions, Greek characters, superscripts and other such things into plots as texts, axis labels, etc. works by first using the function `bquote` to interpret an appropriate *expression* and then handing its result over to the appropriate argument of the respective function (e.g., `labels` of the function `text`; `text` of the function `mtext`; `xlab`, `ylab`, or `main` of the function `plot`, etc.). What can be handed over to `bquote` is explained in the help of `plotmath`. Some of the things I use/need most frequently are explained in the following.

One thing being quite particular with the function `bquote` is that one doesn't use quotes where one would usually put them when using other R functions. For instance, for getting an expression like ' $a+b=c$ ' into a plot one

would just use `plot(1, 1, type="n", axes=F, ann=F)` and then `text(x=1, y=1, labels=bquote(a+b==c))` (notice the two equality signs used here, to get one into the plot; see below for more).

alpha: α	nu: ν	Alpha: A	Nu: N
beta: β	xi: ξ	Beta: B	Xi: Ξ
gamma: γ	omicron: \omicron	Gamma: Γ	Omicron: O
delta: δ	pi: π	Delta: Δ	Pi: Π
epsilon: ϵ	rho: ρ	Epsilon: E	Rho: P
zeta: ζ	sigma: σ	Zeta: Z	Sigma: Σ
eta: η	tau: τ	Eta: H	Tau: T
theta: θ	upsilon: υ	Theta: Θ	Upsilon: Y
iota: ι	phi: ϕ	Iota: I	Phi: Φ
kappa: κ	chi: χ	Kappa: K	Chi: X
lambda: λ	psi: ψ	Lambda: Λ	Psi: Ψ
mu: μ	omega: ω	Mu: M	Omega: Ω

Figure 28: Names of Greek symbols as they can be used in, e.g., `bquote`.

5.8.1 Greek Symbols

You get Greek symbols (α , β , γ , etc.) by just naming them. For instance, `bquote(alpha)` reveals α , `bquote(beta)` reveals β , and so on. Capital Greek symbols you get by just spelling the name of the symbol with the first character being capitalized (e.g., `bquote(Alpha)`). For an overview about the available symbols and their names see Fig. 28. The function `bquote` works in combination with, e.g., the functions `text` and `mtext`, and it also works as a value handed over to the arguments `xlab`, `ylab`, and `main` of the function `plot`. As an example (Fig. 29), try the following code chunk to see that.

```
par(mar=c(1.5, 1.5, 3, 1.2), mgp=c(0.25, 0, 0))
plot(x=1, y=1, axes=F, type="n", xlab=bquote(alpha), ylab=bquote(beta), main=bquote(gamma))
box()
text(x=1, y=1, labels=bquote(Omega))
mtext(side=4, line=0, text=bquote(Sigma))
```

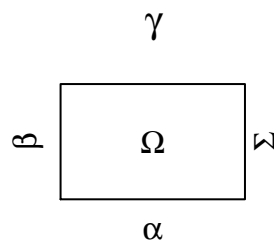


Figure 29: Examples of greek symbols used in the annotation of a plot got into the plot by handing over the results of `bquote` to various arguments of the functions `plot`, `mtext` and `text`.

5.8.2 Mathematical Symbols and Expressions

Other special characters that are quite frequently used in mathematics and statistics are characters with a bar above it (e.g., \bar{x} for the mean of x or \hat{y} for fitted values). These can be got using `bquote(bar(x))` and `bquote(hat(y))`, respectively; and, of course, they also work with other characters. The infinity symbol (∞) one gets using `bquote(infinity)`. See `?plotmath` for many more options.

Most mathematical expressions are rather straightforward, that is you just type them as you would do it in R. Specifically, you get the symbols for addition, subtraction, multiplication, and division using `+`, `-`, `*`, and `/`, respectively. As usual in mathematics, the multiplication sign is not displayed; if you want it being displayed

use `%%` instead of `*`. Similarly, you get $>$, \geq , $<$, \leq , and \neq by using `>`, `>=`, `<`, `<=`, and `!=`, respectively. However, the equality sign is expressed as `==`. Also a squareroot is possible. For instance, the Pythagorean theorem could be displayed using `bquote(sqrt(a^2+b^2)==c)`. Angles with degrees, minutes and seconds can be displayed using, e.g., `bquote(10*degree*11*minute*32*second)` (Fig. 30).

Some formats I need frequently are *subscripts* and *superscripts*, which can be got using `bquote(x[i])` and `bquote(R^2)`, respectively. Other formats like *italic*, *bold*, and *bolditalic* can be got using `bquote(bold(a+b))`, `bquote(italic(a+b))` and `bquote(bolditalic(a+b))`, respectively. Figure 30 shows some examples for how to get mathematical expressions into plots. The respective expression could be handed over to, e.g., the arguments `xlab` and `ylab` of the function `plot` or the arguments `text` and `labels` of the functions `mtext` and `text`, respectively.

(a)	<code>bquote(10*degree*11*minute*32*second):</code>	$10^{\circ}11'32''$
(b)	<code>bquote(1/0==infinity):</code>	$1/0 = \infty$
(c)	<code>bquote(a^2+b^2==c^2):</code>	$a^2 + b^2 = c^2$
(d)	<code>bquote(sqrt(a^2+b^2)==c):</code>	$\sqrt{a^2 + b^2} = c$
(e)	<code>bquote(1+1!=1):</code>	$1 + 1 \neq 1$
(f)	<code>bquote(e^{i*pi}== -1):</code>	$e^{i\pi} = -1$
(g)	<code>bquote(paste("abundance ", group("[",nr,~~ind./km^2,""]))):</code>	abundance [nr. ind./km ²]
(h)	<code>bquote(paste("abundance ", group("[",frac(number~~individuals, km^2),""]))):</code>	abundance $\left[\frac{\text{number individuals}}{\text{km}^2} \right]$

Figure 30: Examples of mathematical expressions and annotations and how to get them into plots. Notice the curly braces in (f) which mean invisible grouping (round brackets would be displayed in the plot).

5.9 Getting Entries out of R Objects into Plots

A little tricky but, of course, totally doable is the task of getting numerical values out of R objects into a plot. For instance one might want to show the R^2 of a regression in a figure or also add a fitted model's equation. The tricky thing is that if one would use `bquote(R^2==model.res$r.squared)` (with `model.res` comprising the summary of the result of a call of the function `lm`) then what one would get to see in the plot is pretty much `'R^2=model.res$r.squared'` (well, something similar), but not the *value* of R^2 . To get that, one needs to use `bquote(R^2==(model.res$r.squared))`. The expression `.(<object>)` leads to the *value* represented by '`<object>`' being displayed rather than just the word '`<object>`'. Here is an example in which I put the R^2 and also the model equation into the plot (see Fig. 31 for what it reveals):

```
set.seed(1)#generate some data:
x=runif(n=25, min=0, max=5)#predictor
y=x+rnorm(n=25, sd=3)#response
par(mar=c(3, 3, 0.2, 0.2), mgp=c(1.8, 0.5, 0), cex.axis=0.8, tcl=-0.25)#set margins etc,
plot(x=x, y=y, xlab="predictor", ylab="response", las=1, pch=19, tcl=-0.25)#create plot
model.res=lm(y~x)#run model
abline(model.res)#add model line
coeffs=coefficients(model.res)#extract coefficients
model.res=summary(model.res)#extract model summary
```

```
#get rounded coefficients into the plot:
text(x=min(x), y=max(y)-diff(range(y))*0.05, adj=c(0, 0.5),
     labels=bquote(y== .(round(coef[1], 2)) + .(round(coef[2], 2))*x))
#get rounded R^2 into the plot:
text(x=min(x), y=max(y)-diff(range(y))*0.2, adj=c(0, 0.5),
     labels=bquote(R^2==.(round(model$res$r.squared, 3))))
```

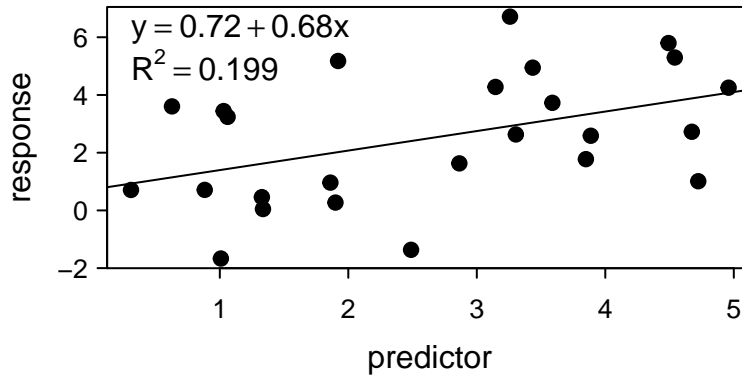


Figure 31: Illustration of the use of `bquote` to get a model result into a figure

Note that both model coefficients and R^2 are rounded (function `round`) because otherwise we would get to see many decimals (which does neither look nice nor its very informative). Note also how I extract the respective coefficients, round them and hand the result over to `.()` (in this order; the same is done with the R^2). Note also that characters like "R" or "x" are not indicated in quotes (which I find kind of weird). Finally, note how the position of the text is determined. Setting the position in x-direction to the minimum of the x-values (`x=min(x)`) and the position on y-direction to the minimum of the y-values minus some fraction of their range (`y=max(y)-diff(range(y))*0.05` in the first call of `text`) has the advantage that the text will appear in the topleft corner of the plot, without needing to manually figure out where exactly this would be. The argument `adj` is used to left align both texts such that they appear nicely aligned below one another.

To finally get also the P-value displayed following the model equation, we need yet another trick which is the *expression list*. This can be used to display several expressions, separated by a comma (e.g., `bquote(list(a==1, b==sqrt(2)))`). The function `list` as used in this context (e.g., within a call of the function `bquote`) is called in the form `list(<expression>, <expression>, ...)` where "..." indicates that more expressions could follow. Here I use it to append the P-value behind the regression equation (see Fig. 32; note how 'P' is italicized):


```

par(mar=c(3, 3, 0.2, 0.2), mgp=c(1.8, 0.5, 0), cex.axis=0.8, tcl=-0.25)#set margins etc,
plot(x=x, y=y, xlab="predictor", ylab="response", las=1, pch=19)
abline(lm(y~x))#add model line
coeffs=model.res$coefficients
#get the coefficients and the P-value into the plot:
text(x=min(x), y=max(y)-diff(range(y))*0.05, adj=c(0, 0.5),
     labels=bquote(list(y== .(round(coeffs[1, 1], 2)) + .(round(coeffs[2, 1], 2))*x,
                       italic(P)==.(round(coeffs[2, 4], 3)))))
#get R^2 into the plot:
text(x=min(x), y=max(y)-diff(range(y))*0.2, adj=c(0, 0.5),
     labels=bquote(R^2==.(round(model.res$r.squared, 3))))

```

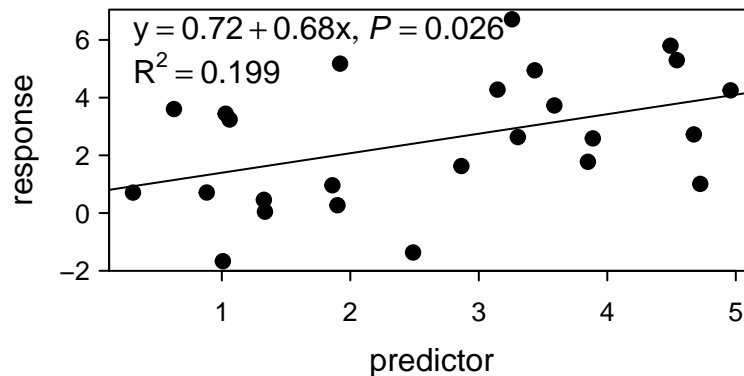


Figure 32: Illustration of the use of `bquote` to get a model result into a figure

To practice the use of `bquote` etc. a little more, here's another example, showing how one could get an F-statistic and its associated degrees of freedom ('df') into a figure. I chose this example because there are different conventions of indicating it, namely "F(df numerator, df denominator)=F-value" (frequently used in psychological journals) and "F_{df numerator, df denominator}=F-value" (frequently used in animal behavior and ecology). One way to get this is as follows:

```

par(mar=rep(0.1, 4))
plot(x=1, y=1, ann=F, axes=F, type="n", ylim=c(0.8, 1.2))#create empty plot
Fstat=model.res$fstatistic#just to make the code slimmer
#F-statistic as used in psychology:
text(x=1, y=1, pos=3, labels=
     bquote(paste(F, group("(", list(. (Fstat[2]), . (Fstat[3])), ")"), "=", .(round(Fstat[1], 2)))))
#F-statistic as used in animal behaviour and ecology:
text(x=1, y=1, pos=1, labels=
     bquote(paste(F[list(. (Fstat[2]), . (Fstat[3]))], "=", .(round(Fstat[1], 2)))))

```

$$F(1, 23)=5.7$$

$$F_{1, 23}=5.7$$

Figure 33: Illustration of the use of `bquote` to indicate an F-statistic in a figure.

Admittedly, the code to get expressions like shown in Fig. 33 is complex (and I'm almost certain that there are simpler ways to get this), but it works. Some comments might help to understand it (but I have to emphasize here that I'm not at all an expert regarding this, so my explanations might be pretty wrong): the function `bquote` provides the context allowing interpretation of what follows; `group` allows to embrace an expression by brackets; `list` concatenates expressions whereby it separates them by a comma, here it is used

to get the two degrees of freedom together; which, in turn, are retrieved within a call of `.()` to get the values and not the literal words displayed; expressions indicated in square brackets are displayed as subscripts; and the function `paste`, finally, glues all that together (but note that its use inside a call of `bquote` is somewhat differing from its use outside this context).

6 Colors

Colors can be very helpful to visualize things, and the (truly amazing) coloration options of R provide lots of options to use colors to illustrate things. In the following I'll first treat the most basic ways of getting colors, and then proceed with more advanced options.

6.1 Colors by Name or Number

Perhaps the most simple way of defining colors is to simply *name* them. A document showing all 657 colors that can be addressed by their names can be found at <http://research.stowers-institute.org/efg/R/Color/Chart/ColorChart.pdf>. To use these, one simply hands over the respective name(s) to the argument `col` which can be handed over to all functions adding graphical elements treated so far. A simple example would be (see Fig. 34 for the result)

```
par(mar=rep(0.1, 4))
plot(x=1:10, y=rep(1, 10), ann=F, axes=F, pch=19, cex=3, bty="n",
     col=c("black", "red", "darkorange4", "chartreuse2", "cyan2", "coral1", "aquamarine3",
           "darksalmon", "burlywood1", "deeppink1"))
```



Figure 34: Illustration of how colors can be addressed by their names.

Of course, such a way of getting colors is pretty laborious, and I use it only very rarely (e.g., if I need a small set of highly distinctive colors, although there are better ways of achieving this; see below). These same 657 colors can also be addressed by their respective *numbers* (which are also indicated in the pdf mentioned above). This is done by using the function `colors` which reveals just the names of the colors available, namely by addressing its output with the numbers of the colors desired. So the exact same plot as shown in Fig. 34 could also be got using

```
par(mar=rep(0.1, 4))
plot(x=1:10, y=rep(1, 10), ann=F, xaxt="n", yaxt="n", pch=19, cex=3, bty="n",
     col=colors()[c(24, 552, 94, 49, 70, 58, 11, 101, 38, 117)])
```

A small set of eight different colors can also be addressed by just an integer *number*, which I frequently use to get a quick overview about the data when there is a factor with just a few levels in the data. The following scatterplot in which points are colored according to which level of a factor they belong to illustrates this (Fig 35; see above for how to add a legend to such a plot):

```
set.seed(1)
xdata=data.frame(x=runif(100), y=runif(100), species=sample(letters[1:5], 100, replace=T))
par(mar=c(3, 3, 0.2, 0.2), mgp=c(1.8, 0.5, 0))
plot(x=xdata$x, y=xdata$y, pch=19, ann=F, las=1, tcl=-0.25, col=as.numeric(xdata$species))
```

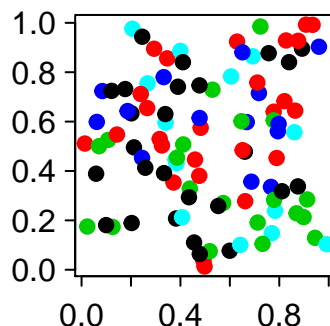


Figure 35: Illustration of how colors can be addressed using just integer numbers from 1 to 8.

Of course, there are more than just 657 different colors available in R, and, in fact, one can get and mix colors in various ways. Some of which I show in the following.

6.2 The Function rainbow

Sometimes it is desired to get a certain number of colors spanning the range from red to purple (i.e., along the rainbow) at 'equal' intervals (whatever that precisely means for colors), that is with colors being as different as possible along that gradient. This can be achieved using the function `rainbow`. To this function at least one integer number (argument `n`) has to be handed over, and then it will create `n` different colors spanning the rainbow at 'equal' intervals. A simple example may illustrate this (Fig 36):

```
n=20
par(mar=rep(0.1, 4))
plot(x=1:n, y=rep(1, n), pch=19, cex=2, ann=F, xaxt="n", yaxt="n", bty="n", col=rainbow(n))
```



Figure 36: Illustration of the function `rainbow`.

Potentially, this might be useful when it comes to coloring the levels of factors with more than eight levels, but, to be honest, I frequently found the colors I obtained too similar and hence preferred other ways of getting `n` different colors. By the way, when you look at what `rainbow(n)` reveals: these are numbers in 'hexadecimal' notation (those being familiar with programming might be able to make sense out of these values).

6.3 Colors in red-green-blue Space (Function `rgb`)

It is also possible to define colors according to their values assigned to the three base colors *red*, *green*, and *blue*. This is done using the function `rgb` which 'mixes' colors as in our perception of light; that is, a maximal value

for each of red, green, and blue reveals *white*, and a minimal value for all the three colors reveals *black*.

I rarely use this function but it could, for instance, be used to create a color gradient. As an example to illustrate the use of the function `rgb` to depict a gradient I use the matrix `volcano` which comes with R (it represents a 'digital elevation model' of the Maunga Whau (Mt Eden); use `?volcano` to learn more about this data set). To plot it with ease one first needs to transform the rectangular matrix into a data frame with three columns, one for longitude, latitude, and altitude, respectively. This can be achieved using

```
volc2=volcano
rownames(volc2)=1:nrow(volcano)#assign rownames (usually this will be latitude values)
colnames(volc2)=1:ncol(volcano)#assign rownames (usually this will be longitude values)
volc2=as.data.frame(as.table(volc2))#turn into data frame
names(volc2)=c("latitude", "longitude", "altitude")#give it nice names
#turn latitude and longitude into numeric:
volc2$latitude=as.numeric(as.character(volc2$latitude))
volc2$longitude=as.numeric(as.character(volc2$longitude))
```

Note that here I could also have used `as.numeric` directly without turning the factors `latitude` and `longitude` into character vectors first (i.e., without using the function `as.character` inside the function `as.numeric`, but since this is most commonly needed I used it here as well).

The next step is to create a vector with altitude values being in a defined range. This can be pretty much everything, but I prefer a range from 0 to 1 because it is most easily handled in the steps following. For this I'd use

```
alt.s=(volc2$altitude-min(volc2$altitude))/diff(range(volc2$altitude))
```

Next one can create the plot. For this I'd use pretty much the same procedure as I used for Fig. 17; that is, I'd add rectangles (actually squares) with the color varying along the altitude. Here I use *white* to indicate low altitudes and *blue* to indicate high altitudes, but, of course, red or green could be used equally well to indicate high altitudes (see Fig. 37 for the result).

```

par(mar=c(2, 2, 0.5, 0.5))#set margin widths for figure
par(mgp=c(0.7, 0, 0))#and where axis labels are displayed (first value)
plot(x=volc2$longitude, y=volc2$latitude, axes=F, xlab="longitude", ylab="latitude", type="n",
     asp=1, xlim=range(volc2$longitude)+c(-0.5, 0.5), ylim=range(volc2$latitude)+c(-0.5, 0.5))
hrw=0.5#define scalar with half the width of the rectangles
#create vector with color values:
col.val=rgb(red=1-alt.s, green=1-alt.s, blue=1)
#add rectangles:
rect(xleft=volc2$longitude-hrw, xright=volc2$longitude+hrw,
     ybottom=volc2$latitude-hrw, ytop=volc2$latitude+hrw,
     border=col.val, col=col.val)

```

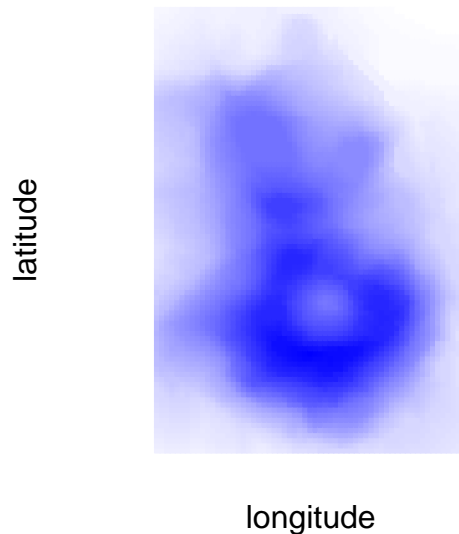


Figure 37: Illustration of the use of the function `rgb` to display a digital elevation model using a white-to-blue color gradient.

The logic of the color definition is as follows: when the altitude is low (i.e., `alt.s=0`) the respective cell should be shown in white, i.e., with maximal values for `red`, `green`, and `blue`. Hence, `blue` should be 1, as should be `red` and `green` (hence, the latter two values being `1-alt.s`). When the altitude is high (i.e., `alt.s=1`) the respective cell should be depicted in blue, i.e., a maximal value for `blue` but minimal values for `red` and `green`. This is achieved by letting the value for `blue` being 1 again, but the values for `red` and `green`, respectively, being 0 (i.e., `1-alt.s`). So what we, in fact, do is to define the gradient from white (i.e., minimal) to blue (i.e., maximal altitude) by setting `blue` to its maximal value throughout but 'turning off' the other two colors (`red` and `green`) as altitude decreases.⁷

6.4 White-to-Black gradient (Function `grey`)

Also if one doesn't like colors in papers as I do (I feel that also people not having color printers (probably the majority of the people living on this planet) should be able to read our plots) one could still depict such a gradient using grey scales. This can be done using the function `grey`. The key argument of this function is `level` which indicates the magnitude of 'non-greyness', where a value of 1 means *white* and a value of 0 means *black* (admittedly somewhat counter intuitive at a first glance; but the concept fits well with the general concept of defining colors in R).

⁷Some of you might ask themselves 'why the heck doesn't the guy just use the function `image` for that?' Well, frequently I need to later add other elements (like the shape of a national park, the boundaries of homeranges of animals, the locations of streets or where certain animals have been detected, or customized legends) to such a plot; and to me this seems much easier when the figure is based on the function `plot` rather than `image`.

I'm using the same example (i.e., `volcano`) as above to show how it works. The code differs only in one line from that used to create Fig. 37, namely the one defining the color, where here I use the function `grey` rather than `rgb`. Note that I hand over `1-alt.s` to the argument `level` of the function `grey`, meaning that higher altitudes are displayed with darker pixels. The plot is created as follows, and the result can be seen in Fig. 38:

```
par(mar=c(2, 2, 0.5, 0.5))#set margin widths for figure
par(mgp=c(0.7, 0, 0))#and where axis labels are displayed (first value)
plot(x=volc2$longitude, y=volc2$latitude, axes=F, xlab="longitude", ylab="latitude", type="n",
     asp=1, xlim=range(volc2$longitude)+c(-0.5, 0.5), ylim=range(volc2$latitude)+c(-0.5, 0.5))
hrw=0.5#define scalar with half the width of the rectangles
#create vector with values for non-greyness:
col.val=grey(level=1-alt.s)
#draw rectangles:
rect(xleft=volc2$longitude-hrw, xright=volc2$longitude+hrw,
     ybottom=volc2$latitude-hrw, ytop=volc2$latitude+hrw,
     border=col.val, col=col.val)
```

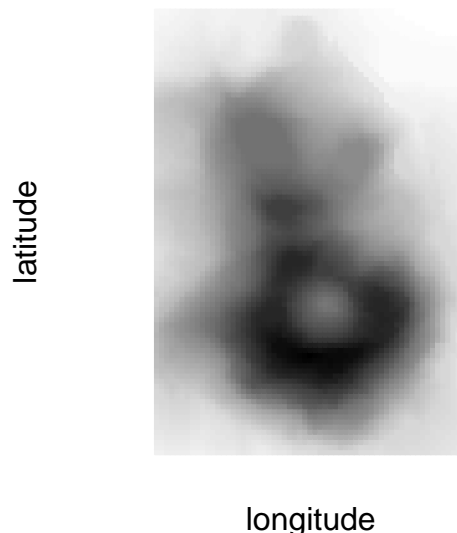


Figure 38: Illustration of the use of the function `grey` to display a digital elevation model using a white-to-black color gradient.

6.5 Custom Color Gradients (Function `colorRamp`)

I want to illustrate one further option to indicate a gradient by color which is to create a color 'ramp'. Probably this is the most useful one among the options to display a gradient by color. In fact, in Fig. 37 low altitudes are just pale... Assume, however, a situation where one wants to plot the degree of threat for wild great apes and show save areas in green and areas of high threat in red (and areas of intermediate threat in, say, yellow). Surely, it would probably be possible to create the respective color coding using the function `rgb`, but that would probably be somewhat tricky. A simpler option is to use the function `colorRamp`. This gets handed over a vector of colors (argument `colors`) and returns a function. This function, in turn, expects values between 0 and 1 as input, which it turns into corresponding colors (in red-green-blue space). Using the example, of the volcano again and aiming to depict low altitudes in blue, intermediate altitudes in yellow, and high altitudes in red, the function would be created using `ramp=colorRamp(c("blue", "yellow", "red"))`. The resulting function (`ramp`) would then turn a value of 0 into blue, a value of 0.5 into yellow, a value of 1 into red, and values in between these into the respective intermediate colors. However, the resulting object is a matrix with three columns, one for the red, green, and blue component (with values ranging from 0 to 255), and this needs to be handed over to the function `rgb` to transform it into colors which can be handed over to the argument

`col` of the function `rect` (and others). This can be done using `rgb(red=ramp(alt.s), maxColorValue=255)`, where the result of the call of `ramp` is directly handed over to `rgb` which accepts as the argument `red` also a three columns matrix assumed to indicate the red, green, and blue component. As stated above, the matrix produced by `ramp` has values ranging between 0 and 255; however, `rgb` by default expects values between 0 and 1. That's why one needs to use the argument `maxColorValue` to which one has to hand over 255. The entire code is as follows (notice that only the two lines where `ramp` and `col.val` are defined differ from the code used to create Fig. 37 and 38, respectively; for the result see Fig. 39):

```
par(mar=c(2, 2, 0.2, 0.2))#set margin widths for figure
par(mgp=c(0.7, 0, 0))#and where axis labels are displayed (first value)
plot(x=volc2$longitude, y=volc2$latitude, axes=F, xlab="longitude", ylab="latitude", type="n",
     asp=1, xlim=range(volc2$longitude)+c(-0.5, 0.5), ylim=range(volc2$latitude)+c(-0.5, 0.5))
hrw=0.5#define scalar with half the width of the rectangles
#create vector with values for non-greyness:
ramp=colorRamp(colors=c("blue", "yellow", "red"))
col.val=rgb(red=ramp(alt.s), maxColorValue=255)
#draw rectangles:
rect(xleft=volc2$longitude-hrw, xright=volc2$longitude+hrw,
     ybottom=volc2$latitude-hrw, ytop=volc2$latitude+hrw,
     border=col.val, col=col.val)
```

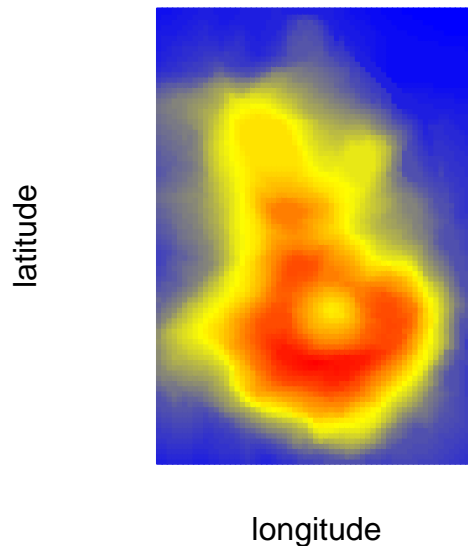


Figure 39: Illustration of the use of the function `ramp` to display a digital elevation model using a customized color gradient (here blue-to-yellow-to-red).

Note that the function `colorRamp` could also have been used to generate Fig. 37 and Fig. 38 for which the vector to be handed over to the argument `colors` of the function `colorRamp` would simply be `c("white", "blue")` or `c("white", "black")`, respectively.

6.6 Transparent Colors (Arguments `alpha.f` or `alpha`)

Another nice and very useful feature is the option to let colors be transparent. With Fig. 10 and 12 you have already seen two examples with transparent grey. An example for generating transparent colors other than grey is the following:

```
par(mar=rep(0.1, 4))#set margin widths for figure
plot(x=c(1, 1.5, 2), y=c(1, 1+sqrt(1-0.25^2), 1), axes=F, xlab="", ylab="", asp=1,
     yaxs="i", xaxs="i", xlim=c(0.5, 2.5), ylim=c(0.5, 1+sqrt(1-0.25^2)+0.5), pch=19, cex=25,
     col=adjustcolor(col=c("red", "green", "blue"), alpha.f=0.5))
```

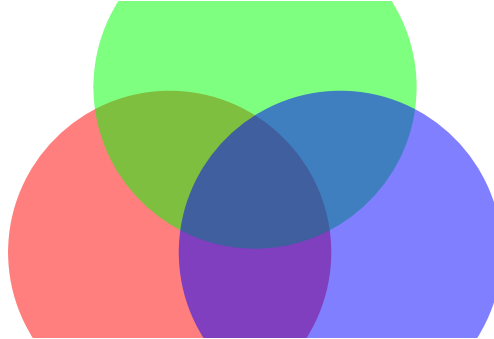


Figure 40: Illustration of the effect of semi-transparent colors generated using the argument `alpha.f` of the function `adjustcolor`.

The key argument used to generate Fig. 40 is `alpha.f` (in functions other than `adjustcolor` it is usually named `alpha`; see below). This parameter/argument sets the *opacity* of colors, whereby 0 means minimal opacity (in which case the color become invisible) and 1 means maximal opacity (i.e., the color is not transparent at all). The effect of intermediate values of `alpha.f` is that when a point (or rectangle, or polygon) is plotted on top of one already existing, it is not entirely hiding the color below but lets it 'shine through' to some extent. Note that the function `rgb` supports transparent colors (argument `alpha`)⁸.

I find this feature particularly useful in case of scatter plots because it represents a simple means to illustrate the density of points. An example of this effect can be seen in Fig. 10 and Fig. 12. For this I used the function `grey` (which supports transparent colors since R 3.0; I think) which leads to concentrations of points showing up with darker shading. Just as a quick repeat: for creating Fig. 12 the function `plot` was called with the argument `col=grey(level=0.5, alpha=0.5)`. Figure 41 illustrates the combined effects of the arguments `alpha` and `level` of the function `grey`.

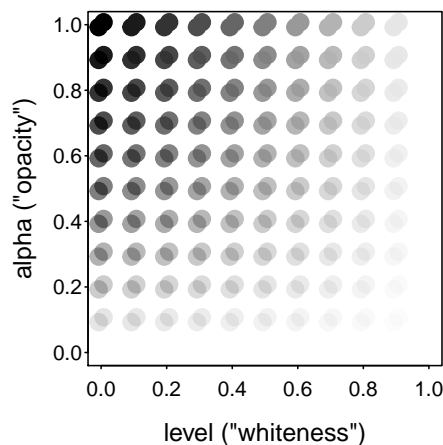


Figure 41: Illustration of the effect of semi-transparent colors generated using the argument `alpha` (in combination with the argument `level` of the function `grey`). Note that different combinations of `alpha` and `level` lead to the same color but that smaller values of `alpha` in combination with smaller values of `level` lead to a better recognizability of the overlapping area.

⁸Btw, the function `adjustcolor` allows to turn color values from various formats into the respective hexadecimal number. There is another function which allows turn color values from various formats into red-green-blue colors, the name of which being `col2rgb` (I just mention this here for completeness).

7 Customized Legends

As said above, certain kinds of legends cannot really be created using the function `legend` (e.g., a legend for such color coding as shown in Figures 37, 38, and 39). For such figures I'd add a legend manually, and the simplest way I'm aware of to achieve that is to just add many short and parallel lines depicting the gradient with different colors (what else, one might ask). I'll use the example of Figure 39 to illustrate how this can be done.

Placing a legend into this plot means to think about a couple of steps, and I'd begin with determining where it should be located. A natural location seems to be at the right of the plot itself where there is quite some space (which arose since the plot was created with an aspect ratio of 1 (`asp=1`), but the extend of the actual map is not the same in east-west and in north-south direction). Hence, to figure out where the legend could be located, I'd create the plot with the axes displayed, i.e.,

```
plot(x=volc2$longitude, y=volc2$latitude, axes=T, xlab="longitude", ylab="latitude", asp=1,
     xlim=range(volc2$longitude)+c(-0.5, 0.5), ylim=range(volc2$latitude)+c(-0.5, 0.5))
```

which suggests to me to let the left edge of the legend be at 70 and its right edge at 75 (which are both clearly outside the range of longitude). With regard to the top-to-bottom extend, I'd choose 60 to 80. In between these y-values I'd just add a large number of lines (e.g., 200) with the color varying along the gradient as displayed in the plot. In the following code there is one argument (`lend`) handed over to the function `segments` that deserves attention. This argument sets the 'line end style' where a value of 2 means a square end (the default of 0 means a round end). The final line of the script adds a scale besides the legend; note that indicating `pos=2` leads to the text being displayed to the left of the position indicated by `x` and `y`. This is the entire code needed to create Fig. 42 (of which only the last four instructions are newly added to the code that was already used to create Fig. 39):

```
par(mar=c(1.5, 1.5, 0.5, 0.5))#set margin widths for figure
par(mgp=c(0.2, 0, 0))#and where axis labels are displayed (first value)
plot(x=volc2$longitude, y=volc2$latitude, axes=F, xlab="longitude", ylab="latitude", type="n",
     asp=1, xlim=range(volc2$longitude)+c(-0.5, 0.5), ylim=range(volc2$latitude)+c(-0.5, 0.5))
hrw=0.5#define scalar with half the width of the rectangles
#create vector with values for non-greyness:
ramp=colorRamp(colors=c("blue", "yellow", "red"))
col.val=rgb(red=ramp(alt.s), maxColorValue=255)
#add rectangles:
rect(xleft=volc2$longitude-hrw, xright=volc2$longitude+hrw, ybottom=volc2$latitude-hrw,
     ytop=volc2$latitude+hrw, border=col.val, col=col.val)
resolution=200#define number of steps from smallest to largest value of altitude
#create vector with y-values at which the legend should appear:
yvals=seq(from=60, to=80, length.out=resolution)
#add legend using the function segments:
segments(x0=70, x1=75, y0=yvals, y1=yvals, lend=2,
         col=rgb(ramp(seq(from=0, to=1, length.out=resolution)), maxColorValue=255))
text(x=72, y=c(60, 80), labels=range(volc2$altitude), pos=2, cex=0.7)
```

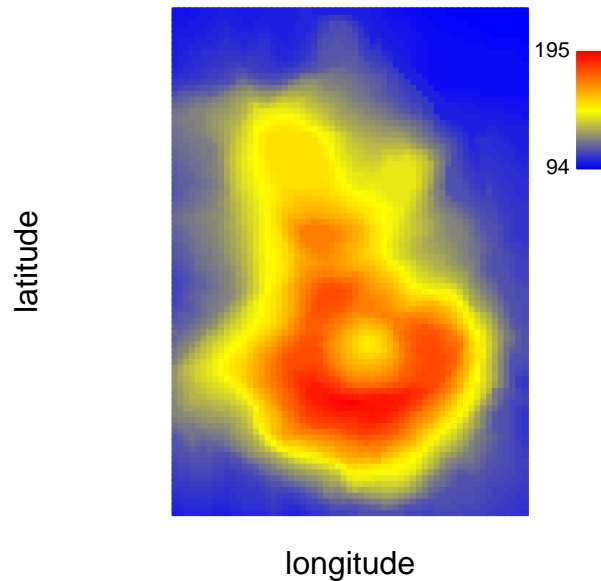


Figure 42: Illustration of the use of the function `ramp` to display a digital elevation model using a customized gradient with a customized legend added.

I want to emphasize here that it hardly ever happens that I just write down such a piece of code, and then it works. Rather, I usually build it up, step-by-step, beginning with the essentials and ending with 'cosmetics', making errors and modifications throughout. When the code run is modified and saved in a script this is actually easy and, from my point of view, the best way of creating such a figure.

8 Fonts

When one doesn't bother about, plots come with some standard font which looks good to me, and changes to it seem only very rarely be needed. Still, I remember cases where people had to change fonts to match journal requirements. Here are a few hints how this can be done. The key to changing the font with which text appears is the argument `family` which can be handed over to most (if not all) functions adding texts to plot (e.g., `plot`, `text`, `mtext`, or `axis`) and also to the function `par`. Changing the font can be a little tricky since the options available might depend on the operating system and also depend on the plotting device used. For instance, using a standard plotting window or the function `pdf` seems to make a difference. Figure 43 shows the fonts easily available when using the function `pdf` (I found them using `?postscript`). These are not the same fonts available when using the standard plotting device used when just calling the function `plot`. For instance, on my laptop, which runs under Ubuntu Linux, `names(X11Fonts())` reveals me the fonts easily available when using the function `plot`.

I am sure that there are plenty of other options. For instance, there is a function named `embedFonts` which seemingly can be used to embed fonts into pdf files (I haven't tested it yet; just mention it as it suggests this to be another ocean of options to be explored). After all, so far I rarely needed to change fonts, and that's why this section is short (I simply don't know much about what is possible here).

```

family=default
family="Times"
family="Palatino"
family="NewCenturySchoolbook"
family="Helvetica-Narrow"
family="Helvetica"
family="Courier"
family="Bookman"
family="AvantGarde"

```

Figure 43: Illustration of the use of the argument `family` (in combination with, e.g., the function `pdf`) to determine the font with which text appears. Note that the (uppermost) default font is the one appearing when not specifying the `family` argument.

9 3D-plots (Function `persp`)

Three-D plots are very useful to display how the interaction between two covariates (i.e., quantitative predictors) affects a response (actually, I believe them to be pretty much the only sensible way of depicting such an interaction; admittedly, though, many researchers initially have a hard time understanding them). A 3D-plot has three axes, and in the context I consider here the x- and a y-axis do represent the two predictors and the z-axis represents the response. Generally, I depict the fitted model as a surface showing the predicted response along both predictors, but also the values of the response itself (see Fig. 46 for an example). There are several functions and packages available for creating such plots, but I personally find the function `persp` most useful (but I have to confess that it is a couple of years ago that I explored functions other than `persp`). In the following I'll illustrate how to create a 3D plot depicting a model and the respective data using `persp`; btw, note that this is the first time in this text that I use a function other than `plot` to create a plot!). The main reason for showing this is completeness (I feel that such a text would be very incomplete without showing 3D plots). However, as you will see such 3D plots need quite a bit of code; and that's why for standard models I have programmed functions which take care of most of the business (these might be treated elsewhere at some point).

As usual, I begin with creating some data. These are two predictors and a response, whereby the effects of the two predictors on the response are interacting. The response is binary, so we fit a logistic model:

```

set.seed(1)
n=500#set sample size
pred1=runif(n=n, min=0, max=1)#create predictor 1
pred2=runif(n=n, min=0, max=1)#create predictor 2
#create linear predictor (basically the model wrt the effects of the predictors on the response):
lp=as.vector(scale(pred1))+as.vector(scale(pred1))*(0.75+as.vector(scale(pred2)))
resp=rbinom(n=n, size=1, prob=exp(lp)/(1+exp(lp)))#create response
#put all variables involved (including the two predictors z-transformed) into a data frame:
xdata=data.frame(pred1, pred2, resp, zpred1=as.vector(scale(pred1)), zpred2=as.vector(scale(pred2)))
rm(n, pred1, pred2, resp, lp)#... and remove them from the workspace
model.res=glm(resp~zpred1*zpred2, family=binomial, data=xdata)#fit the model
round(summary(model.res)$coefficients, 3)#and display its results

```

	Estimate	Std. Error	z value	Pr(> z)
(Intercept)	-0.026	0.131	-0.197	0.843
zpred1	2.263	0.216	10.457	0.000
zpred2	-0.122	0.124	-0.982	0.326
zpred1:zpred2	1.201	0.198	6.070	0.000

As can be seen from the model output, the interaction between the two predictors is clearly significant. The presumably best way of understanding and presenting such an interaction seems to be a plot, and that's what

we're going to create now.

To get the plot I'd begin with getting the surface of the fitted model for various combinations of values of the two predictors. So the first step could be to define the range of values to be considered for the two predictors. Since the model was fitted with both of them being z-transformed (to a mean of 0 and a standard deviation of 1) these should be in z-space, too. One also needs to consider how many different values to consider per predictor. From my experience 11 values usually works quite well (more on that below). To get two vectors spanning from the minimum to the maximum of a given predictor (with equal increments) I'd use the function `seq`, namely:

```
xp1=seq(from=min(xdata$zpred1), to=max(xdata$zpred1), length.out=11)
yp2=seq(from=min(xdata$zpred2), to=max(xdata$zpred2), length.out=11)
```

Note the names I gave to these two vectors which code the predictor (e.g., `p1`) but also have `x` or `y` as a prefix. This helps to avoid a mess later when it comes to assigning each of two predictors to the x- or y-axis of the plot. Next, we need to get the predicted values (based on the model) corresponding to each of the combinations of values of the two vectors. Since the format of the fitted values required for the function `persp` is a matrix, the fitted values could be obtained using the function `outer`. This is how it goes:

```
coeffs=coefficients(model.res)#store model coefficients in a vector with a short name
#call function outer; note that the function is defined within the call or outer
pred.mat=outer(X=xp1, Y=yp2, FUN=function(x, y){
  coeffs["(Intercept)"]+coeffs["zpred1"]*x+coeffs["zpred2"]*y+coeffs["zpred1:zpred2"]*x*y
})
#pred.mat comprises predicted values in link space for each combination of values of xp1 and yp2
#convert them to response space (i.e., probabilities):
pred.mat=exp(pred.mat)/(1+exp(pred.mat))
```

Note that `outer` reveals a matrix with the rows corresponding to what it got handed over to the argument `X` (here `xp1`) and the columns corresponding to what it got handed over to the argument `Y` (here `yp2`). Now we are already prepared to begin with the 3D plot. The function `persp` has quite a few arguments, some of which I explain in the following: The arguments `x` and `y` define the x- and y-axis, respectively. They expect vectors (with values in increasing order), and I'll hand over to them the vectors representing the range of the two predictor variables as defined above (`xp1` and `xp2`). Their lengths must correspond to the number of rows and columns, respectively, in the matrix which will be handed over to the argument `z` which represents the plane or surface to be drawn. Here I let this be the fitted model surface as presented in `pred.mat`. Two arguments deal with the viewing angle of the 3D plot, namely `theta` which defines the horizontal viewing angle and `phi` which defines the vertical viewing angle (both interpret the values handed over as angles indicated in degrees). For `phi` a value of 10 most usually works well, with `theta` one will have to play around until one has a good view on the surface of the plane. There are two further arguments which relate to the overall appearance of the 3D plot which I use a lot. These are `expand` which defines the vertical stretching of the plot whereby I frequently indicate values smaller than the default of 1 (e.g., 0.6). The other is `r` which defines the distance of the viewpoint from the plot. Leaving it at its default (`sqrt(3)`) leads to quite a strong 'deformation' of the 3D plot, so I usually use much larger values such as 10 (probably handing over a small value like 0.2 to the argument `d` would do the same job, but I never tried it). With these last four arguments one basically has to play around (i.e., trying different values) until one likes the overall appearance of the plot, whereby `theta` will be the most important one. There are quite a few further arguments (e.g., `xlim`, `ylim`, `zlim`, `xlab`, `ylab`, and `zlab`) which have self-explaining names and others dealing with more options to display the surface (e.g., `ltheta` and `lphi`) which define the location of a 'light source' illuminating the surface which can be used to get really fancy looking plots (which, though, are probably usually not very useful in the context of getting figures that make a good job in a scientific paper). One final argument I want to mention here, namely `ticktype`; this accepts two entries: `"simple"` (which is the default), in which case no axis ticks and corresponding labels are displayed but just arrows indicating in which direction the predictors increase and `"detailed"`, in which case axis ticks and labels are indicated as usual. Having said all this, it's time to produce a first draft of the 3D plot (note that the names of the vectors `xp1` and `yp2` make it easier to not get lost wrt what is what):

```
#set margins
par(mar=c(0.2, 1.5, 0.2, 0.2))
tmat=persp(x=xp1, y=yp2, z=pred.mat, theta=-70, phi=10, expand=0.6, r=10, xlab="predictor1",
  ylab="predictor2", zlab="response", zlim=c(0, 1))
```

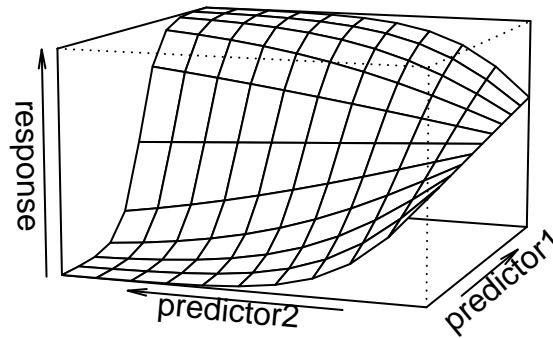


Figure 44: 3D plot of a model surface.

Note that `persp` is not only used to provide a plot but that its 'result' (whatever that is) is stored in an object named `tmat`. The reason for this is explained below.

What is missing in Fig. 44 are the data! I greatly prefer to show them since only these tell the reader of a paper how good the model actually is... Showing just all 500 values of the response, though, doesn't really seem to be an option, since (i) first these are only ones and zeroes, and (ii) these are pretty many, so probably one wouldn't really see that much except a whole lot of points which would make the model surface pretty invisible... What I'd do instead is to determine the probability to observe a one, separately for each of the cells of the model surface shown in Fig. 44 and plot these *probabilities*. This requires to *bin* the two predictors (i.e., turning a potentially continuously varying variable into a set of discrete classes or bins). This can be done using the function `cut`. This function has a few arguments needed for our purpose: `x` takes the values to be binned, `breaks` gets handed over the break points between adjacent bins, `labels` determines what the function reveals (if set to `F` it returns the index of the bin a value in `x` falls into), and `include.lowest` finally determines what happens with values being right at lower the edge of the smallest bin (if set to `T` these will be included into it, otherwise the function will reveal an `NA` for them). Here I want to bin the two z-transformed predictors (`zpred1` and `zpred2`), whereby the bin widths are defined by the fact that we decided to show a model surface consisting of 10 cells along both predictors, and this is how it goes:

```
#bin predictor 1:
bin.width=diff(xp1)[1]#determine bin width for the predictor
binned.pv1=cut(x=xdata$zpred1, breaks=xp1, labels=F, include.lowest=T)
binned.pv1=min(xp1)+bin.width/2+(binned.pv1-1)*bin.width
#bin predictor 2:
bin.width=diff(yp2)[1]#determine bin width for the predictor
binned.pv2=cut(x=xdata$zpred2, breaks=yp2, labels=F, include.lowest=T)
binned.pv2=min(yp2)+bin.width/2+(binned.pv2-1)*bin.width
```

The first line of the above code determines the bin width. Here this can be extracted from the object `xp1` which we created above as a sequence of eleven equidistant values spanning the range of `xdata$zpred1` from its minimum to its maximum. The function `diff` determines the difference between the second and the first value in `xp1`, that between its third and second value, and so on, and from the resulting vector we simply take the very first value. In the second line we use the function `cut` to bin `xdata$zpred1`. The breakpoints can be simply be taken from the object `xp1`, and the arguments `labels` and `include.lowest` are set to `T` and `F`, respectively,

for reasons explained above. Finally, we turn the index of the bin into the value being at the middle of the respective bin. The logic behind this line is as follows: the bins are consecutively numbered from smallest to largest as determined by the vector handed over to the argument `breaks` of the function `cut` (given `xp1` this means that the bins are numbered from 1 to 11). Hence, to determine the bins' mid points we need to add half the bin width to the minimum of respective predictor and then add the bin width times the bin index minus 1. After having done this I'd have a look at `plot(xdata$zpred1, binned.pv1)` and `plot(xdata$zpred2, binned.pv2)`.

Now we can finally determine the probabilities to observe a 1 in each of the cells of the surface of the plot. This can be achieved using

```
#average the response separately per combination of the values of binned.pv1 and binned.pv2:
observed=aggregate(x=xdata$resp, by=list(pv1=binned.pv1, pv2=binned.pv2), FUN=mean)
#determine sample size per combination of values of the two predictors (not counting NAs):
N=aggregate(x=!is.na(xdata$resp), by=list(pv1=binned.pv1, pv2=binned.pv2), FUN=sum)
```

The object `observed` resulting from the call of `aggregate` is a data frame with columns named `pv1`, `pv2` and `x`. Note that for determining the sample size we use `!is.na(xdata$resp)` (handed over to the argument `x`) which turns `xdata$resp` into a logical vector (comprising TRUEs and FALSEs) which then can simply be summed (in R a TRUE counts as a 1 and a FALSE as a 0).

Now we are almost prepared to add the data to the plot. For this I am going to use the function `points`. However, the function `points` accepts only an x- and a y-value, but in our data we actually deal with points having three coordinates (because it is a 3D plot). This is where the object created by `persp` (`tmat` in the code above) comes into play. In fact, whenever we create a '3D' plot and display it on a computer monitor or add it to a pdf-file it is actually presented in 2D, so a tranformation from 3D ot 2D takes place. The parameters of this tranformation is what the function `persp` reveals (in addition to the plot) and what the object `tmat` comprises. Here I use it to plot the observed mean response per cell of the surface (i.e., the probability to observe a 1):

```
#set margins
par(mar=c(0.2, 1.5, 0.2, 0.2))
tmat=persp(x=xp1, y=yp2, z=pred.mat, theta=-70, phi=10, expand=0.6, r=10, xlab="predictor1",
  ylab="predictor2", zlab="response", zlim=c(0, 1))
points(trans3d(x=observed$pv1, y=observed$pv2, z=observed$x, pmat=tmat))
```

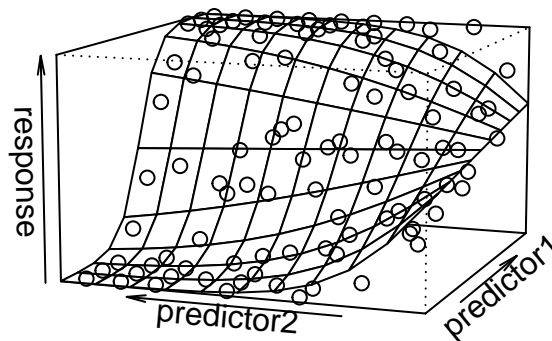


Figure 45: 3D plot of a model surface with the data added (but not in a very accessible form).

The function `trans3d` in the above code chunk might need some explanations. It transforms coordinates

from 3D to 2D, such that graphical elements can be added to an existing plot. Its arguments take the x-, y-, and z-coordinates of the elements to be added and the object returned from a call of `persp` which indicates the transformation parameters. It returns the x- and y-coordinates of the elements to be added. These can then be handed over to functions such as `points`, `lines`, `segments`, etc..

However, not too much can be seen in Fig. 45 which is due to the fact that it is hard to locate the points in the 3D plot. Two means I can think of to improve the appearance of the figure: (i) color the points according to whether they are above or below the fitted surface and (ii) connect them to the surface by lines. Another issue with Fig. 45 is that so far, the points all have the same size, but I'd want them to depict the number of observations per cell of the surface. We already had determined the number of observations per cell, and we could certainly use the argument `cex` to scale the size of the points. However, this is a 3D plot, and people will probably look at the *volume* of the points to judge their size. Hence, I'd let the values handed over to `cex` be proportionate to the 3rd root of the number observations per cell. The following piece of code achieves all that:

```
#get predicted values corresponding to the observed values (from the model) in link space:
expected=coeffs["(Intercept)"]+coeffs["zpred1"]*observed$pv1+
  coeffs["zpred2"]*observed$pv2+coeffs["zpred1:zpred2"]*observed$pv1*observed$pv2
expected=exp(expected)/(1+exp(expected))#transform from link to probability space
par(mar=c(0.2, 1.5, 0.2, 0.2))
#make plot
tmat=persp(x=xp1, y=yp2, z=pred.mat, theta=-70, phi=10, expand=0.7, r=10, xlab="predictor1",
  ylab="predictor2", zlab="response", zlim=c(0, 1))
#add points;
points(trans3d(x=observed$pv1, y=observed$pv2, z=observed$x, pmat=tmat),
  pch=c(1, 19)[1+as.numeric(observed$x>expected)], cex=0.5*N$x^(1/3))
#add lines:
for(i in 1:nrow(observed)){#for each row in observed
  #(can happen that there are no observations in a given combination of the two predictors)
  lines(#add line (see text for more
    trans3d(x=observed$pv1[i], y=observed$pv2[i], z=c(observed$x[i], expected[i]), pmat=tmat),
    lty=c(2, 1)[1+as.numeric(observed$x[i]>expected[i])])
}
```

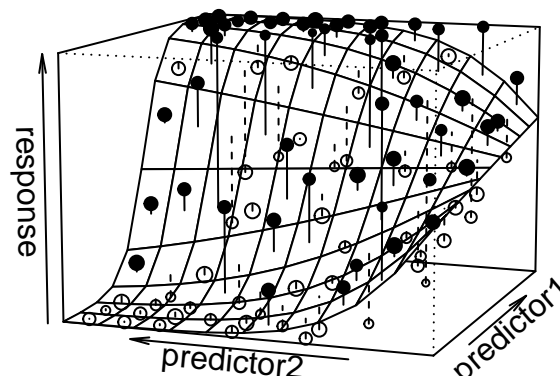


Figure 46: 3D plot of a model surface with the data added and such that it is perceivable how observations link to the fitted surface. The points are filled when the observed value is larger than the one expected given the model (and otherwise open), and the lines connecting observations with the fitted value are solid when the observed value is larger than the respective expected value (and otherwise dashed). The *volume* of the points corresponds to the number of observations falling in the respective cell of the depicted surface.

The way how the points and lines are added to the plot (Fig. 46) deserves some comments. As mentioned above, creating such a plot means to transform 3-dimensional coordinates defining the location of objects (e.g., points or lines) along three axes into the two dimensions available on a sheet of paper. This transformation has parameters (determined by parameters of the function `persp` such as `theta`, `phi`, and `r`), and these are saved in the object returned by `persp`, here named `tmat`. In order to add points and lines to the figure I use standard

functions (i.e., `points` and `lines`). However, since these can handle only two-dimensional input (i.e., x- and y-coordinates) the 3D locations of the objects need to be transformed into 2D which is done by the function `trans3d`. This function, in turn, has arguments for the locations of the object to be transformed (`x`, `y`, and `z`) whereby the values handed over to these arguments can be vectors (which is here needed for the z-coordinates of the lines which are different for their starts and ends). Furthermore, `trans3d` has an argument `pmat` which needs to get handed over the parameters of the transformation (here the object revealed by the function `persp`). The result of the call of the function `trans3d` can then be handed over to functions like `points` or `lines` which are then used as usual (i.e., with additional arguments handed over to determine the appearance of the points or lines, respectively).

Admittedly, it is quite a chunk of code needed to get Fig. 46. However, anyone thinking that this is complex should ask her-/himself how s/he would get such a plot in any other graphical software. Also, when the data or the model change, one simply needs to run the code again and would get the updated version of the plot with ease. Finally, if one needs such a plot more frequently, one could program a function taking over all the business (which I actually did; ask me in case you need it).

10 Complex Plots (getting "high-level" Plots using low-level Functions)

By now we already have assembled quite a set of tools to create quite a variety of plots showing data and/or models and/or whatever. What follows in this section are examples of how these tools can be combined in various ways to create pretty much whatever plot you want. In fact, as initially stated, R provides many 'high-level' plotting functions creating all sorts of plots (e.g., barplots, boxplots, bubbleplots) with quite some ease. However, I don't really use them at all because frequently I have to get further elements (e.g., model lines, axes at the original scales, etc.) into such plots, and then things tend to get tricky. In this section I want to show a few examples of how somewhat complex ('high level') plots can be created by combining low-level plotting functions.

10.1 A Bubbleplot indicating Sample Size

Frequently I use bubbleplots (or some derivatives of them) to indicate sample sizes (i.e., the number of identical observations), and frequently I combine them with other graphical elements. For instance, when having tied observations, I like to make this visible by connecting observations made on the same individual with a line. This section shows (two examples for) how one could create such a plot.

As usual I begin with generating some data:

```
set.seed(1)
obs1=sample(x=1:8, size=20, replace=T)
obs2=sample(x=1:8, size=20, replace=T)
```

`obs1` and `obs2` are assumed to represent two observations made on the same individuals whereby the linking is represented by the position in the vectors (i.e., entries at corresponding positions belong to the same individual). What I want to achieve is a plot which depicts the values of `obs1` besides those of `obs2` (say at values of `x` being 1 and 2, respectively), whereby observations of the same individual should be linked by a dashed line. Furthermore, there will be tied observations (i.e., some identical values in `obs1` and also in `obs2`), and these should be made visible by the *area* of the dots. The reason why I'd depict sample size by the area of the points is that most people judge the relative size of points by comparing their area rather than their diameter. This is how one could get this (note that I also add the results of a paired t-test on top of the graph and a legend; see Fig. 47 for the result):


```

par(mar=c(1.5, 3, 1, 0.5), mgp=c(1.8, 0.5, 0))#set margins etc.
#create empty plot:
plot(x=1, y=1, xlim=c(0.5, 3), ylim=c(1, 8.4), xaxs="i", type="n", las=1, xaxt="n", tcl=-0.25,
     ylab="response", xlab="", bty="l")
#add lines connecting observations made for the same individual:
segments(x0=rep(x=1, n=length(obs1)), x1=rep(x=2, n=length(obs1)), y0=obs1, y1=obs2, lty=2)
#add points; step 1, get frequency table for obs1:
xx=table(obs1)
#step 2, add points:
points(x=rep(x=1, times=length(xx)), y=as.numeric(names(xx)), cex=0.8*sqrt(xx), pch=19, col="grey")
xx=table(obs2)#and the same for obs2:
points(x=rep(x=2, times=length(xx)), y=as.numeric(names(xx)), cex=0.8*sqrt(xx), pch=19, col="grey")
mtext(text=c("obs1", "obs2"), side=1, line=0.2, at=c(1, 2))#add labels at x-axis
xx=t.test(obs1, obs2, paired=T)#add result of t-test
mtext(side=3, line=-0.2, cex=0.75, at=1.5,
      text=bquote(list(t[.(xx$parameter)]==.(round(xx$statistic, 2)), P==.(round(xx$p.value, 3)))))
segments(x0=1, x1=2, y0=8.4, y1=8.4)
#add legend:
legend(x="topright", legend=1:5, pch=19, pt.cex=0.8*sqrt(1:5), cex=0.75, bty="n", col="grey")

```

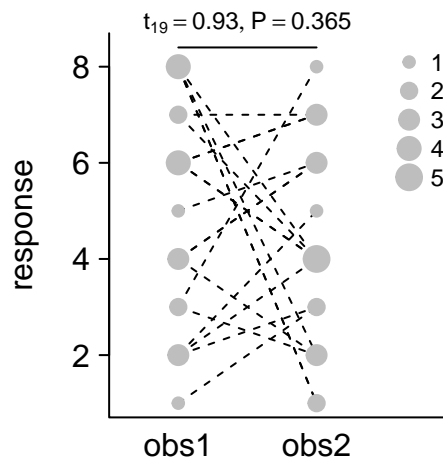


Figure 47: Illustration of a bubble plot used to indicate tied observations.

Note that certain parts of the above code arose 'empirically', i.e., by just trying different values, until I liked how it looked like. This, for instance, is the case for the limits of the y-axis as indicated in the call of the function `plot` or the value of 0.8 by which the size of the dots is multiplied (functions `points` and `legend`). The reason for displaying the points in grey rather than black is simply because I frequently feel larger black dots to look somewhat 'brutal' whereas grey ones look more 'modest' to me... This also requires to add the lines connecting observations of the same individual (call of `segments`) *before* adding the points (call of `points`) because otherwise the black lines would appear on top of the grey dots which definitely wouldn't look nice... The indication of the y-values at which the points are finally placed (`y=as.numeric(names(xx))` in the call of the function `points`) is required to be like that because they are actually the names of the object `xx` which are character-values that need to be turned into numbers before they can be used to indicate the elevation of the points. Also note `bty="l"` in the call of `plot` which leads to the box around the plotting region being 'L'-shaped. Finally, note that the function `legend` needs one argument for the size of the text (`cex`) and one for the size of the points (`pt.cex`), and that `bty="n"` leads to no box appearing around the legend. Btw, it is the point *area* that represents the number of observations (due to `cex=sqrt(<...>)` in the calls of the functions `points` and `legend`), and the caption should be explicit about this point in case there is no legend (i.e., don't refer to the 'size' of the dots which could mean 'area' or 'diameter').

Some time ago I came across another neat way to indicate sample size in bubble plots such that a legend is not needed; that is, displaying sample size by a number of concentric circles (Fig. 48). This is how that goes:

```

par(mar=c(1.5, 3, 1, 0.5), mgp=c(1.8, 0.5, 0))#set margins etc.
#create empty plot:
plot(x=1, y=1, xlim=c(0.5, 3), ylim=c(1, 8.4), xaxs="i", type="n", las=1, xaxt="n", tcl=-0.25,
     ylab="response", xlab="", bty="l")
#add lines connecting observations made for the same individual:
segments(x0=rep(x=1, n=length(obs1)), x1=rep(x=2, n=length(obs1)), y0=obs1, y1=obs2, lty=2)
#add points, step 1, get frequency table for obs1:
xx=table(obs1)
#add white points to cover the lines below the points to be shown later:
points(x=rep(x=1, times=length(xx)), y=as.numeric(names(xx)), cex=0.8*xx, pch=19, col="white")
#step 2, add points:
for(i in 1:length(xx)){#loop through each entry in xx
  for(j in 1:xx[i]){#loop from 1 to the value in xx[i]
    #and add one point with the resp. diameter at the resp. location (x and y):
    points(x=1, y=as.numeric(names(xx)[i]), cex=0.8*j)
  }
}
xx=table(obs2)#and the same for obs2:
points(x=rep(x=2, times=length(xx)), y=as.numeric(names(xx)), cex=0.8*xx, pch=19, col="white")
for(i in 1:length(xx)){
  for(j in 1:xx[i]){
    points(x=2, y=as.numeric(names(xx)[i]), cex=0.8*j)
  }
}
#add labels at x-axis:
mtext(text=c("obs1", "obs2"), side=1, line=0.2, at=c(1, 2))

```

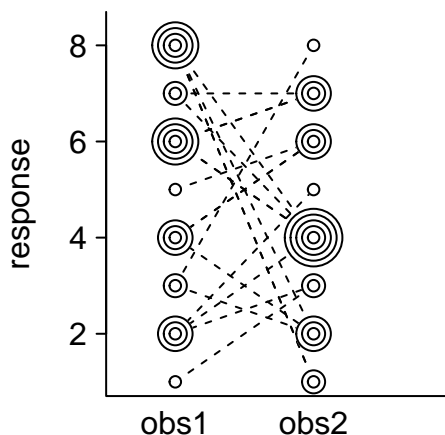


Figure 48: Illustration of another style of bubble plot used to indicate tied observations. Here sample size per combination of obs and response is indicated by the number of concentric circles shown.

Some comments on the code used to create Fig. 48 might be helpful: First of all note that I first add white points to cover the lines connecting observations of the same individual below the points to be later added. After that two nested loops are used to add the points; the outer loop runs over each entry in the table (vector `xx`) and the inner over the integer numbers from 1 to the i^{th} value in `xx[i]`.

10.2 A Boxplot for a Two-Factor Design

Another situation I encounter quite frequently is a data set comprising two categorical predictors, and the response should be shown separately for each of their combinations. This can be achieved quite easily by creating a new factor combining the two factors and creating the plot based on that. What follows shows how

one could do that. As usual, I begin with creating some data:

```
set.seed(1)
age=as.factor(sample(x=c(3, 5), size=100, replace=T))
when=as.factor(sample(x=c("pre", "post"), size=100, replace=T))
response=rnorm(n=100, mean=c(4, 6)[1+as.numeric(age=="5" & when=="post")])
```

For combining the two factors in a single factor, the function `paste` is an option:

```
both.facs=as.factor(paste(age, when, sep="_"))
levels(both.facs)#inspect levels

[1] "3_post" "3_pre" "5_post" "5_pre"
```

However, I wouldn't be happy with the levels as they are indicated. The reason is that I'd like to create the figure with the function `plot` getting `as.numeric(both.facs)` handed over to the argument `x`. This would lead to the data to appear with first post and then pre within age groups (remember `as.numeric(both.facs)` will number the levels according to the order by which they are returned by the function `levels`). An easy solution is to redefine `both.facs` using the function `factor`. This function allows to define the order of the levels of a factor by handing over a vector with the levels in the desired sequence to its argument `levels`. Since the order of the first two and also that of the last two levels should be reversed one could use:

```
table(both.facs)#inspect original frequency distribution

both.facs
3_post 3_pre 5_post 5_pre
    29    23     25     23

#redefine levels of both.facs:
both.facs=factor(x=both.facs, levels=levels(both.facs)[c(2, 1, 4, 3)])
table(both.facs)#inspect new frequency distribution

both.facs
3_pre 3_post 5_pre 5_post
    23    29     23     25
```

As one can see from the output above, the assignment of the cases to the levels does not change, but only the order in which they are indicated (note that `table(<factor>)` reveals the levels in the same order as `levels(<factor>)`). Now everything is set to create the plot (Fig. 49). Note in the following code that when calling `plot`, x-axis ticks are suppressed (`xaxt="n"`), and that x-axis labels are later added using the function `mtext`. Letting the symbols being filled circles (`pch=19`) and the color being transparent grey (`col=grey(level=0.5, alpha=0.5)`) leads to concentrations of data being visible as darker shading of the points. Btw, in the first call of the function `mtext` the vector handed over to the argument `text` needs to comprise the levels of the factor `when` in reversed order (because of the redefinition of `both.facs`) which is achieved by use of the function `rev`. Finally, I add medians and quartiles. Here's the code needed:

```
par(mar=c(2.5, 3, 1, 0.5), mgp=c(1.8, 0.5, 0))#set margins etc.
#create plot:
plot(x=as.numeric(both.facs), y=response, xlim=c(0.5, 4.5), las=1, ylab="response", xlab="",
     xaxt="n", tcl=-0.25, pch=19, col=grey(level=0.5, alpha=0.5))
mtext(text=rev(levels(when)), side=1, line=0.2, at=1:4)#add text for age
mtext(text=levels(age), side=1, line=1.4, at=c(1.5, 3.5))#... and when
medians=tapply(X=response, INDEX=both.facs, FUN=median)#calculate medians
q1=tapply(X=response, INDEX=both.facs, FUN=quantile, prob=0.25)#...and quartiles
q3=tapply(X=response, INDEX=both.facs, FUN=quantile, prob=0.75)#still...
hbw=0.25#define 'half-box-width'
rect(xleft=(1:4)-hbw, xright=(1:4)+hbw, ybottom=q1, ytop=q3)#add quartiles
segments(x0=(1:4)-hbw, x1=(1:4)+hbw, y0=medians, y1=medians, lwd=2)#add medians
```

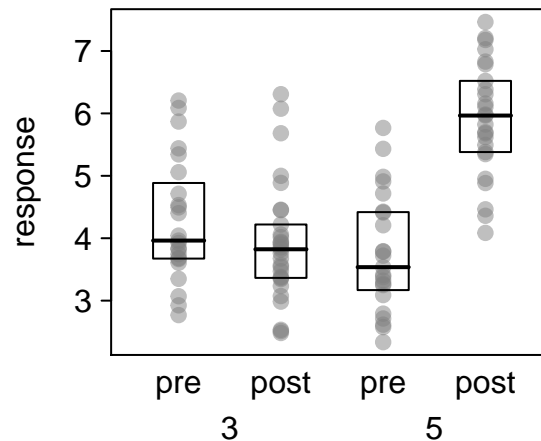


Figure 49: Illustration of how the effects of two categorical predictors on a response could be illustrated and how a display of data can be combined with summary statistics.

10.3 Combining Bubbleplots, Boxplots and Models

This section shows an example of how a more complex data design and the respective model could be depicted. The design I chose as an example comprises two factors (i.e., categorical predictors; here both with two levels) and one covariate (i.e., a quantitative predictor: here chosen to be an integer) whereby they are assumed to interact in their influence on the response. The response is a count, implying that there will be many tied observations (here meant in the sense of data points being identical with regard to all three predictors and the response). The figure I chose to create should show the impact of the covariate on the response, separately for each combination of levels of the two factors (which means that four plots have to be arranged in a two-by-two panel). The response and the model depicting the impact of the covariate on the response should be shown, separately for each of the four combinations of levels of the two factors, and tied observations should be indicated by the size of the dots. Finally, I'd like to add boxes indicating medians and quartiles of the response to enable easier reading of the message (Fig. 51). Hence, the desired plot needs to combine different graphical elements like bubbleplots, boxplots, and lines indicating fitted models.

As always, it is first needed to generate some data. I cannot go into the details of how the data are generated here (which is beyond the scope of this little tutorial), but what follows is what needs to be done:

```
set.seed(2)
n=500#set sample size
#create factors species and sex:
species=as.factor(sample(x=letters[1:2], size=n, replace=T))
sex=as.factor(sample(x=c("F", "M"), size=n, replace=T))
covariate=sample(x=1:10, size=n, replace=T)#... and a covariate
#create the response:
response=rpois(n=n, lambda=exp((as.numeric(species)+as.numeric(sex))*0.3*as.vector(scale(covariate))))
z.cov=as.vector(scale(covariate))#z-transform the covariate
xdata=data.frame(sex, species, covariate, z.cov, response)#put all variables in data frame...
rm(sex, species, covariate, response)#...and remove them from the workspace
```

The next step is to fit a model. This needs to be a Generalized Linear Model (GLM; McCullagh & Nelder 2008) with Poisson error structure and log link function, and it should include all possible interactions between the two factors and the covariate. This can be fitted as follows:

```
model.res=glm(response~species*sex*z.cov, family=poisson, data=xdata)
```

Now I have the data and a model and want to plot all that⁹. The idea of the plot is to depict the response as a function of the covariate, but separately for each of the four combinations of species and sex because the effect

⁹Note that, when it comes to proper modeling one would need to worry about model stability, overdispersion, goodness of fit and potentially the significance of the full model and the interactions. Here I drop all this since this exercise is solely about plotting.

of the covariate on the response might differ between them. Hence, four plots are needed, and these should be arranged in a two-by-two design. The labels of the x- and y-axis should be displayed only once to the left and below this two-by-two arrangement. Furthermore, on top and to the left of the plots the levels of the two factors should be indicated. Such an arrangement can be created using the function `layout` as explained in section 4. Here is how that goes (a very similar piece of code was already explained in more detail above):

```
lmat=matrix(1:4, ncol=2, byrow=T)#begin with matrix for the data
#append two cells on top (for factor levels) and one cell below (for axis label):
lmat=rbind(c(7, 8), lmat, 5)
#append two cells and one cell to the left for factor levels and axis label, respectively
lmat=cbind(c(11, 9, 10, 12), c(11, 6, 6, 12), lmat)#(cells 11 and 12 serve to complete the matrix)
#set layout with appropriate cell widths and heights:
layout(mat=lmat, widths=c(1, 1, 6, 6), heights=c(1, 6, 6, 1))
layout.show(n=max(lmat))#and show it
```

11		7	8
	9	1	2
	6		
	10	3	4
12		5	

Figure 50: Layout to be used to plot a design comprising two factors with two levels each and a covariate (for details see text).

The plotting regions 1 to 4 (see Fig. 50) serve to depict the data and the model; cells number 5 and 6 serve to indicate labels for the x-and y-axis, respectively; cells 7 to 10 will indicate the respective factor levels; and cells 11 and 12, finally, are simply needed to complete the matrix (but they will remain empty). What follows is quite a bit of code which fills all these cells one after the other. I'll first show the code and further below comment on selected parts of it:

```
layout(mat=lmat, widths=c(1, 1, 6, 6), heights=c(1, 6, 6, 1))#use layout as already created
par(mar=c(1.8, 1.8, 0.2, 0.2))#set margins
par(mgp=c(1, 0.5, 0))#set where to put labels (key is second value)
hbw=0.25#define width of the boxes to be shown (for quartiles and medians)
#create vector of covariate for plotting the data:
plot.xvals=min(xdata$covariate):max(xdata$covariate)
#create vector of covariate for plotting the model:
xvals=(plot.xvals-mean(xdata$covariate))/sd(xdata$covariate)
sex.levels=levels(xdata$sex)#define vectors with levels of factor sex...
species.levels=levels(xdata$species)#... and species
for(i in 1:length(sex.levels)){#for both sexes
  for(j in 1:length(species.levels)){#for both species
    #select data to be plotted:
    plot.data=subset(xdata, sex==sex.levels[i] & species==species.levels[j])
    #determine median and quartiles of the response, separately per value of the covariate:
    qs=tapply(X=plot.data$response, INDEX=plot.data$covariate, FUN=quantile, prob=c(0.25, 0.5, 0.75))
    xnames=names(qs)#extract values of the covariate for which data do exist
    qs=matrix(unlist(qs), nrow=length(qs), byrow=T)#turn qs into a matrix:
    #determine number of observations per combination of values of covariate and response:
```

```

plot.data=aggregate(x=!is.na(plot.data$covariate), by=plot.data[, c("covariate", "response")], FUN=sum)
#create the plot:
plot(plot.data$covariate, plot.data$response, xlim=range(xdata$covariate), col=grey(level=0.5),
      cex=sqrt(plot.data$x), ylim=range(xdata$response), las=1, pch=19, xlab="", ylab="", tcl=-0.25)
#add medians:
segments(x0=as.numeric(xnames)+hbw, x1=as.numeric(xnames)-hbw, y0=qs[,2], y1=qs[,2], lwd=2)
#add quartiles:
rect(xleft=as.numeric(xnames)+hbw, xright=as.numeric(xnames)-hbw, ybottom=qs[,1], ytop=qs[,3])
#determine new data object to get predictions:
xvals2=data.frame(sex=sex.levels[i], species=species.levels[j], z.cov=xvals)
#get predicted values:
yvals=predict(object=model.res, newdata=xvals2, type="response")
lines(x=plot.xvals, y=yvals, lty=2, lwd=2)#and add them to the plot
}
}
#add axis label for x-axis:
par(mar=c(0.2, 1.8, 0.2, 0.2))#set margins for x-axis label (2 and 4 need to remain)
plot(x=1, y=1, type="n", xaxt="n", yaxt="n", ann=F, bty="n")#get empty plot
text(x=1, y=1, labels="covariate", cex=2)#add text
#add axis label for y-axis:
par(mar=c(1.8, 0.2, 0.2, 0.2))#set margins for y-axis label (1, and 3 need to remain)
plot(x=1, y=1, type="n", xaxt="n", yaxt="n", ann=F, bty="n")#get empty plot
text(x=1, y=1, labels="response", cex=2, srt=90)#add rotated text
#add factor level labels:
par(mar=c(0.2, 1.8, 0.2, 0.2))#set margins for factor species level's names (keep 2 and 4)
for(i in 1:length(species.levels)){#for both species
  plot(x=1, y=1, type="n", xaxt="n", yaxt="n", ann=F, bty="n")
  text(x=1, y=1, labels=species.levels[i], cex=2)
}
par(mar=c(1.8, 0.2, 0.2, 0.2))#set margins for factor species level's names (keep 1 and 3)
for(i in 1:length(sex.levels)){#for both sexes
  plot(x=1, y=1, type="n", xaxt="n", yaxt="n", ann=F, bty="n")
  text(x=1, y=1, labels=sex.levels[i], cex=2, srt=90)
}
}

```

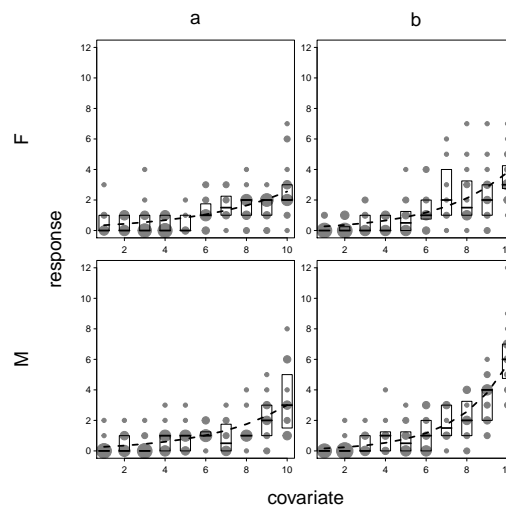


Figure 51: Illustration of how boxes and bubbles can be combined. The plot was created using the functions `points` (for the data shown), `rect` and `segments` (for the boxes), and `lines` (to depict the model lines).

This seems like a lot of code to get Fig. 51, and it surely is. However, in case you need such plots more frequently, you should consider writing a function which takes over all that business (see section 11). This piece

here largely serves in demonstrating/explaining how to do such a thing. Furthermore, the code uses more lines than needed. For instance, it isn't really needed to create `plot.data` as a subset of `xdata`. However, it makes subsequent code slimmer and also more easy to be read, and that's why the code has more lines than needed. I also want to emphasize again that such code is probably rarely ever just written down, but instead built up slowly until the result looks as desired. Apart from all that: if you think this is complicated, think for a minute about how much effort it would take to create the same plot in any other software you know (and whether you would even think of trying that). And if that doesn't convince you that R is cool, think a minute about how much time it would take you to adjust the plot in case the data have changed (something happening suprisingly frequently)... in R: not a minute; in any other program: I don't want to even think about...

In any case, here are a few comments about the above piece of code: the first thing to comment about is that there are *two* vectors representing the covariate (`xvals` and `plot.xvals`). One of them (`plot.xvals`) represents the covariate at its original scale and is used to plot the model; the other `xvals` represents the covariate in z-transformed scale and is used to get the predicted values. The two are needed since we are generating the plot with the covariate being at its original scale, but the model was fitted with the covariate being z-transformed. The other thing to worry about is the way how I obtained predicted values. For this I first generated a data frame (using `xvals2=data.frame(sex=sex.levels[i], species=species.levels[j], z.cov=xvals)`) and then used the function `predict` to obtain predicted values. The function `predict` can be used to get predicted values for some new data (argument `newdata`) based on a GLM (argument `object`). Here the argument `newdata` gets handed over the particular constellation of the factors being relevant for the respective plot, in combination with all values of `xvals` (the covariate in z-transformed space which was used to fit the model (in this context it is important to be aware that the data frame handed over to the argument `newdata` of the function `predict` comprises a column for each of the predictors used to fit the model, and that these must have names being equal to the predictors in the model)). The predicted values obtained from the function `predict` and that I used to add the line were, hence, based on the covariate in z-transformed space. To get it into the plot, however, it needs to be plotted against the covariate being at its original scale since this is what the figure is based on. Finally, the margins deserve some attention when the plotting regions 5 to 10 are filled with axis labels and factor level names. For these, some margins were changed to achieve a maximum of space available for plotting, but others were not. The margins that I didn't change are those that align to the respective axis of the actual plotting window. For instance, when preparing plotting region number 5 (Fig. 50), I changed margins 1 and 3 but not 2 and 4 because the latter two are those being relevant for the alignment of the text displayed in plotting region 5 in relation to the figures presented in the plotting regions 1 to 4. Correspondingly, the margins number 1 and 3 of plotting region number 6 need to be equal to the respective margins of the plotting regions 1 to 4 to achieve proper alignment of the axis label.

10.4 Two y-axes in one Plot

Sometimes one wants to combine two or more line charts in a single plot, for instance, to illustrate daily weather over the course of the year. The problem with such a plot can be that the different variables vary at different scales and, hence, one needs two different y-axes. Here I show how such a plot can be achieved.

I begin with creating two variables, say 'rain' and 'temperature', to be depicted and also a vector with dates against which to plot them:

```
set.seed(3)
xdate=as.Date(as.Date("2010-01-01"):as.Date("2010-12-31"), origin="1970-01-01")
rain=(2+sin(2*pi*as.numeric(xdate)/365))*9+rnorm(n=length(xdate))
temp=20+cos(2*pi*as.numeric(xdate)/365)*2+rnorm(n=length(xdate), sd=0.25)
par(mar=c(3, 3, 0.2, 0.2), mgp=c(1.7, 0.3, 0), las=1, tcl=-0.15, mfrow=c(2, 1))
plot(x=xdate, y=rain, xlab="date", ylab="rainfall", type="l")
plot(x=xdate, y=temp, xlab="date", ylab="temperature", type="l")
```

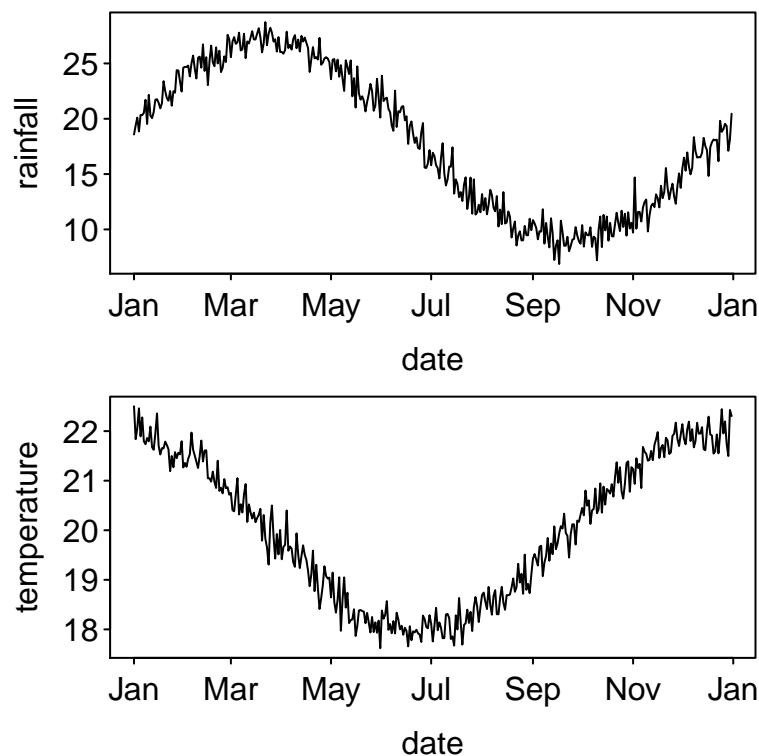



Figure 52: Data to be plotted in a single line chart.

Now the two plots shown in Fig. 52 should be combined into one. Certainly one could depict both of them at the same scale, but this would depict the variation in temperature very badly (since it would be fairly minor as compared to the variation in rainfall); and it also would be somewhat meaningless since 20 degree Celsius and 20 millimeters of rain aren't really the same thing. A simple solution to this is to begin with plotting rain as it is, and then adding temperature after having rescaled it such that its variation as it appears in the plot is comparable to that of rainfall. I would first rescale it to a range from 0 to 1 (by subtracting the minimum temperature and then dividing by the difference between its maximum and its minimum). Then I'd rescale it to the range of rainfall (by first multiplying with the difference between maximum and minimum rainfall and then adding minimum rainfall; this is actually just the inversion of the scaling to a range from 0 to 1, but with different parameters). After having added the respective line to the plot, the last thing needed is to add a second y-axis at the right margin of the plot. For this I'd use the function `pretty` to determine reasonable labels of ticks and then apply the same operations as to temperature to determine where these ticks should be put. Here is how that goes (see Fig. 53 for the result):

```
#set parameters (note space left to the right margin):
par(mar=c(3, 3, 0.2, 3), mgp=c(1.7, 0.3, 0), las=1, tcl=-0.15)
#begin with plot
#suppress y-axis ticks and labels since these should be depicted in blue (other than the x-axis):
plot(x=xdate, y=rain, xlab="date", ylab="", type="l", col="blue", yaxt="n")
axis(side=2, at=pretty(rain), col.axis="blue")#add axis
mtext(text="rainfall", col="blue", side=2, line=1.7, las=0)
#add temperature:
#first rescale to range from 0 to 1:
resc.temp=(temp-min(temp))/diff(range(temp))#now numbers fall between 0 and 1
#then rescale to range of rain:
resc.temp=resc.temp*diff(range(rain))+min(rain)
lines(x=xdate, y=resc.temp, col="red")
#add axis for rain to the right margin (note how what is handed over to argument 'at' is rescaled
#as was temperature itself:
labels=pretty(temp)#determine labels to be depicted...
```



```

at=(labels-min(temp))/diff(range(temp))#rescale to temperature range from 0 to 1
at=at*diff(range(rain))+min(rain)#rescale to range of rain
axis(side=4, at=at, labels=labels, col.axis="red")#add axis
mtext(text="temperature", col="red", side=4, line=1.7, las=0)

```

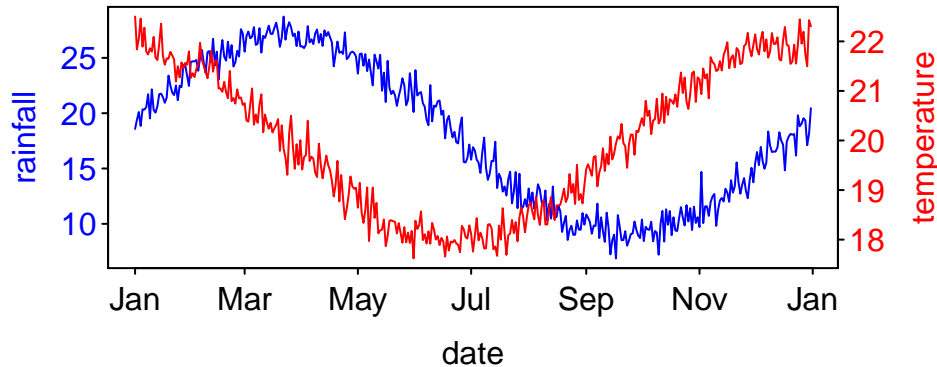


Figure 53: Depicting two line charts in one (note that the two variables vary at different scales because of which two y-axes are needed).

10.5 Data availability per Individual

Another example could be that one wants to illustrate data availability/sampling of a couple of individuals over time and also show some property of the individuals as it varied over the course of the study period. Here I take as an example female sexual swellings which vary from 0 to 3 and the availability of urine samples per day. Not each individual was observed each day, but for each day a female was observed a swelling score is available. Urine samples were hard to get, and, hence, not each day a female was observed a uring sample is available. As usual, I begin with creating some data to play around with:

```

set.seed(1)
n.females=10
n.days.per.female=sample(x=50:70, size=n.females, replace=T)
xdate=as.Date(as.Date("2010-01-01"):as.Date("2010-03-31"), origin="1970-01-01")
xdate=unlist(lapply(X=n.days.per.female, sample, x=xdate, replace=F))
xdate=as.Date(xdate, origin="1970-01-01")
fem.id=as.factor(rep(paste("F", 1:n.females, sep="."), times=n.days.per.female))
swell.score=sample(x=0:3, size=length(fem.id), replace=T, prob=c(4:1))
urine.sample=sample(x=0:1, size=length(fem.id), replace=T, prob=c(3, 1))

```

Now the task is to plot these data. Data for each female should be depicted at a particular elevation above the x-axis which should be one to ten for the ten females. For each day a female was observed, a rectangle should depict its swelling score. Scores from 0 to 3 should be depicted by rectangles being white, yellow, orange, and red, respectively. However, white rectangles wouldn't be visible on a white background and, hence, the background of the plotting region should be grey. Finally, days at which urine samples could be collected should be indicated by rectangles having a black edge (otherwise they should have no edge). Here's one way of how this could be achieved (the resulting plot can be seen in Fig. 54):

```

#set parameters:
par(mar=c(3, 3, 0.2), mgp=c(1.7, 0.3, 0), las=1, tcl=-0.15)
#begin with plot
#suppress y-axis ticks and labels (should show female ids later); suppress showing data;
#also suppress x-axis ticks as these don't show up nicely:
plot(x=1, y=1, xlab="date", ylab="", type="n", xlim=range(xdate), ylim=c(0.5, length(levels(fem.id))+0.5),
     yaxs="i", yaxt="n", xaxt="n")
#add grey background:
#this is based on that the argument 'usr' of the function par returns the coordinates

```

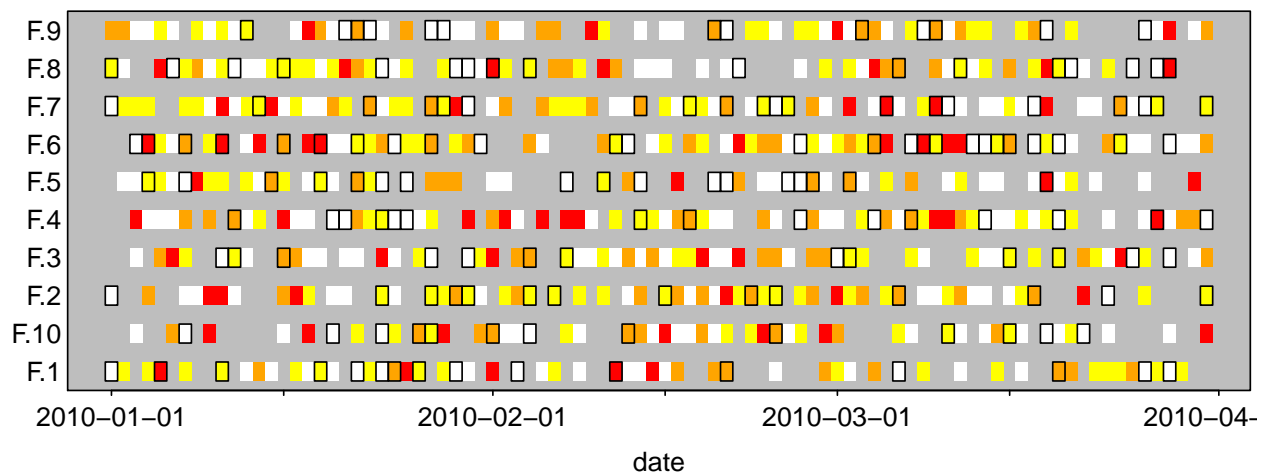


Figure 54: Example of a plot showing some observed properties of individuals (here a sexual swelling score; depicted by the color of the filling of the rectangles) and data availability (here the availability of urine samples, depicted by rectangles having a black edge).

```
#of the edges of the plotting region after a plot has been created
rect(xleft=par()$usr[1], xright=par()$usr[2], ybottom=par()$usr[3], ytop=par()$usr[4], col="grey")
#add female ID to y-axis:
mtext(text=levels(fem.id), side=2, at=1:length(levels(fem.id)), line=0.2, las=1)
#add x-axis:
axis(side=1, at=as.numeric(pretty(xdate)), labels=pretty(xdate))
#create scalars denoting half the rectangle width and height:
hrw=0.5#half the rectangles' width = half a day
hrh=0.25#half the rectangles' height = a quarter of the distance between lines per female
#create vector with colors depicting swelling score for the filling of the rectangles:
f.col=c("white", "yellow", "orange", "red")[1+swell.score]
#add rectangles depicting swelling score (border is not doing to be depicted):
rect(xleft=xdate-hrw, xright=xdate+hrw, ybottom=as.numeric(fem.id)-hrh, ytop=as.numeric(fem.id)+hrh,
     col=f.col, border=NA)
#create vector with colors of the edges of the rectangles:
e.col=rep(NA, length(f.col))
#set edge color to black when sample is available:
e.col[urine.sample==1]="black"#(edge color=NA equates to no edge appearing)
#add rectangles (only border) on top of them (has the effect that the borders are not covered by
#adjacent rectangles) showing availability of urine samples:
rect(xleft=xdate-hrw, xright=xdate+hrw, ybottom=as.numeric(fem.id)-hrh, ytop=as.numeric(fem.id)+hrh,
     col=NA, border=e.col)
```

10.6 Getting started with your own customized Plots

As said above, the number of possible data and/or models to be depicted are as innumerable as the options available are countless. Hence, it will never be possible to give an even only half-way (not even 10th-way or 100th-way) complete overview of what could be done. So it will frequently be up to you to come up with a solution. When thinking about how to depict your data and/or models, don't limit your thinking! Ideally, at the beginning you don't even think about R, the plotting functions you know, and all that; but just take a blank piece of paper and a pen and then think about what would be the best possible visualization for what you want to see or show. Only once you have a good idea about how the plot should look like, start thinking about how to create it in R. So really don't limit your thinking! Be assured that whatever the plot should look like, it will be possible to create it in R! Actually, quite regularly I get a question like "Roger, is it possible to create a plot that..."; and whenever I get such a question my answer is "yes", even before I've heard the complete question...

11 Writing your own plotting Functions

Sometimes you will have the impression of reinventing the wheel again and again, and this is certainly not what one should waste one's time with. For instance, I quite regularly need x-axes showing dates, but at the same time never was happy with how standard R-functions depicted them. So I each time added the axis manually, using quite a few function calls which each time was somewhat laborious and time consuming. One day I discovered that I had done more or less the same thing in the context of different projects and data already several times and got really tired of it. So I decided to write myself a function adding dates to the x-axis of a plot the way I wanted it. Since then, whenever I need such a thing, I source the function and then call it. This replaces some 10 to 15 lines of code (which need to be written and in part reinvented) by two (which solely consist of sourcing the function and handing over objects to its arguments) - very easy and convenient.

In this section I want to show you how to write and use your own plotting functions. The example I use is a plot for related data (or repeated measures). The basic structure of the data is that a couple of, for instance, individuals was observed under a few different conditions whereby all or most individuals were observed under each of the conditions. The plot should show the average response per individual and condition and depict the number of observations per individual and condition by the area of the points. In addition, observations of the same individual from 'adjacent' conditions should be linked by a dashed line.

Writing a function in R isn't such a big deal. Here I begin with thinking about what the function needs as input or, in other words, what its arguments need to be: it certainly needs vectors for subject ID, the condition, the response, and the sample size. So let's begin with just such a basic version of it:

```
related.data.plot<-function(subject.id, condition, response, n){
  #catch a couple of possible input errors:
  #input vectors vary in lengths:
  if(length(unique(c(length(subject.id), length(condition), length(response), length(n))))>1){
    stop("Error: not all of subject.id, condition, response, and n are of identical length")
  }
  if(!is.numeric(response)){#response isn't numeric
    stop("Error: response isn't numeric")
  }
  if(!is.numeric(n)){#sample size isn't numeric
    stop("Error: n isn't numeric")
  }
  #determine average response per individual and condition (needed in case there are some individuals
  #for which there is more than one observation per condition):
  to.plot=apply(X=response, INDEX=list(subject.id, condition), FUN=mean, na.rm=T)
  #determine sample size per individual and condition:
  n=apply(X=n, INDEX=list(subject.id, condition), FUN=sum, na.rm=T)
  #set up the plot:
  par(mar=c(2, 3, 0.2, 0.2), mgp=c(1.7, 0.3, 0), tcl=-0.15, las=1)
  plot(x=1, y=1, xlim=c(0.5, ncol(to.plot)+0.5), xaxs="i", ylim=range(to.plot, na.rm=T), type="n",
    ylab="response", xlab="", xaxt="n")#nothing plotted so far
  #add lines connecting observations from the same individual in 'adjacent' conditions:
  #here I loop through all but the first column of to.plot
  for(i in 2:ncol(to.plot)){
    #add segments from column i-1 to column i of to.plot
    segments(x0=i-1, x1=i, y0=to.plot[, i-1], y1=to.plot[, i], lty=2)
  }
  #add points:
  for(i in 1:ncol(to.plot)){
    #first a white point to cover the line:
    points(x=rep(i, nrow(to.plot)), y=to.plot[, i], pch=19, col="white", cex=sqrt(n[, i]))
    #then a semi transparent point:
    points(x=rep(i, nrow(to.plot)), y=to.plot[, i], pch=19, col=grey(level=0.5, alpha=0.5), cex=sqrt(n[, i]))
    #note how it is ensured that the area of the points is proportionate to sample size
  }
  #add labels to x-axis:
  mtext(text=colnames(to.plot), side=1, at=1:ncol(to.plot), line=0.2)
}
```

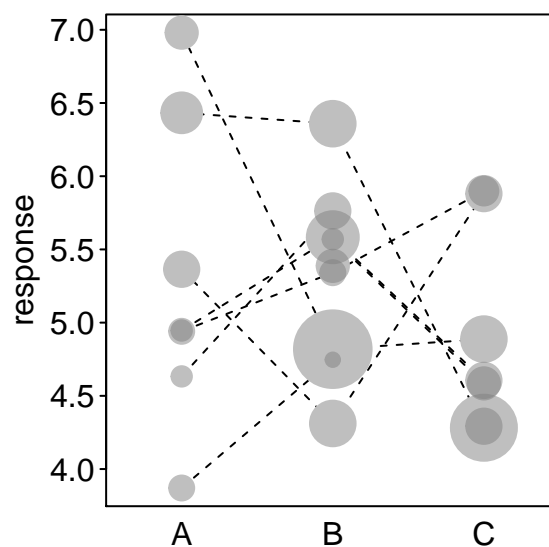


Figure 55: Example output from a self-written function creating a plot for repeated measures data.

Now that the function is written it needs to be pasted into R, and then it can be used (you can also save it and use the function `source` to make it available to R; certainly the better option on the long run). Here's an example (Fig. 55):

```
#create some data to test the function:
set.seed(1)
N=30
ind.id=as.factor(sample(x=letters[1:10], size=N, replace=T))
treat=as.factor(sample(x=LETTERS[1:3], size=N, replace=T))
rv=rnorm(n=N, mean=5)
n.obs=sample(x=1:10, size=N, replace=T)
#now you can call the function:
related.data.plot(subject.id=ind.id, condition=treat, response=rv, n=n.obs)
```

So far, the function lacks flexibility as you certainly notice. For instance, the size of the points is not scalable meaning that when sample sizes are large, then the points will be incredibly large. This could be changed by adding another argument `size.fac` and giving it a default value of 1:

```
related.data.plot<-function(subject.id, condition, response, n, size.fac=1){
  #catch a couple of possible input errors:
  #input vectors vary in lengths:
  if(length(unique(c(length(subject.id), length(condition), length(response), length(n))))>1){
    stop("Error: not all of subject.id, condition, response, and n are of identical length")
  }
  if(!is.numeric(response)){#response isn't numeric
    stop("Error: response isn't numeric")
  }
  if(!is.numeric(n)){#sample size isn't numeric
    stop("Error: n isn't numeric")
  }
  #determine average response per individual and condition (needed in case there are some individuals
  #for which there is more than one observation per condition):
  to.plot=tapply(X=response, INDEX=list(subject.id, condition), FUN=mean, na.rm=T)
  #determine sample size per individual and condition:
  n=tapply(X=n, INDEX=list(subject.id, condition), FUN=sum, na.rm=T)
  #set up the plot:
```

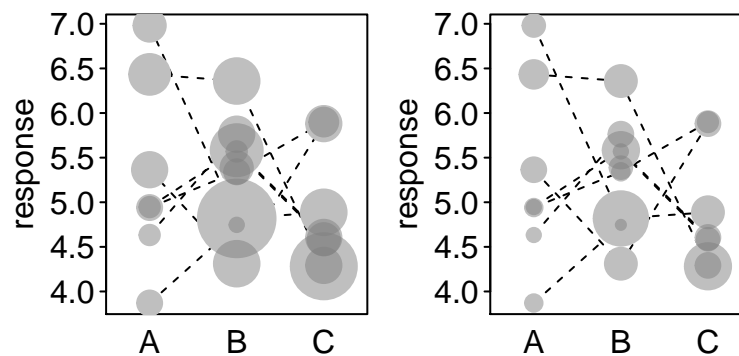


Figure 56: Example output from a self-written function creating a plot for repeated measures data. Note how the two plots differ regarding the size of the points.

```
par(mar=c(2, 3, 0.2, 0.2), mgp=c(1.7, 0.3, 0), tcl=-0.15, las=1)
plot(x=1, y=1, xlim=c(0.5, ncol(to.plot)+0.5), xaxs="i", ylim=range(to.plot, na.rm=T), type="n",
     ylab="response", xlab="", xaxt="n")#nothing plotted so far
#add lines connecting observations from the same individuals in adjacent conditions:
#here I loop through all but the first column of to.plot
for(i in 2:ncol(to.plot)){
  #add segments from column i-1 to column i of to.plot
  segments(x0=i-1, x1=i, y0=to.plot[, i-1], y1=to.plot[, i], lty=2)
}
#add points:
for(i in 1:ncol(to.plot)){
  #first a white point to cover the line:
  points(x=rep(i, nrow(to.plot)), y=to.plot[, i], pch=19, col="white", cex=sqrt(n[, i])*size.fac)
  #then a semi transparent point:
  points(x=rep(i, nrow(to.plot)), y=to.plot[, i], pch=19, col=grey(level=0.5, alpha=0.5),
        cex=sqrt(n[, i])*size.fac)
  #note how it is ensured that the area of the points is proportionate to sample size
}
#add labels to x-axis:
mtext(text=colnames(to.plot), side=1, at=1:ncol(to.plot), line=0.2)
}
```

Now one can use the function as before, and when one doesn't hand over anything to the argument `size.fac` it will assume a value of 1. But when one does hand over a value other than 1 to it the plot will change. Here's an example (Fig. 56):

```
#create some data to test the function:
par(mfrow=c(1, 2))
related.data.plot(subject.id=ind.id, condition=treat, response=rv, n=n.obs)
related.data.plot(subject.id=ind.id, condition=treat, response=rv, n=n.obs, size.fac=0.7)
```

Certainly, there would be many other things on the wishlist for such a function. I would, for instance, add the following functionality to it: (i) a possibility to customize the order in which the conditions are displayed along the x-axis; (ii) a possibility to show labels at the x-axis which differ from the levels of what is handed over to the argument `condition`; (iii) a possibility to freely chose the colors with which the points are depicted; and possibly many others. Perhaps try some of these (or others) as an exercise.

When adding new arguments to a self-written function it is an extremely good idea to give each of them a default which reproduces the behaviour of the function before the addition of the argument. This has several advantages: first, older projects which used the function in its original state would still work without any need to change the call of the function, and they would produce the exact same result as before (if the default is chosen appropriately). Second, probably the original choice was a reasonable one, so make it the default,

and give the user (also you yourself) the option to not beeing forced to bother about *all* the arguments of the function (which also means less code to produce when calling the function, reduced risk of typos, etc.).

I really use the option to write my own plotting functions a lot and use such functions on a daily basis. Over the past years I've written some which create quite complex plots and replace several hundred lines of code by a single function call; and some of them work pretty generically (meaning that I can apply them in a whole bunch of situations/contexts). With increasing experience you might become more ambitious with regard to writing your own functions. When doing so, be aware that also plotting functions - like plots - are usually not written at once. Rather, begin with a basic version and then later add arguments as you (or others) need them. Most of my own more complex plotting functions have quite a bit of 'history' in the sense that the number of available arguments grew over the years. The first version of most of them was fairly simple to be programmed, and they gained complexity only later on. Keep that in mind when trying to write your own functions.

12 Defining the Size of Plotting Windows

So far I have just shown how to create plots using the function `plot`. When calling it, plots appear in a quadratic plotting window of a standard size. However, this might not be an optimal choice. For instance, many plots I create are wider than they are high. In such a case its certainly possible to use the mouse to adjust the width or height of the plotting window. However, this has the disadvantage that it'll be pretty impossible to exactly reproduce the dimensions/proportions of the plotting window later (and the additional disadvantage that one has to use the ugly the mouse). An alternative is the use of a function determining the size of the plotting window. This is done using the function `X11` (Linux), `windows` (Windows), or `cairo` or `quartz` (Mac OS). As usual, these functions provide plenty of options, but so far I have only used two of their arguments, namely `width` and `height`, both of which default to 7. To alter the size and/or proportions of the plotting window one just calls one of these functions with the desired value handed over to `width` and/or `height`. Each time one calls one of these functions, a new plotting window is opened without closing the already existing one(s). Hence, to get rid of the already created plot, I usually call it after calling `dev.off` which (by default) closes the last plotting window created. I use this really a lot and can just recommend the use of `X11` (or whatever the function on your OS is) whenever one wants to define the proportions or size of the plotting window.

13 Saving Plots

After having shown how to create various kinds of plots, it seems worth to briefly consider how to save them. Before that, however, I feel I should add a few words about the pros and cons of different file types available.

13.1 File Types

Let me begin with the ones I would *not* use which are `'jpg'`, `'jpeg'`, and `'bmp'`. The former two I don't use because the compression involved to create them leads to edge artifacts and texts not looking nice at all (although modifying the compression level might alleviate the issue to some extent). Bitmaps (`'bmp'`) I never use because they are a very inefficient way of saving plots (i.e., may need lots of disc space), they usually look simply awful (pixelish), and they also are not scalable (i.e., enlarging them may lead to the individual pixels becoming very visible).

My favourite file types are the following:

- `'wmf'` (`'emf'`) (`'Windows Meta File'`): these represent vector graphics which are very efficient (i.e., lead to small files) when it comes to saving figures with simple graphical elements (most plots produced as part of this tutorial except for those which display many pixels in different colors like maps with gradients displayed by color gradients). They also have the advantage of being scalable (i.e., they can be enlarged without losing resolution' (well, they don't really have a 'resolution' since they are 'vector graphics')). However, most people using operating systems other than Windows may have difficulties to open them, and wmf-files also do *not* support transparent colors;
- `'pdf'` (`'eps'`/`'ps'`): these are very efficient (i.e., usually lead to small files), support transparent colors (only pdf,

as far as I know), work well with complex plots (e.g., high resolution maps with gradients displayed by color gradients), and particularly pdf-files can be easily opened on every operating system;

- 'tif' ('tiff'): both are pretty inefficient since they save the color values for each pixel in the plot, separately, and, hence, easily lead to very large files (particularly when using higher resolutions). This is the reason why I hardly ever use them. However they are required by some journals (don't know why) and this is why I use them occasionally;

- 'png': this is a compressed format which leads to pretty reasonably looking plots. However, edge artifacts can appear, and I generally find edges to look a little 'blurry', so never use it for figures to be submitted to journals, printed on posters, shown in presentations (well, I frequently use them in presentations...), etc. However, it is useful for a 'quick look', e.g., when many plots are produced to inspect distributions of many variables, the availability of data for many individuals, whatever; and they are good for being sent to others via email because png-files are quite small (due to the compression), and they work well with all operating systems.

To my knowledge, png, pdf, and tiff are the only file types allowing for transparent colors.

13.2 Saving Plots using `savePlot` or `dev.copy2pdf`

Saving plots on a Windows machine is easy: just create the plot and then use the function `savePlot`. This has only two essential arguments, namely `filename` which expects character input and may include the path (e.g., "d:\temp\my_first_plot") and `type` which expects a character entry, too, which has to be one of "wmf", "emf", "png", "jpg", "jpeg", "bmp", "tif", "tiff", "ps", "eps", or "pdf". On a Mac or Linux machine one can use the same function. However, it accepts fewer filetypes, namely "png", "jpeg", "tiff", and "bmp" (not a very useful choice), and on a MacOS machine it works only when the plot was created using the function `quartz` (to my knowledge).

Another option to save a plot which works on every OS is to first create the plot as usual (e.g., using the function `plot`) and then save it into a pdf file using the function `dev.copy2pdf`. The most basic way to use it is to just hand over the file name (potentially including the path) to the argument `file` (which is actually an argument of the function `pdf` to which it is passed). However, also other arguments of the function `pdf` can be handed over to the function `dev.copy2pdf`.

On all operating systems it is also possible to create/write plots directly within files (e.g., functions `tiff`, `pdf`), an option particularly desirable when it is needed to create tiff-files with high resolution (which some journals do require). However, this is somewhat tricky because certain adjustments need to be made in order to let these plots look nice/as desired/expected, and as off now I can't treat that topic sufficiently (since I rarely do/need that; but see the next section for some hints).

13.3 Saving Plots using the Function `tiff`

A somewhat special issue are tiff files which are occasionally required by journals (without allowing any alternatives). I use the tiff format very rarely, simply because the resulting files tend to get very large. However, there are plots which are 'pixelish' by nature, for instance, a map of the spatial distribution of habitat types or elevation (e.g., Fig. 37, 38, or 39), and for such figures the tiff format might be a useful option. Using Windows as the operating system one can certainly create the plot as usual and then save it using the function `savePlot`. However, doing so does not allow the adjustment of the resolution (and the journals requiring tiff files usually have certain requirements regarding the resolution or the size of the figure), nor is this option available on other operating systems. Hence, one will usually use the function `tiff` for this purpose. The principle use of the function comprises three steps: (i) create the file using the function `tiff`; (ii) add a plot and perhaps other graphical elements to it; (iii) close the file using the function `dev.off`.

The arguments determining the size of the figure are `width`, `height`, and `units`. The arguments `width` and `height` take numbers as input, and `units` takes the units in which these are indicated (can be one of "px" (for pixels), the default), "in" (inches), "cm", or "mm"). The resolution is defined using `res` which sets the number of 'pixels per inch' ('ppi'; when `units` is set to, e.g., "cm" `res` determines the number 'pixels per centimeter'). I rarely use this function and always found it a bit troublesome since the resulting figure tends to look strikingly different from what I got when creating the plot using a standard R plotting window. However, recently I discovered that the trouble disappears when using inches as the unit. The following code chunk shows an example (you will need to change the working directory and then open the file to see it):

```

set.seed(1)
n=10
setwd("/home/temp")
tiff(filename="test_tiff.tiff", units="in", width=5, height=5, res=300)#open file
par(mar=c(3, 3, 0.5, 0.5), mgp=c(1.7, 0.3, 0), tcl=-0.15)#begin with plot
set.seed(1)
plot(x=runif(10), y=runif(10), las=1)#create plot
dev.off()#close file ('device')

```

When handing "cm" over to the argument `units` the resulting plot looks strikingly different as can be seen when trying the following example:

```

set.seed(1)
n=10
setwd("/home/temp")
tiff(filename="test_tiff.tiff", units="cm", width=5, height=5, res=300)#open file
par(mar=c(3, 3, 0.5, 0.5), mgp=c(1.7, 0.3, 0), tcl=-0.15)#begin with plot
set.seed(1)
plot(x=runif(10), y=runif(10), las=1)#create plot
dev.off()#close file ('device')

```

One way of fixing this to some extent is use of the argument `pointsize`. It has a default of 12 which means 1/72 inch when using `units="in"`. Leaving it as its default leads to an okay looking result with `units="in"` but not when setting `units` to "cm". But with `pointsize=6` things already look more like one would expect:

```

set.seed(1)
n=10
setwd("/home/roger/temp")
tiff(filename="test_tiff.tiff", units="cm", width=5, height=5, res=300, pointsize=6)#open file
par(mar=c(3, 3, 0.5, 0.5), mgp=c(1.7, 0.3, 0), tcl=-0.15)#begin with plot
plot(x=runif(10), y=runif(10), las=1)#create plot
dev.off()#close file ('device')

```

Still not quite exactly what I'd expect, but closer. To modify the appearance (actually thickness) of the lines, points, etc. I'd fiddle around with the functions `axis` and `box` (after having created the plot suppressing axes and the box around it using `axes=F` and `bty="n"`, respectively). But simply using `units="in"` is certainly a way easier option.

13.4 Saving a Plot using the CMYK Color Model

Some journals require figures so be submitted to be based on the CMYK color model ('cyan, magenta, yellow, and key (black)'). Of course, also this is possible in R. The probably simplest option is to call the function `dev.copy2pdf` (see section 13.2) with the argument `colormodel="cmyk"`. That's it.

If the desired file format is tiff then one can first create the plot in a standard plotting window and then save it using the function `dev2bitmap`. This has arguments for the size of the plot (`width`, `height`, and `units`) and its resolution (`res`; see section 13.3 for more). When using this function you have a lot of options for the file format (see `?dev2bitmap`). One of these is is a tiff using CYMK which you get when calling the function with `type="tiff32nc"`. Like the function `tiff`, also `dev2bitmap` doesn't necessarily give you a plot looking exactly like that in a standard plotting window, and some adjustments of line widths, character expansion and the like might be required.

14 Plots for Papers and Presentations

A frequently neglected or overlooked issue is that showing the same plots in papers and presentations is rarely a good idea. The reason is that for plots to be presented in papers one should aim at minimizing the 'ink-to-information ratio' but for presentations this is a very bad idea. There are many reasons for minimizing

the ink-to-information ratio for plots to be shown in papers, one of which, in my view, is to simply save toner. Assume you have a widely read paper which includes a couple of bar charts. If the bars are filled black, eventually a couple of squaremeters of plain black paper will be printed; but if the bars are unfilled and only have a black edge, the amount of toner needed will be way smaller (reducing resource consumption). In addition to this environmental concern, I also find unfilled bars with black edges, open circles and other such somewhat more modestly looking elements aesthetically nicer than filled elements (which I tend to find somewhat 'brutal'). However, in presentations such graphical elements don't really work at all, simply because the crucial information is depicted by thin lines. Such thin lines, in turn, might be barely visible (if at all), for instance, when lightning conditions aren't optimal, the beamer's resolution isn't too great, or also for people who sit in the back half of the audience. So, when preparing plots for a presentation one should think about how the information can be depicted as salient as possible, and simply not using *any* thin lines (with the exception of the axes) will be very helpful in this context. This means, for instance, that points should be filled, confidence intervals of models should not be shown using lines but polygons, or the boxes in boxplots should be colored. Below you can find two examples for how plots could differ depending on whether they are shown in a paper or a presentation (Fig. 57).

```
set.seed(1)
n=50
pv=as.factor(sample(LETTERS[1:2], n, replace=T))
rv=rnorm(n=n, mean=5+as.numeric(pv), sd=0.5)
xx=apply(X=rv, INDEX=pv, FUN=quantile, prob=c(0.025, 0.25, 0.5, 0.75, 0.975))
xx=matrix(unlist(xx), nrow=2, byrow=T)
par(mar=c(3, 4, 0.2, 0.2), mgp=c(1.5, 0.2, 0), tcl=-0.1, las=1, mfrow=c(2, 2))
hbw=0.35
#boxplot for paper:
plot(x=1, y=1, xlim=c(0.5, 2.5), ylim=range(xx), xaxs="i", xast="n", xlab="", ylab="response", type="n",
     cex.axis=0.7)
segments(x0=1:2, x1=1:2, y0=xx[, 1], y1=xx[, 5])
rect(xleft=(1:2)-hbw, xright=(1:2)+hbw, ybottom=xx[, 2], ytop=xx[, 4])
segments(x0=(1:2)-hbw, x1=(1:2)+hbw, y0=xx[, 3], y1=xx[, 3], lwd=2, lend=2)
mtext(text=levels(pv), side=1, line=0.2, at=1:2)
legend(x="topleft", legend="(a)", bty="n")
#boxplot for presentation:
plot(x=1, y=1, xlim=c(0.5, 2.5), ylim=range(xx), xaxs="i", xast="n", xlab="", ylab=bquote(bold(response)),
     type="n", cex.axis=0.7, cex.lab=1.5)
segments(x0=1:2, x1=1:2, y0=xx[, 1], y1=xx[, 5], lwd=4, lend=2)
rect(xleft=(1:2)-hbw, xright=(1:2)+hbw, ybottom=xx[, 2], ytop=xx[, 4], border=NA, col="red", lwd=4)
segments(x0=(1:2)-hbw, x1=(1:2)+hbw, y0=xx[, 3], y1=xx[, 3], lwd=4, lend=1)
lapply(X=1:length(levels(pv)), FUN=function(x){
  mtext(text=bquote(bold(. (levels(pv)[x]))), side=1, line=0.5, at=x, cex=1.5)
})
legend(x="topleft", legend="(b)", bty="n")
#another example:
pv=runif(n=n, min=0, max=10)
rv=rnorm(n=n, mean=pv, sd=3)
model.res=lm(rv~pv)
pred.data=data.frame(pv=seq(from=min(pv), to=max(pv), length.out=100))
ci=predict.lm(object=model.res, newdata=pred.data, interval="confidence")
#for paper:
plot(x=pv, y=rv, xlab="predictor", ylab="response", cex.axis=0.7)
segments(x0=min(pv), x1=max(pv),
        y0=coefficients(model.res)["(Intercept)"]+coefficients(model.res)["pv"]*min(pv),
        y1=coefficients(model.res)["(Intercept)"]+coefficients(model.res)["pv"]*max(pv), lty=1)
lines(x=pred.data$pv, y=ci[, "lwr"], lty=3)
lines(x=pred.data$pv, y=ci[, "upr"], lty=3)
legend(x="topleft", legend="(c)", bty="n")
#for presentation:
plot(x=pv, y=rv, xlab=bquote(bold(predictor)), ylab=bquote(bold(response)), cex.axis=0.7, cex.lab=1.5, type="n")
polygon(x=c(pred.data$pv, rev(pred.data$pv)), y=c(ci[, "lwr"], rev(ci[, "upr"])),
        border=NA, col="grey")
```

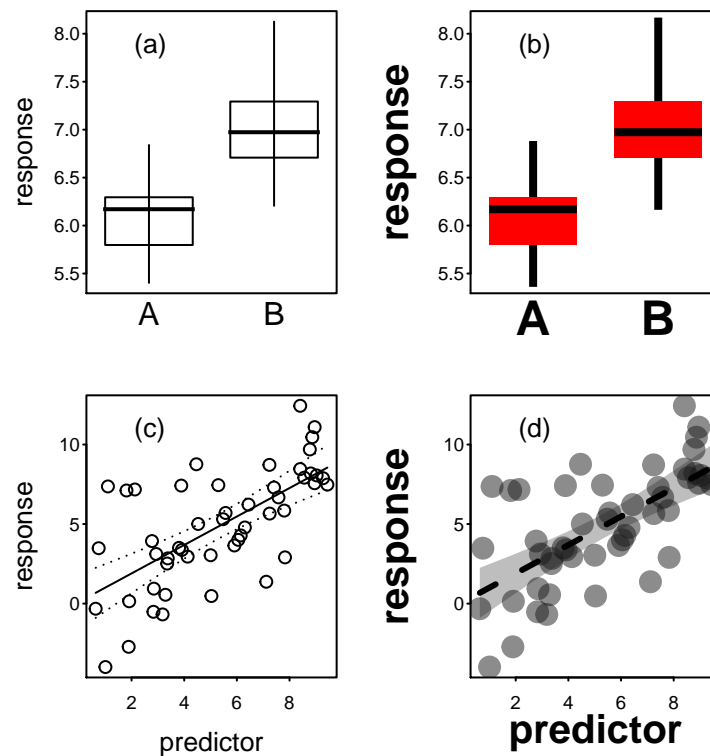


Figure 57: Examples for how plots could differ between when shown in a paper (left) and a presentation (right).

```
points(x=pv, y=rv, pch=19, cex=1.75, col=grey(level=0.1, alpha=0.5))
segments(x0=min(pv), x1=max(pv),
  y0=coefficients(model.res)["(Intercept)"]+coefficients(model.res)["pv"]*min(pv),
  y1=coefficients(model.res)["(Intercept)"]+coefficients(model.res)["pv"]*max(pv), lty=2, lwd=3)
legend(x="topleft", legend="(d)", bty="n")
```

15 and finally...

I hope that with this little tutorial I opened your eyes for how great R is (actually, its the greatest of all times) when it comes to creating fully customized plots. Quite frequently someone shows up in my office asking something like "Roger, is it possible to create a plot in R showing...", and my answer is always "yes" without having even heard what she or he wants to depict. Certainly, the way to create plots I showed here is somewhat laborious. But at the same time it opens such a multitude of options. And it pretty much guarantees that one will never get stuck in a dead-end-road, simply because one doesn't use canned functions which might or might not allow for a certain option.

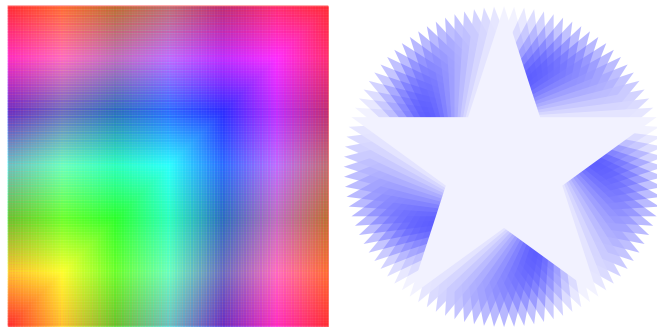
Certainly, this tutorial is pretty incomplete. In fact, I consider it a work in progress, and every once in a while I add smaller or larger sections to it. What I add depends pretty much on my current interests, what I've recently learned, or what I think people could be interested in. You can contribute to improving the usefulness of this tutorial by letting me know what you miss - I'll try to incorporate it in future versions of this tutorial. And you can improve it by letting me know about typos you found (the environment I use to work in doesn't have a spell checker, so there might be many...).

16 Acknowledgements

Thanks to Franka for her helpful comments on an earlier version of this tutorial and for pointing out many typos!

17 References

- McCullagh P & Nelder JA. 2008. Generalized linear models. Chapman and Hall. London.
- R Core Team. 2017. R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing. Vienna, Austria.
- Xie Y. 2013. Dynamic Documents with R and knitr. Chapman and Hall/CRC. ISBN 978-1482203530
- Xie Y. 2014b. knitr: A Comprehensive Tool for Reproducible Research in R. In Victoria Stodden, Friedrich Leisch and Roger D. Peng, editors, Implementing Reproducible Computational Research. Chapman and Hall/CRC. ISBN 978-1466561595
- Xie Y. 2016. knitr: A general-purpose package for dynamic report generation in R. R package version 1.15.1.



Index

3-D plots, 43

abline, 13

- a*, 13
- b*, 13
- from *lm*, 14
- h*, 14
- v*, 14

adding points, 19

adj, **26**, 32

adjustcolor, 39

aggregate, 46

alpha, **40**

alpha.f, **40**

angle

- of shading lines in polygons, 22
- of shading lines in rectangles, 20

ann, **8**

arrows, 17

as.numeric, 10, 17, 51

asp, **21**

aspect ratio, 21

at, 27

axes, **9**

axis

- color, 57
- rescaling, 22

axis, 22

- color, 7

background

- of plotting region, 57

background color, 7

bar plot, 6

bg, **7**

binning, 45

bmp, 62

box, 9

boxplot, 50, 52, 54

bquote, 29, 32

bty, 10, 25, 49

bubble plot, 48–50, 52

cairo, 62

cex, 10, 19, 47, 49

cex.axis, **9**

cex.lab, **9**

CMYK color model, 64

col.lab, **7**

color

- axis, 7, 57
- axis labels, 7, 57
- background, 7

- by index, 34

- by name, 34

- by number, 34

- foreground, 7

- of border of rectangles, 20

- of rectangles, 20

- rainbow*, 35

- red-green-blue, 35

- transparent, 39

color gradient

- colorRamp, 38**

- grey*, 37

- rgb*, 37

colorRamp, 38

customized plots, 58

cut, 45

degrees of freedom, 33

density

- of shading lines in polygons, 22

- of shading lines in rectangles, 20

dev.copy2pdf, 63

dev.off, 62, 63

df, 33

digital elevation model, 36, 38

emf, 63

eps, 62, 63

expand.grid, 21

extent of axis limit beyond data range, 9

factor, 17

factor, 51

family, 42

fg, 7

file type

- bmp, 62

- emf, 63

- eps, 62, 63

- jpg, 62, 63

- pdf, 62, 63

- png, 63

- ps, 62, 63

- tiff, 63

- wmf, 62, 63

file types, 62

font, 42

foreground color, 7

getting several plots in one window, 11

Greek letters, 30

grey, 40

grey, 17, 37, 51

- grid lines, 14
- grouping by factor, 9
- height of plotting window, 62
- histogram, 6, 7
- interaction, 43, 52
- jpg, 62, 63
- predict**, 55
- labels*, 25
- las*, 9
- layout**, 11, 53
- layout.show**, 12
- legend, 41
- legend**, 24, 49
- lend*, 18, 41
- length of axis ticks, 9
- level*, 40
- levels**, 10, 17, 51
- line*, 27
- line plot, 6
- line type, 14, 18
- lines**, 15, 48
- logistic regression, 43
- lty*, 14, 18
- lwd*, 17, 18
- main*, 10, 29
- map, 36
- mar*, 7, 27
- mathematical expressions, 30
- mathematical symbols, 30
- mgp*, 7, 21
- model coefficients, 31
- model results in plots, 31
- mosaic plot, 5, 6
- mtext**, 17, 27, 29, 51
- outer**, 44
- P-value, 32
- par**, 6
 - bg*, 7
 - col.axis*, 7
 - col.lab*, 7
 - fg*, 7
 - general, 7
 - mar*, 7
 - mfc*, 11
 - mfrow*, 11
 - mgp*, 7
 - tcl*, 7
 - xpd*, 28
- paste**, 51
- pch*, 9, 19, 51
- pdf, 62, 63
- persp**, 43, 44, 46, 47
- plot**
 - asp*, 21
 - type, 6
 - type="n", 7
- plotmath**, 29
- plotting region
 - background, 57
- plotting symbols, 9
- png, 63
- points**, 19, 46, 48, 49
 - lwd*, 19
- Poisson regression, 15, 52
- poly**, 21
- polygon, 21
- pos*, 25
- position of text, 25
- presentation, 64
- pretty**, 23, 56
- programming plotting functions, 59
- ps, 62, 63
- quartz**, 62
- rainbow**, 35
- rect**, 20
- rectangles
 - color, 20
- regression line, 14
- related data, 48
- rev**, 51
- rgb**, 35, 38
- rotating text, 26
- sample size, 48
- savePlot**, 63
- scatter plot, 6
- segments**, 16, 41
- seq**, 15, 44
- set.seed**, 5
- side*, 27
- size of axis text, 9
- size of plotting window, 62
- size of symbols, 10
- srt*, 26
- straight line, 13
- style of box around plot, 10
- style of line end, 18
- suppress annotation, 8
- suppress axes, 9
- tcl*, 7, 9
- text
 - bold, 31

- bottom top adjustment, 26
- italic, 31
- left right adjustment, 26
- position relative to location, 25
- R^2 , 31
- rotation, 26
- subscripted, 31
- superscripted, 31
- text**, **25**, 27, 29
- tick length, 7, 9
- tied observations, 48, 52
- tiff, 63
- tiff**, 63
- trans3d**, 46, 48
- type*, **6**
- width of plot margins, 7
- width of plotting window, 62
- windows**, **62**
- wmf, 62, 63
- X11**, **62**
- xaxs*, **9**
- xaxt*, **9**, 17, 22, 28, 51
- xlab*, **8**, 29
- xlim*, 8, 17, 28
- yaxs*, **9**
- yaxt*, **9**
- ylab*, **8**, 29
- ylim*, 8