Chapter 3
High Quality Graphics in R

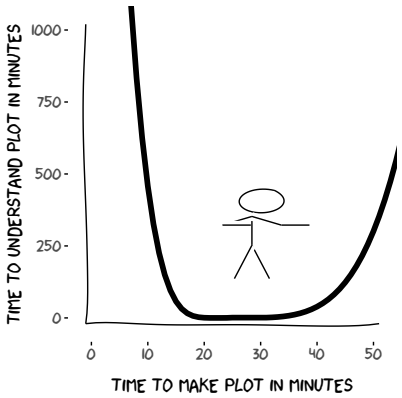


Figure 3.1: An elementary law of visualization.

There are (at least) two types of data visualization. The first enables a scientist to effectively explore data and make discoveries about the complex processes at work. The other type of visualization provides informative, clear and visually attractive illustrations of her results that she can show to others and eventually include in a publication.

Both of these types of visualizations can be made with R. In fact, R offers multiple graphics systems. This is because R is extensible, and because progress in R graphics has proceeded largely not by replacing the old functions, but by adding packages. Each of the different graphics systems has its advantages and limitations. In this chapter we'll use two of them. First, we have a cursory look at the base R plotting functions[1]. Subsequently we will switch to *ggplot2*.

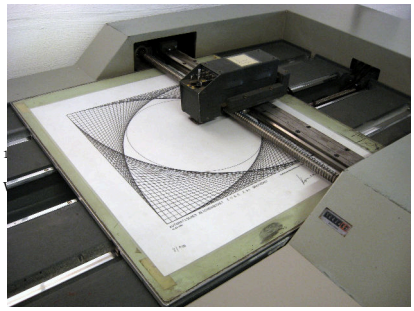


Figure 3.2: The ZUSE Plotter Z64 (presented in 1961). Source: `https://en.wikipedia.org/wiki/Plotter`.

Base R graphics were historically first: simple, procedural, canvas-oriented. There are specialized functions for different types of plots. These are easy to call – but when you want to combine them to build up more complex plots, or exchange one for another, this quickly gets messy to program, or even impossible. The user plots (the word stems back to some of the first graphics devices – see Figure 3.2) directly onto a (conceptual) canvas. She explicitly needs to deal with decisions such as how many inches to allocate to margins, axes labels, titles, legends, subpanels; once something is "plotted" it cannot be easily moved or erased.

There is a more high-level approach: in the **grammar of graphics**, graphics are built up from modular logical pieces, so that we can easily try different visualization types for our data in an intuitive and easily deciphered way, like we can switch in and out parts of a sentence in human language. There is no concept of a canvas or a plotter, rather, the user gives *ggplot2* a high-level description of the plot she wants, in the form of an R object, and the rendering engine takes a wholistic view on the scene to lay out the graphics and render them on the output device.

We'll explore **faceting**, for showing more than 2 variables at a time. Sometimes this is also called lattice[2] graphics, and it allows us to visualise data in up to four or five dimensions.

[2] The first major R package to implement this was *lattice*; nowadays much of such functionality is also provided through *ggplot2*.

3.1 Goals for this chapter

- Learn how to rapidly and flexibly explore datasets by visualization.
- Create beautiful and intuitive plots for scientific presentations and publications.
- Review the basics of *base* R plotting.
- Understand the logic behind the **grammar of graphics** concept.
- Introduce *ggplot2*'s `ggplot` function.
- See how to plot 1D, 2D, 3-5D data, and understand faceting.
- Creating "along-genome" plots for molecular biology data (or along other sequences, e. g., peptides).
- Discuss our options of interactive graphics.

3.2 Base R plotting

The most basic function is `plot`. In the code below, the output of which is shown in Figure 3.3, it is used to plot data from an enzyme-linked immunosorbent assay (ELISA) assay. The assay was used to quantify the activity of the enzyme deoxyribonuclease (DNase), which degrades DNA. The data are assembled in the R object `DNase`, which conveniently comes with base R. `DNase` is a dataframe whose columns are `Run`, the assay run; `conc`, the protein concentration that was used; and `density`, the measured optical density.

```
head(DNase)
```

```
##   Run       conc density
## 1   1 0.04882812   0.017
## 2   1 0.04882812   0.018
## 3   1 0.19531250   0.121
## 4   1 0.19531250   0.124
## 5   1 0.39062500   0.206
## 6   1 0.39062500   0.215
```

```
plot(DNase$conc, DNase$density)
```

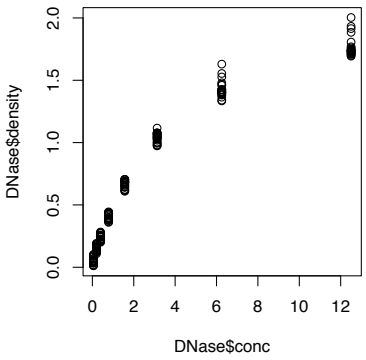


Figure 3.3: Plot of concentration vs. density for an ELISA assay of DNase.

This basic plot can be customized, for example by changing the plot symbol and axis labels using the parameters `xlab`, `ylab` and `pch` (plot character), as shown in Figure 3.4. Information about the variables is stored in the object `DNase`, and we can access it with the `attr` function.

```
plot(DNase$conc, DNase$density,
  ylab = attr(DNase, "labels")$y,
  xlab = paste(attr(DNase, "labels")$x, attr(DNase, "units")$x),
  pch = 3,
  col = "blue")
```

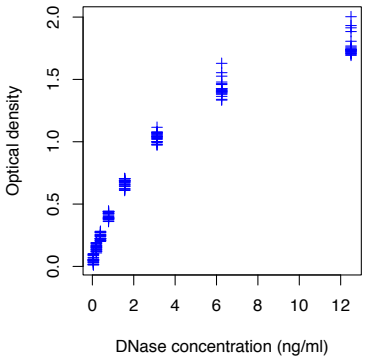


Figure 3.4: Same data as in Figure 3.3 but with better axis labels and a different plot symbol.

▶ Question **3.2.1.** Annotating dataframe columns with "metadata" such as longer descriptions, physical units, provenance information, etc., seems like a useful feature. Is this way of storing such information, as in the `DNase` object, standardized or common across the R ecosystem? Are there other standardized or common ways for doing this?

▶ Answer **3.2.1.** Have a look at the *DataFrame* class in the Bioconductor package *S4Vectors*. Among other things it is used to annotate the rows and columns of a *SummarizedExperiment*[3].

Besides scatterplots, we can also use built-in functions to create histograms and boxplots (Figure 3.5).

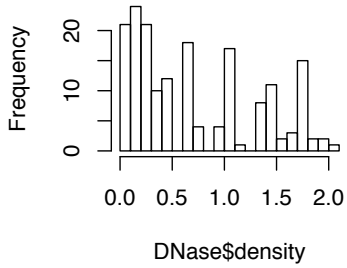
```
hist(DNase$density, breaks=25, main = "")
```

```
boxplot(density ~ Run, data = DNase)
```

Boxplots are convenient for showing multiple distributions next to each other in a compact space, and they are universally preferable to the barplots with error bars sometimes still seen in biological papers. We will see more about plotting multiple univariate distributions in Section 3.6.

The base R plotting functions are great for quick interactive exploration of data; but we run soon into their limitations if we want to create more sophisticated displays. We are going to use a visualization framework called the grammar of graphics, implemented in the package *ggplot2*, that enables step by step construction of high quality graphics in a logical and elegant manner. First let us introduce and load an example dataset.

[3] So the meatadata of the *DataFrame* that itself serves as metadata to the matrix in a *SummarizedExperiment* could be called metametadata. This recursion can be repeated ad infinitum. Some people dislike the "meta" prefix since it is more a subjective comment on rather than an objective property of a datum.



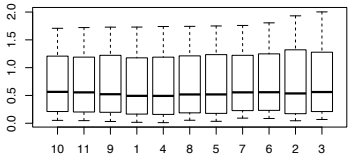


Figure 3.5: Histogram of the density from the ELISA assay, and boxplots of these values stratified by the assay run. The boxes are ordered along the axis in lexicographical order because the runs were stored as text strings. We could use R's type conversion functions to achieve numerical ordering.

3.3 An example dataset

To properly testdrive the *ggplot2* functionality, we are going to need a dataset that is big enough and has some complexity so that it can be sliced and viewed from many different angles. We'll use a gene expression microarray dataset that reports the transcriptomes of around 100 individual cells from mouse embryos at different time points in early development. The mammalian embryo starts out as a single cell, the fertilized egg. Through synchronized waves of cell divisions, the egg multiplies into a clump of cells that at first show no discernible differences between them. At some point, though, cells choose different lineages. By further and further specification, the different cell types and tissues arise that are needed for a full organism. The aim of the experiment, explained by Ohnishi et al. (2014), was to investigate the gene expression changes associated with the first symmetry breaking event in the embryo. We'll further explain the data as we go. More details can be found in the paper and in the documentation of the Bioconductor data package *Hiiragi2013*. We first load the package and the data:

```
library("Hiiragi2013")
data("x")
dim(exprs(x))
```

```
## [1] 45101   101
```

You can print out a more detailed summary of the *ExpressionSet* object `x` by just typing `x` at the R prompt. The 101 columns of the data matrix (accessed above through the `exprs` function) correspond to the samples (and each of these to a single cell), the 45101 rows correspond to the genes probed by the array, an Affymetrix mouse4302

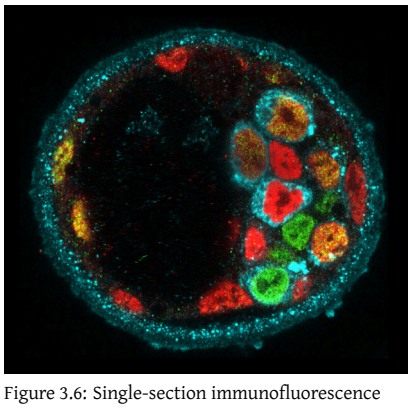


Figure 3.6: Single-section immunofluorescence image of the E3.5 mouse blastocyst stained for Serpinh1, a marker of primitive endoderm (blue), Gata6 (red) and Nanog (green).

array. The data were normalized using the RMA method (Irizarry et al., 2003). The raw data are also available in the package (in the data object `a`) and at EMBL-EBI's ArrayExpress database under the accession code E-MTAB-1681.

Let's have a look at what information is available about the samples.[4]

```
head(pData(x), n = 2)
```

```
##         File.name Embryonic.day Total.number.of.cells lineage
## 1 E3.25  1_C32_IN         E3.25                   32
## 2 E3.25  2_C32_IN         E3.25                   32
##         genotype   ScanDate sampleGroup sampleColour
## 1 E3.25       WT 2011-03-16       E3.25      #CAB2D6
## 2 E3.25       WT 2011-03-16       E3.25      #CAB2D6
```

The information provided is a mix of information about the cells (i.e., age, size and genotype of the embryo from which they were obtained) and technical information (scan date, raw data file name). By convention, time in the development of the mouse embryo is measured in days, and reported as, for instance, `E3.5`. Moreover, in the paper the authors divided the cells into 8 biological groups (`sampleGroup`), based on age, genotype and lineage, and they defined a color scheme to represent these groups (`sampleColour`[5]). Using the `group_by` and `summarise` functions from the package *dplyr*, we'll define a little dataframe `groups` that contains summary information for each group: the number of cells and the preferred color.

```
library("dplyr")
groups = group_by(pData(x), sampleGroup) %>%
  summarise(n = n() , color = unique(sampleColour))
groups
```

```
## # A tibble: 8 x 3
##       sampleGroup     n   color
##             <chr> <int>   <chr>
## 1           E3.25    36 #CAB2D6
## 2 E3.25 (FGF4-KO)    17 #FDBF6F
## 3      E3.5 (EPI)    11 #A6CEE3
## 4  E3.5 (FGF4-KO)     8 #FF7F00
## 5       E3.5 (PE)    11 #B2DF8A
## 6      E4.5 (EPI)     4 #1F78B4
## 7  E4.5 (FGF4-KO)    10 #E31A1C
## 8       E4.5 (PE)     4 #33A02C
```

The cells in the groups whose name contains `FGF4-KO` are from embryos in which the FGF4 gene, an important regulator of cell differentiation, was knocked out. Starting from E3.5, the wildtype cells (without the FGF4 knock-out) undergo the first symmetry breaking event and differentiate into different cell lineages, called pluripotent epiblast (EPI) and primitive endoderm (PE).

3.4 ggplot2

ggplot2 is a package by Hadley Wickham (Wickham, 2016) that implements the idea

[4] The notation #CAB2D6 is a hexadecimal representation of the RGB coordinates of a color; more on this in Section 3.10.2.

[5] This identifier in the dataset uses the British spelling. Everywhere else in this chapter, we use the US spelling (color). The *ggplot2* package generally accepts both spellings.

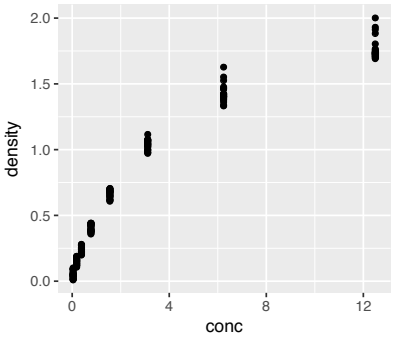


Figure 3.7: Our first *ggplot2* figure, similar to the base graphics Figure 3.3.

of **grammar of graphics** – a concept created by Leland Wilkinson in his eponymous book (Wilkinson, 2005). Comprehensive documentation for the package can be found on its website. The online documentation includes example use cases for each of the graphic types that are introduced in this chapter (and many more) and is an invaluable resource when creating figures.

Let's start by loading the package and redoing the simple plot of Figure 3.3.

```
library("ggplot2")
ggplot(DNase, aes(x = conc, y = density)) + geom_point()
```

We just wrote our first "sentence" using the grammar of graphics. Let us deconstruct this sentence. First, we specified the dataframe that contains the data, `DNase`. Then we told `ggplot` via the `aes`[6] argument which variables we want on the x- and y-axes, respectively. Finally, we stated that we want the plot to use points, by adding the result of calling the function `geom_point`.

Now let's turn to the mouse single cell data and plot the number of samples for each of the 8 groups using the `ggplot` function. The result is shown in Figure 3.8.

```
ggplot(data = groups, aes(x = sampleGroup, y = n)) +
  geom_bar(stat = "identity")
```

With `geom_bar` we now told `ggplot` that we want each data item (each row of `groups`) to be represented by a bar. Bars are one geometric object (**geom**) that `ggplot` knows about. We've already seen another geom in Figure 3.7: points. We'll encounter many other possible geometric objects later. We used the `aes` to indicate that we want the groups shown along the x-axis and the sizes along the y-axis. Finally, we provided the argument `stat = "identity"` (in other words, do nothing) to the `geom_bar` function, since otherwise it would try to compute a histogram of the data (the default value of `stat` is `"count"`). `stat` is short for **statistic**, which is what we call any function of data. Besides the identity and count statistic, there are others, such as smoothing, averaging, binning, or other operations that reduce the data in some way.

These concepts –data, geometrical objects, statistics– are some of the ingredients of the grammar of graphics, just as nouns, verbs and adverbs are ingredients of an English sentence.

▶ Question **3.4.1.**

Flip the x- and y-aesthetics to produce a horizontal barplot.

The plot in Figure 3.8 is not bad, but there are several potential improvements. We can use color for the bars to help us quickly see which bar corresponds to which group. This is particularly useful if we use the same color scheme in several plots. To this end, let's define a named vector `groupColor` that contains our desired colors for each possible value of `sampleGroup`.[7]

```
groupColor = setNames(groups$color, groups$sampleGroup)
```

Another thing that we need to fix is the readability of the bar labels. Right now

[6] This stands for **aesthetic**, a terminology that will become clearer below.

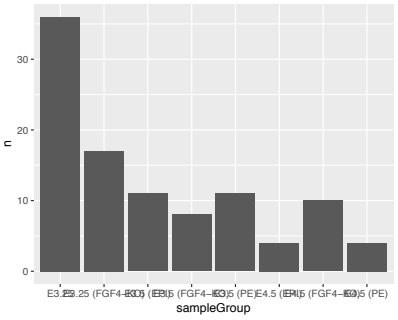


Figure 3.8: A barplot, produced with the `ggplot` function from the table of group sizes in the mouse single cell data.

[7] The information is completely equivalent to that in the `sampleGroup` and `color` columns of the dataframe `groups`, we're just adapting to the fact that *ggplot2* expects this information in the form of a named vector.

they are running into each other — a common problem when you have descriptive names.

```
ggplot(groups, aes(x = sampleGroup, y = n, fill = sampleGroup)) +
  geom_bar(stat = "identity") +
  scale_fill_manual(values = groupColor, name = "Groups") +
  theme(axis.text.x = element_text(angle = 90, hjust = 1))
```

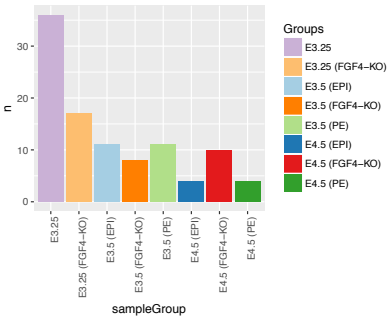


Figure 3.9: Similar to Figure 3.8, but with colored bars and better bar labels.

This is now already a longer and more complex sentence. Let us dissect it. We added an argument, `fill` to the `aes` function that states that we want the bars to be colored (filled) based on `sampleGroup` (which in this case co-incidentally is also the value of the `x` argument, but that need not be so). Furthermore we added a call to the `scale_fill_manual` function, which takes as its input a color map – i. e., the mapping from the possible values of a variable to the associated colors – as a named vector. We also gave this color map a title (note that in more complex plots, there can be several different color maps involved). Had we omitted the call to `scale_fill_manual`, *ggplot2* would have used its choice of default colors. We also added a call to `theme` stating that we want the *x*-axis labels rotated by 90 degrees and right-aligned (`hjust`; the default would be to center).

3.4.1 Data flow

ggplot2 expects your data in a dataframe[8]. If they are in a matrix, in separate vectors, or other types of objects, you will have to convert them. The packages *dplyr* and *broom*, among others, offer facilities to this end, we'll discuss this more in Section 14.11, and you'll see examples of such conversions sprinkled throughout the book.

[8] This includes the base R *data.frame* as well as the *tibble* (and synonymous *data_frame*) classes from the *tibble* package in the tidyverse.

The result of a call to the `ggplot` is a *ggplot* object. Let's recall a piece of code from above:

```
gg = ggplot(DNase, aes(x = conc, y = density)) + geom_point()
```

We have now assigned the output of `ggplot` to the object `gg`, instead of sending it directly to the console, where it was "printed" and produced Figure 3.7. The situation is completely analogously to what you are used to from working with the R console: when you enter an expression like `1+1` and hit "Enter", the result is printed. When the expression is an assignment, such as `s = 1+1`, the side effect takes place (the name `"s"` is bound to an object in memory that represents the value of `1+1`), but nothing is printed. Similarly, when an expression is evaluated as part of a script called with `source`, it is not printed. Thus, the above code also does not create any graphic output, since no `print` method is invoked. To print the `gg`, type its name (in an interactive session) or call `print` on it:

```
gg
print(gg)
```

3.4.2 Saving figures

ggplot2 has a built-in plot saving function called `ggsave`:

```
ggsave("DNAse-histogram-demo.pdf", plot = gg)
```

There are two major ways of storing plots: vector graphics and raster (pixel) graphics. In vector graphics, the plot is stored as a series of geometrical primitives such as points, lines, curves, shapes and typographic characters. The prefered format in R for saving plots into a vector graphics format is PDF. In raster graphics, the plot is stored in a dot matrix data structure. The main limitation of raster formats is their limited resolution, which depends on the number of pixels available. In R, the most commonly used device for raster graphics output is `png`. Generally, it's preferable to save your graphics in a vector graphics format, since it is always possible later to convert a vector graphics file into a raster format of any desired resolution, while the reverse is fundamentally limited by the resolution of the original file. And you don't want the figures in your talks or papers look poor because of pixelization artefacts!

3.5 The grammar of graphics

The components of *ggplot2*'s grammar of graphics are

1. one or more datasets,
2. one or more geometric objects that serve as the visual representations of the data, – for instance, points, lines, rectangles, contours,
3. descriptions of how the variables in the data are mapped to visual properties (aesthetics) of the geometric objects, and an associated scale (e. g., linear, logarithmic, rank),
4. one or more coordinate systems,
5. statistical summarization rules,
6. a facet specification, i. e. the use of multiple similar subplots to look at subsets of the same data,
7. optional parameters that affect the layout and rendering, such text size, font and alignment, legend positions, and the like.

In the examples above, Figures 3.8 and 3.9, the dataset was `groupsize`, the variables were the numeric values as well as the names of `groupsize`, which we mapped to the aesthetics y-axis and x-axis respectively, the scale was linear on the y and rank-based on the x-axis (the bars are ordered alphanumerically and each has the same width), the geometric object was the rectangular bar.

Items 4.–7. in the above list are optional. If you don't specify them, then the Cartesian is used as the coordinate system, the statistical summary is the trivial one (i. e., the identity), and no facets or subplots are made (we'll see examples later on, in Section 3.8). The first three items are mandatory, you always need to specify at least one of them: they are the minimal components of a valid *ggplot2* "sentence".

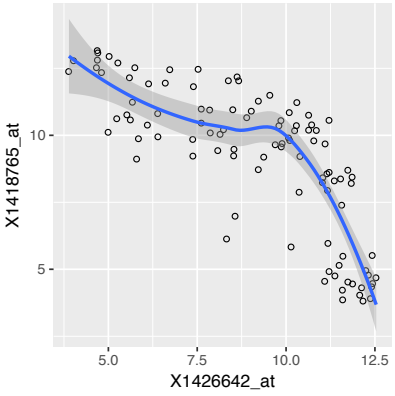


Figure 3.10: A scatterplot with three layers that show different statistics of the same data: points, a smooth regression line, and a confidence band.

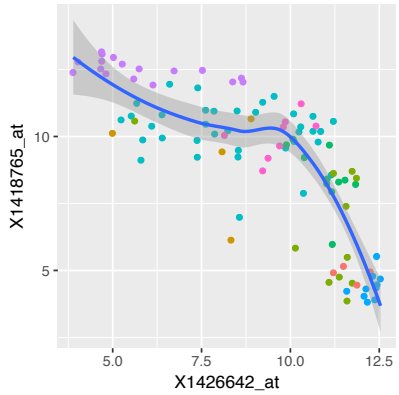


Figure 3.11: As Figure 3.10, but in addition with points colored by the sample group (as in Figure 3.9). We can now see that the expression values of the gene Timd2 (whose mRNA is targeted by the probe 1418765_at) are consistently high in the early time points, whereas its expression goes down in the EPI samples at days 3.5 and 4.5. In the FGF4-KO, this decrease is delayed - at E3.5, its expression is still high. Conversely, the gene Fn1 (1426642_at) is off in the early timepoints and then goes up at days 3.5 and 4.5. The PE samples (green) show a high degree of

In fact, *ggplot2*'s implementation of the grammar of graphics allows you to use the same type of component multiple times, in what are called layers (Wickham, 2010). For example, the code below uses three types of geometric objects in the same plot, for the same data: points, a line and a confidence band.

```
# dftx = as_tibble(t(exprs(x))) %>% cbind(pData(x))
dftx = data.frame(t(exprs(x)), pData(x))
ggplot( dftx, aes( x = X1426642_at, y = X1418765_at)) +
  geom_point( shape = 1 ) +
  geom_smooth( method = "loess" )
```

Here we had to assemble a copy of the expression data (`exprs(x)`) and the sample annotation data (`pData(x)`) all together into the dataframe `dftx` – since this is the data format that *ggplot2* functions most easily take as input (more on this in Section 14.11).

We can further enhance the plot by using colors – since each of the points in Figure 3.10 corresponds to one sample, it makes sense to use the `sampleColour` information in the object `x`.

```
ggplot( dftx, aes( x = X1426642_at, y = X1418765_at ))  +
  geom_point( aes( color = sampleColour), shape = 19 ) +
  geom_smooth( method = "loess" ) +
  scale_color_discrete( guide = FALSE )
```

▶ Question **3.5.1.** In the code above we defined the `color` aesthetics (`aes`) only for the `geom_point` layer, while we defined the `x` and `y` aesthetics for all layers. What happens if we set the `color` aesthetics for all layers, i. e., move it into the argument list of `ggplot`? What happens if we omit the call to `scale_color_discrete`?

▶ Question **3.5.2.** Is it always meaningful to visualize scatterplot data together with a regression line as in Figures 3.10 and 3.11?

As a small side remark, if we want to find out which genes are targeted by these probe identifiers, and what they might do, we can call.[9]

```
library("mouse4302.db")
```

```
AnnotationDbi::select(mouse4302.db,
   keys = c("1426642_at", "1418765_at"), keytype = "PROBEID",
   columns = c("SYMBOL", "GENENAME"))
```

```
##      PROBEID SYMBOL
## 1 1426642_at    Fn1
## 2 1418765_at  Timd2
##                                          GENENAME
## 1                                     fibronectin 1
## 2 T cell immunoglobulin and mucin domain containing 2
```

Often when using `ggplot` you will only need to specify the data, aesthetics and a geometric object. Most geometric objects implicitly call a suitable default statistical summary of the data. For example, if you are using `geom_smooth`, *ggplot2* by default uses `stat = "smooth"` and then displays a line; if you use `geom_histogram`, the

[9] Note that here were need to use the original feature identifiers (e. g., "1426642_at", without the leading "X"). This is the notation used by the microarray manufacturer, by the Bioconductor annotation packages, and also inside the object `x`. The leading "X" that we used above when working with `dftx` was inserted during the creation of `dftx` by the constructor fuction `data.frame`, since its argument `check.names` is set to `TRUE` by default. Alternatively, we could have kept the original identifer notation by setting `check.names=FALSE`, but then we would need to work with the backticks, such as `aes( x = `1426642_at`, ...)`, to make sure R understands the identifiers correctly.

data are binned, and the result is displayed in barplot format. Here's an example:

```
dfx = as.data.frame(exprs(x))
ggplot(dfx, aes(x = '20 E3.25')) + geom_histogram(binwidth = 0.2)
```

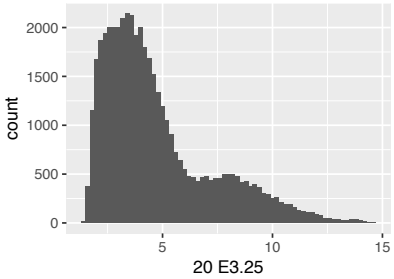


Figure 3.12: Histogram of probe intensities for one particular sample, cell number 20, which was from day E3.25.

▶ Question **3.5.3.** What is the difference between the objects `dfx` and `dftx`? Why did we need to create both of the?

Let's come back to the barplot example from above.

```
pb = ggplot(groups, aes(x = sampleGroup, y = n))
```

This creates a plot object `pb`. If we try to display it, it creates an empty plot, because we haven't specified what geometric object we want to use. All that we have in our `pb` object so far are the data and the aesthetics (Fig. 3.13)

```
class(pb)

## [1] "gg"     "ggplot"

pb
```

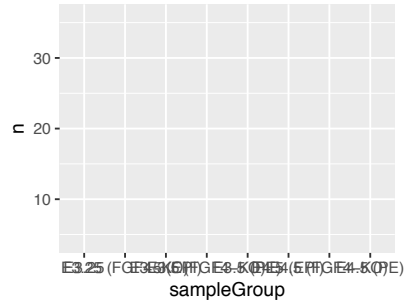


Figure 3.13: `pb`: without a geometric object, the plot remains empty.

Now we can literally add on the other components of our plot through using the + operator (Fig. 3.14):

```
pb = pb + geom_bar(stat = "identity")
pb = pb + aes(fill = sampleGroup)
pb = pb + theme(axis.text.x = element_text(angle = 90, hjust = 1))
pb = pb + scale_fill_manual(values = groupColor, name = "Groups")
pb
```

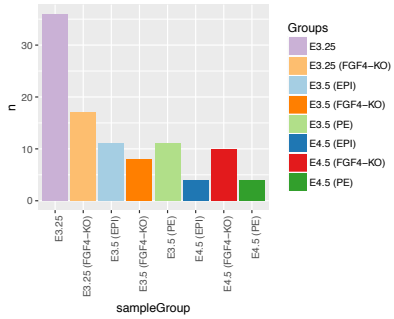


Figure 3.14: The graphics object `bp` in its full glory.

This step-wise buildup –taking a graphics object already produced in some way and then further refining it– can be more convenient and easy to manage than, say, providing all the instructions upfront to the single function call that creates the graphic. We can quickly try out different visualization ideas without having to rebuild our plots each time from scratch, but rather store the partially finished object and then modify it in different ways. For example we can switch our plot to polar coordinates to create an alternative visualization of the barplot.

```
pb.polar = pb + coord_polar() +
  theme(axis.text.x = element_text(angle = 0, hjust = 1),
        axis.text.y = element_blank(),
        axis.ticks = element_blank()) +
  xlab("") + ylab("")
pb.polar
```

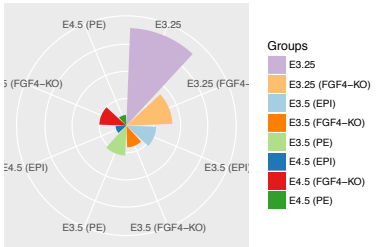


Figure 3.15: A barplot in a polar coordinate system.

Note above that we can override previously set `theme` parameters by simply setting them to a new value – no need to go back to recreating `pb`, where we originally set them.

3.6 Visualization of 1D data

A common task in biological data analysis is the comparison between several samples of univariate measurements. In this section we'll explore some possibilities for visual-

izing and comparing such samples. As an example, we'll use the intensities of a set of four genes Fgf4, Gata4, Gata6 and Sox2[10]. On the array, they are represented by

```
selectedProbes = c( Fgf4 = "1420085_at", Gata4 = "1418863_at",
                    Gata6 = "1425463_at",  Sox2 = "1416967_at")
```

[10] You can read more about these genes in the paper associated with the data.

To extract data from this representation and convert them into a dataframe, we use the function `melt` from the *reshape2* package[11].

```
library("reshape2")
genes = melt(exprs(x)[selectedProbes, ], varnames = c("probe", "sample"))
head(genes)
```

```
##        probe  sample    value
## 1 1420085_at 1 E3.25 3.027715
## 2 1418863_at 1 E3.25 4.843137
## 3 1425463_at 1 E3.25 5.500618
## 4 1416967_at 1 E3.25 1.731217
## 5 1420085_at 2 E3.25 9.293016
## 6 1418863_at 2 E3.25 5.530016
```

[11] We'll talk more about the concepts and mechanics of different data representations in Section 14.11.

For good measure, we also add a column that provides the gene symbol along with the probe identifiers.

```
genes$gene = names(selectedProbes)[ match(genes$probe, selectedProbes) ]
```

3.6.1 Barplots

A popular way to display data such as in our dataframe `genes` is through barplots. See Fig. 3.16.

```
ggplot(genes, aes( x = gene, y = value)) +
  stat_summary(fun.y = mean, geom = "bar")
```

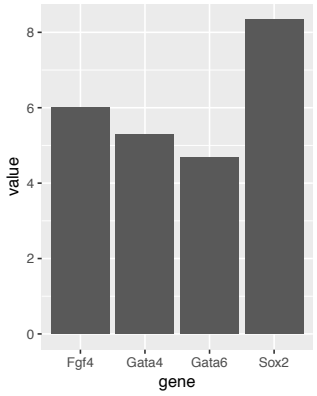


Figure 3.16: Barplots showing the means of the distributions of expression measurements from 4 probes.

In Figure 3.16, each bar represents the mean of the values for that gene. Such plots are seen a lot in the biological sciences, as well as in the popular media. The data summarization into only the mean looses a lot of information, and given the amount of space it takes, a barplot can be a poor way to visualise data[12].

[12] In fact, if the mean is not an appropriate summary, such as for highly skewed distributions, or datasets with outliers, the barplot can be outright misleading.

Sometimes we want to add error bars, and one way to achieve this in *ggplot2* is as follows.

```
library("Hmisc")
ggplot(genes, aes( x = gene, y = value, fill = gene)) +
  stat_summary(fun.y = mean, geom = "bar") +
  stat_summary(fun.data = mean_cl_normal, geom = "errorbar",
               width = 0.25)
```

Here, we see again the principle of layered graphics: we use two summary functions, `mean` and `mean_cl_normal`, and two associated geometric objects, `bar` and `errorbar`. The function `mean_cl_normal` is from the *Hmisc* package and computes the standard error (or confidence limits) of the mean; it's a simple function, and we

could also compute it ourselves using base R expressions if we wished to do so. We also colored the bars to make the plot more pleasant.

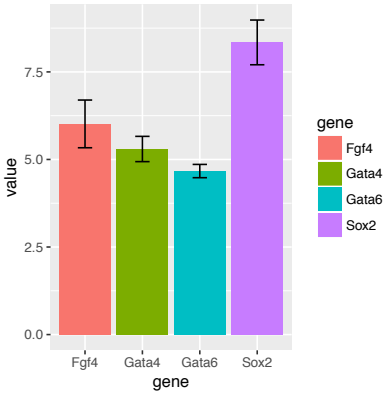


Figure 3.17: Barplots with error bars indicating standard error of the mean.

3.6.2 Boxplots

It's easy to show the same data with boxplots.

```
p = ggplot(genes, aes( x = gene, y = value, fill = gene))
p + geom_boxplot()
```

Compared to the barplots, this takes a similar amount of space, but is much more informative. In Figure 3.18 we see that two of the genes (Gata4, Gata6) have relatively concentrated distributions, with only a few data points venturing out to the direction of higher values. For Fgf4, we see that the distribution is right-skewed: the median, indicated by the horizontal black bar within the box is closer to the lower (or left) side of the box. Analogously, for Sox2 the distribution is left-skewed.

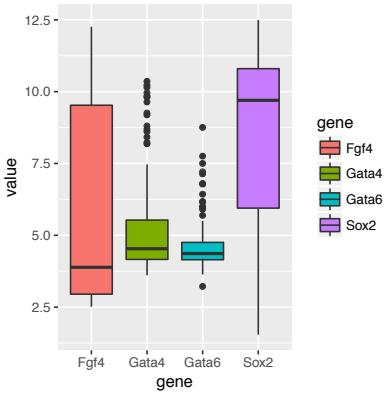


Figure 3.18: Boxplots.

3.6.3 Violin plots

A variation of the boxplot idea, but with an even more direct representation of the shape of the data distribution, is the violin plot. Here, the shape of the violin gives a rough impression of the distribution density.

```
p + geom_violin()
```

3.6.4 Dot plots and beeswarm plots

If the number of data points is not too large, it is possible to show the data points directly, and it is good practice to do so, compared to using more abstract summaries. However, plotting the data directly will often lead to overlapping points, which can be visually unpleasant, or even obscure the data. We can try to layout the points so that they are as near possible to their proper locations without overlap (Wilkinson, 1999).

```
p + geom_dotplot(binaxis = "y", binwidth = 1/6,
       stackdir = "center", stackratio = 0.75,
       aes(color = gene))
```

The plot is shown in the left panel of Figure 3.20. The y-coordinates of the points are discretized into bins (above we chose a bin size of 1/6), and then they are stacked next to each other.

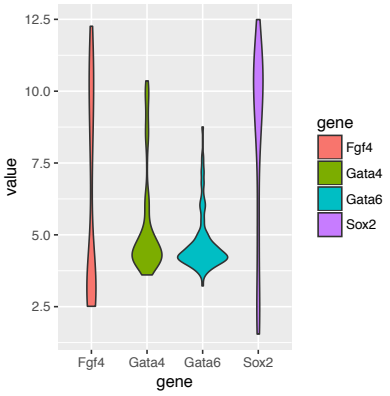


Figure 3.19: Violin plots.

An alternative is provided by the package *ggbeeswarm*, which provides `geom_beeswarm`.

```
library("ggbeeswarm")
p + geom_beeswarm(aes(color = gene))
```

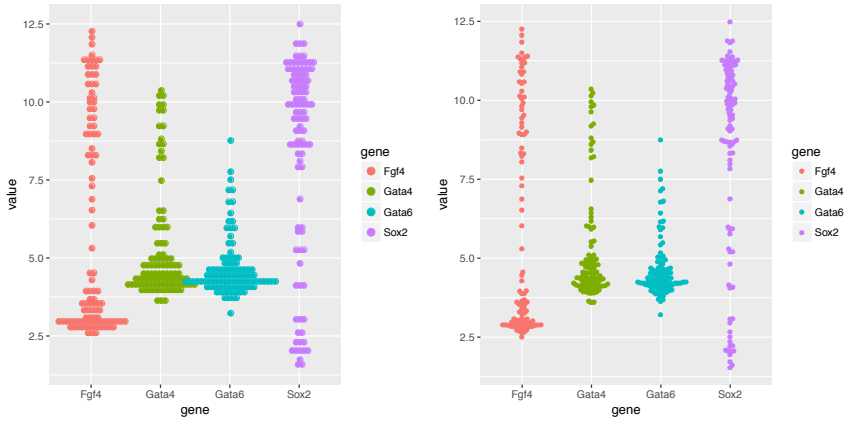


Figure 3.20: Left: dot plots, made using `geom_dotplot` from *ggplot2*. Right: beeswarm plots, made using `geom_beeswarm` from *ggbeeswarm*.

The plot is shown in the right panel of Figure 3.20. The layout algorithm tries to avoid overlaps between the points. If a point were to overlap an existing point, it is shifted sideways (along the x-axis) by a small amount sufficient to avoid overlap. Some twiddling with layout parameters is usually needed for each new dataset to make a dot plot or a beeswarm plot look good.

3.6.5 Density plots

Yet another way to represent the same data is by density plots (Figure 3.21).

```
ggplot(genes, aes( x = value, color = gene)) + geom_density()
```

Density estimation has a number of complications, in particular, the need for choosing a smoothing window. A window size that is small enough to capture peaks in the dense regions of the data may lead to instable ("wiggly") estimates elsewhere. On the other hand, if the window is made bigger, pronounced features of the density, such as sharp peaks, may be smoothed out. Moreover, the density lines do not convey the information on how much data was used to estimate them, and plots like Figure 3.21 can be especially problematic if the sample sizes for the curves differ.

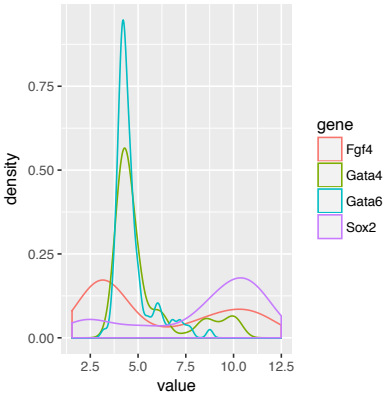


Figure 3.21: Density plots.

3.6.6 ECDF plots

The mathematically most convenient way to describe the distribution of a one-dimensional random variable X is its cumulative distribution function (CDF), i. e., the function

$$F(x) = P(X \leq x), \tag{3.1}$$

where x takes all values along the real axis. The density of X is then the derivative of F, if it exists[13]. The definition of the CDF can also be applied to finite samples of X, i. e., samples $x_1, \ldots, x_n$. The empirical cumulative distribution function (ECDF) is

[13] By its definition, F tends to 0 for small x ($x \to -\infty$) and to 1 for large x ($x \to +\infty$).

simply (Figure 3.22):

$$F_n(x) = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}_{x \leq x_i}. \tag{3.2}$$

```
ggplot(genes, aes( x = value, color = gene)) + stat_ecdf()
```

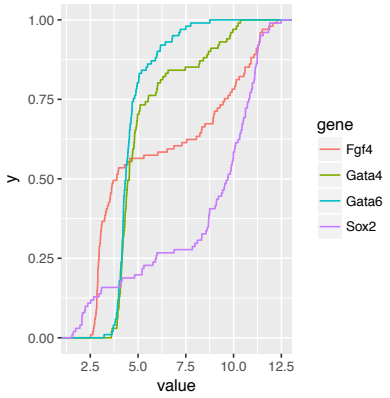


Figure 3.22: Empirical cumulative distribution functions (ECDF).

The ECDF has several nice properties:

- It is lossless - the ECDF $F_n(x)$ contains all the information contained in the original sample $x_1, \ldots, x_n$ (except the –unimportant– order of the values).
- As n grows, the ECDF $F_n(x)$ converges to the true CDF $F(x)$. Even for limited sample sizes n, the difference between the two functions tends to be small. Note that this is not the case for the empirical density! Without smoothing, the empirical density of a finite sample is a sum of Dirac delta functions, which is difficult to visualize and quite different from any underlying smooth, true density. With smoothing, the difference can be less pronounced, but is difficult to control, as we discussed above.

3.6.7 The effect of transformations on densities

It is tempting to look at histograms or density plots and inspect them for evidence of bimodality (or multimodality) as an indication of some underlying biological phenomenon. Before doing so, it is important to remember that the number of modes of a density depends on scale transformations of the data, via the **chain rule**. A simple example, with a mixture of two normal distributions, is shown in Figure 3.23.

```
sim <- data_frame(
   x = exp(rnorm(
     n    = 1e5,
     mean = sample(c(2, 5), size = 1e5, replace = TRUE))))

ggplot(sim, aes(x)) +
    geom_histogram(binwidth = 10, boundary = 0) + xlim(0, 400)
ggplot(sim, aes(log(x))) + geom_histogram(bins = 30)
```

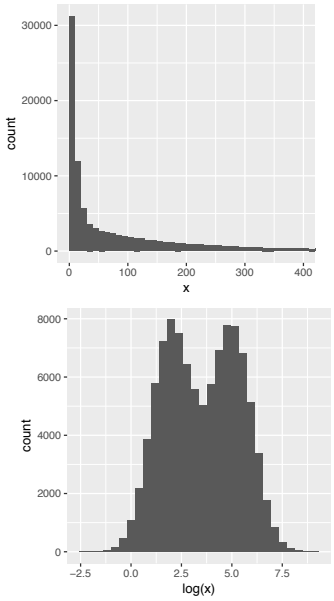


Figure 3.23: Histograms of the same data, with and without logarithmic transformation. The number of modes is different.

▶ Question **3.6.1.** Consider a log-normal mixture model as in the code above. What is the density function of X? What is the density function of $log(X)$? How many modes do these densities have, as a function of the parameters of the mixture model (mean and standard deviation of the component normals, and mixture fraction)?

3.7 Visualization of 2D data: scatterplots

Scatterplots are useful for visualizing treatment–response comparisons (as in Figure 3.4), associations between variables (as in Figure 3.11), or paired data (e. g., a disease biomarker in several patients before and after treatment). We use the two

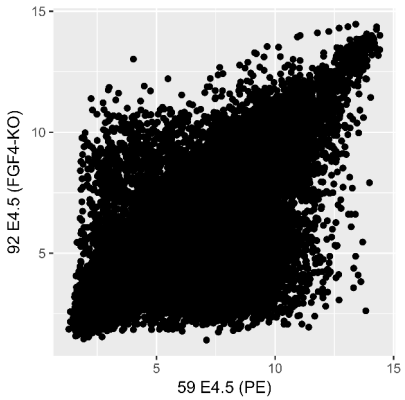


Figure 3.24: Scatterplot of 45101 expression measurements for two of the samples.

dimensions of our plotting paper, or screen, to represent the two variables. Let us take a look at differential expression between a wildtype and an FGF4-KO sample.

```
scp = ggplot(dfx, aes(x = `59 E4.5 (PE)` ,
                      y = `92 E4.5 (FGF4-KO)`))
scp + geom_point()
```

The labels `59 E4.5 (PE)` and `92 E4.5 (FGF4-KO)` refer to column names (sample names) in the dataframe `dfx`, which we created above. Since they contain special characters (spaces, parentheses, hyphen) and start with numerals, we need to enclose them with the downward sloping quotes to make them syntactically digestible for R. The plot is shown in Figure 3.24. We get a dense point cloud that we can try and interpret on the outskirts of the cloud, but we really have no idea visually how the data are distributed within the denser regions of the plot.

One easy way to ameliorate the overplotting is to adjust the transparency (alpha value) of the points by modifying the `alpha` parameter of `geom_point` (Figure 3.25).

```
scp  + geom_point(alpha = 0.1)
```

This is already better than Figure 3.24, but in the more dense regions even the semi-transparent points quickly overplot to a featureless black mass, while the more isolated, outlying points are getting faint. An alternative is a contour plot of the 2D density, which has the added benefit of not rendering all of the points on the plot, as in Figure 3.26.

```
scp + geom_density2d()
```

However, we see in Figure 3.26 that the point cloud at the bottom right (which contains a relatively small number of points) is no longer represented. We can somewhat overcome this by tweaking the bandwidth and binning parameters of `geom_density2d` (Figure 3.27, left panel).

```
scp + geom_density2d(h = 0.5, bins = 60)
```

We can fill in each space between the contour lines with the relative density of points by explicitly calling the function `stat_density2d` (for which `geom_density2d` is a wrapper) and using the geometric object **polygon**, as in the right panel of Figure 3.27.

```
library("RColorBrewer")
colorscale = scale_fill_gradientn(
    colors = rev(brewer.pal(9, "YlGnBu")),
    values = c(0, exp(seq(-5, 0, length.out = 100))))

scp + stat_density2d(h = 0.5, bins = 60,
          aes( fill = ..level..), geom = "polygon") +
  colorscale + coord_fixed()
```

We used the function `brewer.pal` from the package *RColorBrewer* to define the color scale, and we added a call to `coord_fixed` to fix the aspect ratio of the plot, to make sure that the mapping of data range to x- and y-coordinates is the same for the two variables. Both of these issues merit a deeper look, and we'll talk more about plot

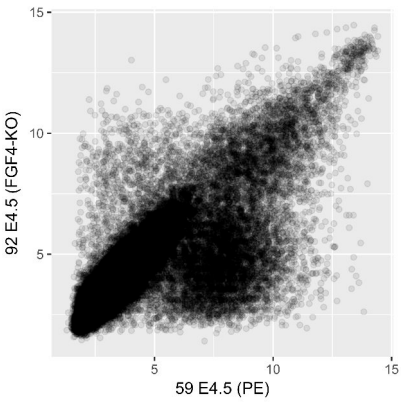


Figure 3.25: As Figure 3.24, but with semi-transparent points to resolve some of the over-plotting.

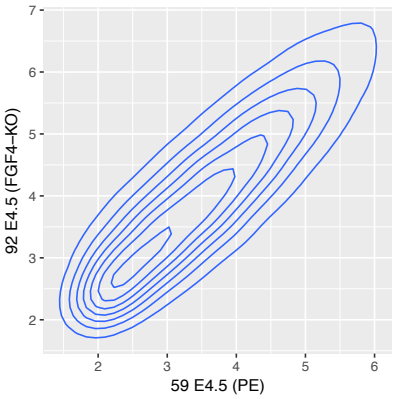


Figure 3.26: As Figure 3.24, but rendered as a contour plot of the 2D density estimate.

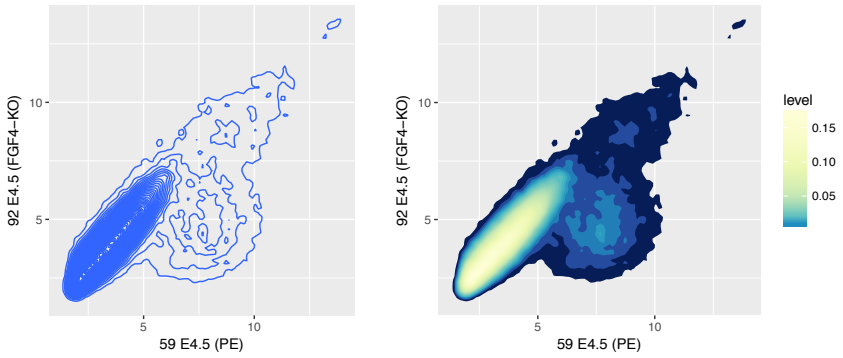


Figure 3.27: Left: as Figure 3.26, but with smaller smoothing bandwidth and tighter binning for the contour lines. Right: with color filling.

shapes in Section 3.7.1 and about colors in Section 3.9.

The density based plotting methods in Figure 3.27 are more visually appealing and interpretable than the overplotted point clouds of Figures 3.24 and 3.25, though we have to be careful in using them as we loose a lot of the information on the outlier points in the sparser regions of the plot. One possibility is using `geom_point` to add such points back in.

But arguably the best alternative, which avoids the limitations of smoothing, is hexagonal binning (Carr et al., 1987).

```
scp + geom_hex() + coord_fixed()
scp + geom_hex(binwidth = c(0.2, 0.2)) + colorscale +
  coord_fixed()
```

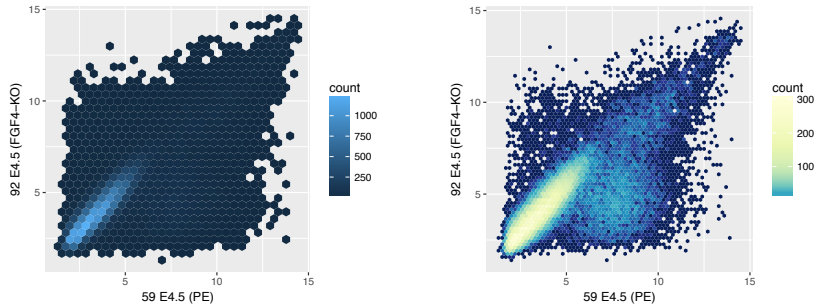


Figure 3.28: Hexagonal binning. Left: default parameters. Right: finer bin sizes and customized color scale.

3.7.1 Plot shapes

Choosing the proper shape for your plot is important to make sure the information is conveyed well. By default, the shape parameter, that is, the ratio, between the height of the graph and its width, is chosen by *ggplot2* based on the available space in the current plotting device. The width and height of the device are specified when it is opened in R, either explicitly by you or through default parameters[14]. Moreover, the

[14] E. g., see the manual pages of the `pdf` and `png` functions.

graph dimensions also depend on the presence or absence of additional decorations, like the color scale bars in Figure 3.28.

There are two simple rules that you can apply for scatterplots:

- If the variables on the two axes are measured in the same units, then make sure that the same mapping of data space to physical space is used – i. e., use `coord_fixed`. In the scatterplots above, both axes are the logarithm to base 2 of expression level measurements, that is a change by one unit has the same meaning on both axes (a doubling of the expression level). Another case is principal component analysis (PCA), where the x-axis typically represents component 1, and the y-axis component 2. Since the axes arise from an orthonormal rotation of input data space, we want to make sure their scales match. Since the variance of the data is (by definition) smaller along the second component than along the first component (or at most, equal), well-done PCA plots usually have a width that's larger than the height.
- If the variables on the two axes are measured in different units, then we can still relate them to each other by comparing their dimensions. The default in many plotting routines in R, including *ggplot2*, is to look at the range of the data and map it to the available plotting region. However, in particular when the data more or less follow a line, looking at the typical slope of the line can be useful. This is called banking (Cleveland et al., 1988).

To illustrate banking, let's use the classic sunspot data from Cleveland's paper.

```
library("ggthemes")
sunsp = tibble(year   = time(sunspot.year),
               number = as.numeric(sunspot.year))
sp = ggplot(sunsp, aes(x = year, y = number)) + geom_line()
sp
```

The resulting plot is shown in the upper panel of Figure 3.29. We can clearly see long-term fluctuations in the amplitude of sunspot activity cycles, with particularly low maximum activities in the early 1700s, early 1800s, and around the turn of the 20th century. But now lets try out banking.

```
ratio = with(sunsp, bank_slopes(year, number))
sp + coord_fixed(ratio = ratio)
```

What the algorithm does is to look at the slopes in the curve, and in particular, the above call to `bank_slopes` computes the median absolute slope, and then with the call to `coord_fixed` we shape the plot such that this quantity becomes 1. The result is shown in the lower panel of Figure 3.29. Quite counter-intuitively, even though the plot takes much smaller space, we see more on it! Namely, we can see the saw-tooth shape of the sunspot cycles, with sharp rises and more slow declines.

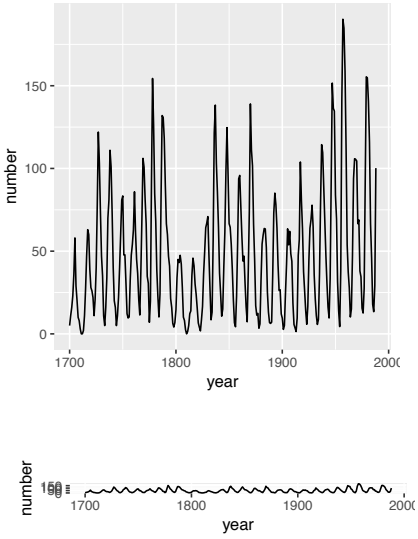


Figure 3.29: The sunspot data. In the upper panel, the plot shape is roughly quadratic, a frequent default choice. In the lower panel, a technique called **banking** was used to choose the plot shape.

3.8 3–5D data

Sometimes we want to show the relations between more than two variables. Obvious choices for including additional dimensions are the plot symbol shapes and colors. The `geom_point` geometric object offers the following aesthetics (beyond `x` and `y`):

- `fill`
- `color`
- `shape`
- `size`
- `alpha`

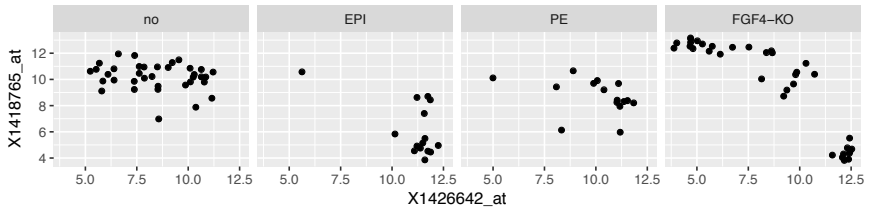
They are explored in the manual page of the `geom_point` function. `fill` and `color` refer to the fill and outline color of an object, `alpha` to its transparency level. Above, in Figures 3.25 and following, we have used color or transparency to reflect point density and avoid the obscuring effects of overplotting. Instead, we can use them show other dimensions of the data (but of course we can only do one or the other). In principle, we could use all the 5 aesthetics listed above simultaneously to show up to 7-dimensional data; however, such a plot would be hard to decipher, and usually we are better off with sticking to at most one or two additional dimensions and mapping them to a choice of the available aesthetics.

3.8.1 Faceting

Another way to show additional dimensions of the data is to show multiple plots that result from repeatedly subsetting (or "slicing") our data based on one (or more) of the variables, so that we can visualize each part separately. So we can, for instance, investigate whether the observed patterns among the other variables are the same or different across the range of the faceting variable. Let's look at an example[15]

```
library("magrittr")
dftx$lineage %<>% sub("^$", "no", .)
dftx$lineage %<>% factor(levels = c("no", "EPI", "PE", "FGF4-KO"))

ggplot(dftx, aes( x = X1426642_at, y = X1418765_at)) +
  geom_point() + facet_grid( . ~ lineage )
```



[15] The first two lines this code chunk are not strictly necessary – they're just reformatting the `lineage` column of the `dftx` dataframe, to make the plots look better.

Figure 3.30: An example for **faceting**: the same data as in Figure 3.10, but now split by the categorical variable `lineage`.

The result is shown in Figure 3.30. We used the formula language to specify by which variable we want to do the splitting, and that the separate panels should be in

different columns: `facet_grid( . ~ lineage )`. In fact, we can specify two faceting variables, as follows; the result is shown in Figure 3.31.

```
ggplot( dftx,
  aes( x = X1426642_at, y = X1418765_at)) + geom_point() +
   facet_grid( Embryonic.day ~ lineage )
```

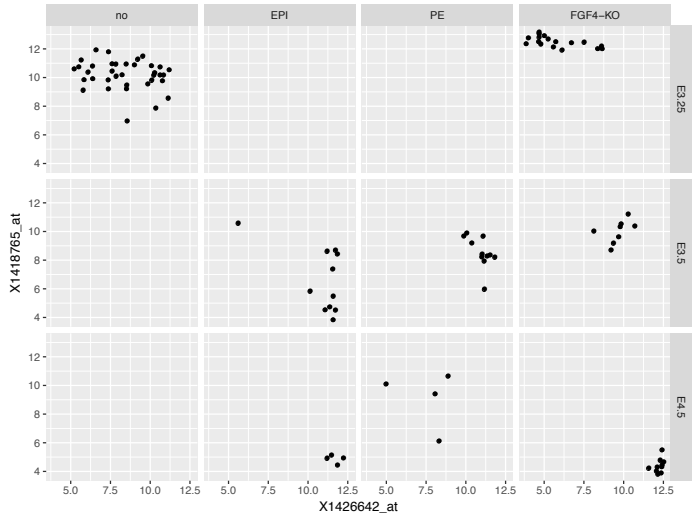


Figure 3.31: **Faceting**: the same data as in Figure 3.10, split by the categorical variables `Embryonic.day` (rows) and `lineage` (columns).

Another useful function is `facet_wrap`: if the faceting variable has too many levels for all the plots to fit in one row or one column, then this function can be used to wrap them into a specified number of columns or rows.

We can use a continuous variable by discretizing it into levels. The function `cut` is useful for this purpose.

```
ggplot(mutate(dftx, Tdgf1 = cut(X1450989_at, breaks = 4)),
   aes( x = X1426642_at, y = X1418765_at)) + geom_point() +
   facet_wrap( ~ Tdgf1, ncol = 2 )
```

We see in Figure 3.32 that the number of points in the four panel is different, this is because `cut` splits into bins of equal length, not equal number of points. If we want the latter, then we can use `quantile` in conjunction with `cut`.

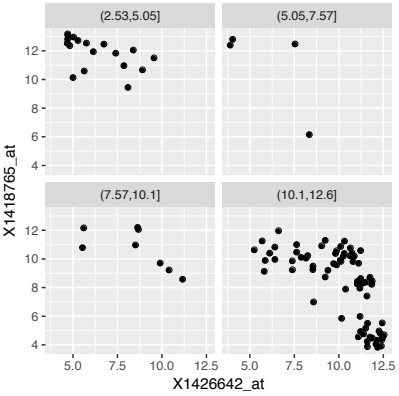


Figure 3.32: **Faceting**: the same data as in Figure 3.10, split by the continuous variable `X1450989_at` and arranged by `facet_wrap`.

Axes scales In Figures 3.30–3.32, the axes scales are the same for all plots. Alternatively, we could let them vary by setting the `scales` argument of the `facet_grid` and `facet_wrap`; this parameters allows you to control whether to leave the x-axis, the y-axis, or both to be freely variable. Such alternatives scalings might allows us to see the full detail of each plot and thus make more minute observations about what is going on in each. The downside is that the plot dimensions are not comparable across the groupings.

Implicit faceting You can also facet your plots (without explicit calls to `facet_grid` and `facet_wrap`) by specifying the aesthetics. A very simple version of implicit faceting is using a factor as your x-axis, such as in Figures 3.16–3.20

3.8.2 Interactive graphics

The plots generated thus far have been static images. You can add an enormous amount of information and expressivity by making your plots interactive. It is impossible here for us to convey interactive visualizations, but we provide pointers to some important resources.

shiny

Rstudio's *shiny* is a web application framework for R. It makes it easy to create interactive displays with sliders, selectors and other control elements that allow changing all aspects of the plot(s) shown – since the interactive elements call back directly into the R code that produces the plot(s). See the shiny gallery for some great examples.

ggvis

ggvis is a graphics system for R that builds upon *shiny* and has an underlying theory, look-and-feel and programming interface similar to *ggplot2*.

As a consequence of interactivity in *shiny* and *ggvis*, there needs to be an R interpreter running with the underlying data and code to respond to the user's actions while she views the graphic. This R interpreter could be on the local machine, or on a server; in both cases, the viewing application is a web browser, and the interaction with R goes through web protocols (http, or https). That is, of course, different from a regular static graphic in a PDF file or in an HTML report, which is produced once and can then be viewed without any

plotly

A great web-based tool for interactive graphic generation is **plotly**. You can view some examples of interactive graphics online at `https://plot.ly/r`. To create your own interactive plots in R, you can use code like

```
library("plotly")
plot_ly(economics, x = date, y = unemploy / pop)
```

Like with *shiny* and *ggvis*, the graphics are viewed in an HTML browser, however, no running R session is required. The graphics are self-contained HTML documents whose "logic" is coded in JavaScript, or more precisely, in the D3.js system.

rgl, webgl

For visualising 3D objects (say, a geometrical structure), there is the package *rgl*. It produces interactive viewer windows (either in specialized graphics device on your screen, or through a web browser) in which you can rotate the scene, zoom and in out, etc. An screenshot of the scene produced by the code below is shown in Figure 3.33.

```
data("volcano")
volcanoData = list(
  x = 10 * seq_len(nrow(volcano)),
  y = 10 * seq_len(ncol(volcano)),
  z = volcano,
  col = terrain.colors(500)[cut(volcano, breaks = 500)]
)
library("rgl")
with(volcanoData, persp3d(x, y, z, color = col))
```

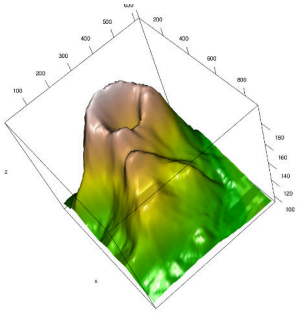


Figure 3.33: *rgl* rendering of the `volcano` data, the topographic information for Maunga Whau (Mt Eden), one of about 50 volcanos in the Auckland volcanic field.

In the code above, the base R function `cut` computes a mapping from the value range of the `volcano` data to the integers between 1 and 500[16], which we use to index the color scale, `terrain.colors(500)`.

[16] More precisely, it returns a factor with as many levels, which we let R autoconvert to integers.

For more information, consult the package's excellent vignette.

3.9 Color

An important consideration when making plots is the coloring that we use in them. Most R users are likely familiar with the built-in R color scheme, used by base R graphics, as shown in Figure 3.34.

```
pie(rep(1, 8), col=1:8)
```

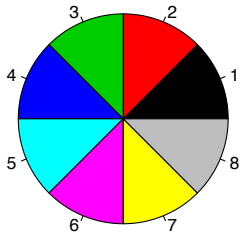


Figure 3.34: Basic R colors.

These color choices date back from 1980s hardware, where graphics cards handled colors by letting each pixel either fully use or not use each of the three basic color channels of the display: red, green and blue (RGB): this leads to $2^3 = 8$ combinations, which lie at the 8 the extreme corners of the RGB color cube[17]. The colors in Figure 3.34 are harsh on the eyes, and there is no good excuse any more for creating graphics that are based on this palette. Fortunately, the default colors used by some of the more modern visualization oriented packages (including *ggplot2*) are much better already, but sometimes we want to make our own choices.

[17] Thus the 8th color should be white; in R, this was replaced by grey, as you can see in Figure 3.34.

In Section 3.7 we saw the function `scale_fill_gradientn`, which allowed us to create the color gradient used in Figures 3.27 and 3.28 by interpolating the basic color palette defined by the function `brewer.pal` in the *RColorBrewer* package. This package defines a great set of color palettes. We can see all of them at a glance by using the function `display.brewer.all` (Figure 3.35).

```
display.brewer.all()
```

We can get information about the available color palettes from `brewer.pal.info`.

```
head(brewer.pal.info)
```

```
##      maxcolors category colorblind
## BrBG        11      div       TRUE
## PiYG        11      div       TRUE
## PRGn        11      div       TRUE
## PuOr        11      div       TRUE
## RdBu        11      div       TRUE
## RdGy        11      div      FALSE
```

```
table(brewer.pal.info$category)
```

```
##
##  div qual  seq
##    9    8   18
```

The palettes are divided into three categories:

- qualitative: for categorical properties that have no intrinsic ordering. The `Paired` palette supports up to 6 categories that each fall into two subcategories - like **before** and **after**, **with** and **without** treatment, etc.
- sequential: for quantitative properties that go from **low** to **high**
- diverging: for quantitative properties for which there is a natural midpoint or neutral value, and whose value can deviate both up- and down; we'll see an example in Figure 3.37.

To obtain the colors from a particular palette we use the function `brewer.pal`. Its first argument is the number of colors we want (which can be less than the available maximum number in `brewer.pal.info`).

```
brewer.pal(4, "RdYlGn")
```

```
## [1] "#D7191C" "#FDAE61" "#A6D96A" "#1A9641"
```

If we want more than the available number of preset colors (for example so we can plot a heatmap with continuous colors) we can interpolate using the `colorRampPalette` function[18].

```
mypalette  = colorRampPalette(c("darkorange3", "white", "darkblue"))(100)
head(mypalette)
```

```
## [1] "#CD6600" "#CE6905" "#CF6C0A" "#D06F0F" "#D17214" "#D27519"
```

```
par(mai = rep(0.1, 4))
image(matrix(1:100, nrow = 100, ncol = 10), col = mypalette,
        xaxt = "n", yaxt = "n", useRaster = TRUE)
```

[18] `colorRampPalette` returns a function of one parameter, an integer. In the code shown, we call that function with the argument `100`.

Figure 3.36: A quasi-continuous color palette derived by interpolating between the colors `darkorange3`, `white` and `darkblue`.

3.10 Heatmaps

Heatmaps are a powerful of visualising large, matrix-like datasets and giving a quick overview over the patterns that might be in there. There are a number of heatmap drawing functions in R; one that is convenient and produces good-looking output is the function `pheatmap` from the eponymous package[19]. In the code below, we first select the top 500 most variable genes in the dataset `x` and define a function `rowCenter` that centers each gene (row) by subtracting the mean across columns. By default, `pheatmap` uses the **RdYlBu** color palette from *RcolorBrewer* in conjuction with the `colorRampPalette` function to interpolate the 11 color into a smooth-looking palette (Figure 3.37).

[19] A very versatile and modular alternative is the *ComplexHeatmap* package.

```
library("pheatmap")
topGenes = order(rowVars(exprs(x)), decreasing = TRUE)[ seq_len(500) ]
rowCenter = function(x) { x - rowMeans(x) }
pheatmap( rowCenter(exprs(x)[ topGenes, ] ),
  show_rownames = FALSE, show_colnames = FALSE,
  breaks = seq(-5, +5, length = 101),
  annotation_col =
    pData(x)[, c("sampleGroup", "Embryonic.day", "ScanDate") ],
  annotation_colors = list(
    sampleGroup = groupColor,
    genotype = c(`FGF4-KO` = "chocolate1", `WT` = "azure2"),
    Embryonic.day = setNames(brewer.pal(9, "Blues")[c(3, 6, 9)],
                             c("E3.25", "E3.5", "E4.5")),
    ScanDate = setNames(brewer.pal(nlevels(x$ScanDate), "YlGn"),
                        levels(x$ScanDate))
  ),
  cutree_rows = 4
)
```

Let us take a minute to deconstruct this rather massive-looking call to `pheatmap`. The options `show_rownames` and `show_colnames` control whether the row and column names are printed at the sides of the matrix. Because our matrix is large in relation to the available plotting space, the labels would anyway not be readable, and we suppress them. The `annotation_col` argument takes a data frame that carries additional information about the samples. The information is shown in the colored bars on top of the heatmap. There is also a similar `annotation_row` argument, which we haven't used here, for colored bars at the side. `annotation_colors` is a list of named vectors by which we can override the default choice of colors for the annotation bars. Finally, with the `cutree_rows` argument we cut the row dendrogram into four (an arbitrarily chosen number) clusters, and the heatmap shows them by leaving a bit of white space in between. The `pheatmap` function has many further options, and if you want to use it for your own data visualizations, it's worth studying them.

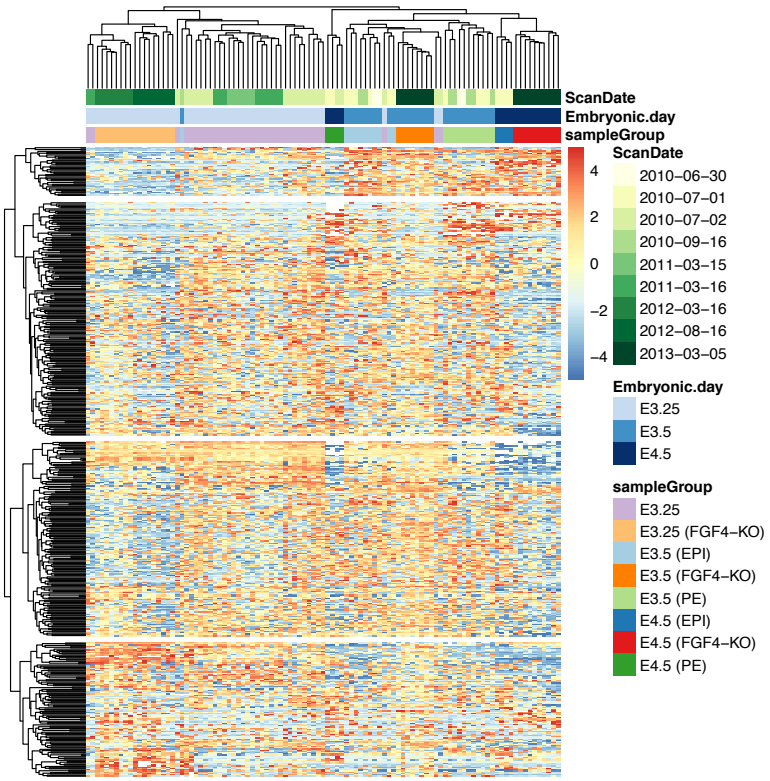


Figure 3.37: A heatmap of relative expression values, i. e., $\log_2$ fold change compared to the average expression of that gene (row) across all samples (columns). The color scale uses a diverging palette, whose neutral midpoint is at 0.

3.10.1 Dendrogram ordering

The ordering of the rows and columns in a heatmap has an enormous impact on the visual impact it makes, and it can be difficult to decide which patterns are real, and which are consequences of arbitrary layout decisions[20]. Let's keep in mind that:

- Ordering the rows and columns by cluster dendrogram (as in Figure 3.37) is an arbitrary choice, and you could just as well make other choices.
- Even if you settle on dendrogram ordering, there is an essentially arbitrary choice at each internal branch, as each branch could be flipped without changing the topology of the tree.

▶ Question **3.10.1.** What other ordering methods can you think of?

▶ Answer **3.10.1.** Among the methods proposed is the travelling salesman problem (McCormick Jr et al., 1972) or projection on the first principal component (for instance, see the examples in the manual page of `pheatmap`).

▶ Question **3.10.2.** Check the manual page of the `hclust` function (which, by default, is used by `pheatmap`) for how it deals with the decision of how to pick which branches of the subtree go left and right.

▶ Question **3.10.3.** Check the argument `clustering_callback` of the `pheatmap` function.

[20] In Chapter 5, we will learn about formal methods for evaluating cluster significance.

3.10.2 Color spaces

Color perception in humans (von Helmholtz, 1867) is three-dimensional[21]. There are different ways of parameterizing this space. Above we already encountered the RGB color model, which uses three values in [0,1], for instance at the beginning of Section 3.4, where we printed out the contents of `groupColor`:

```
groupColor[1]

##      E3.25
## "#CAB2D6"
```

Here, CA is the hexadecimal representation for the strength of the red color channel, B2 of the green and D6 of the green color channel. In decimal, these numbers are 202, 178 and 214, respectively. The range of these values goes from to 0 to 255, so by dividing by this maximum value, an RGB triplet can also be thought of as a point in the three-dimensional unit cube.

The function `hcl` uses a different coordinate system, which consists of the three coordinates hue H, an angle in $[0, 360]$, chroma C, and lightness L as a value in $[0, 100]$. The possible values for C depend on some constraints, but are generally between 0 and 255.

The `hcl` function corresponds to polar coordinates in the CIE-LUV[22] and is designed for area fills. By keeping chroma and luminescence coordinates constant and only varying hue, it is easy to produce color palettes that are harmonious and avoid irradiation illusions that make light colored areas look bigger than dark ones. Our attention also tends to get drawn to loud colors, and fixing the value of chroma makes the colors equally attractive to our eyes.

There are many ways of choosing colors from a color wheel. **Triads** are three colors chosen equally spaced around the color wheel; for example, $H = 0, 120, 240$ gives red, green, and blue. **Tetrads** are four equally spaced colors around the color wheel, and some graphic artists describe the effect as "dynamic". **Warm colors** are a set of equally spaced colors close to yellow, **cool colors** a set of equally spaced colors close to blue. **Analogous color** sets contain colors from a small segment of the color wheel, for example, yellow, orange and red, or green, cyan and blue. **Complementary colors** are colors diametrically opposite each other on the color wheel. A tetrad is two pairs of complementaries. **Split complementaries** are three colors consisting of a pair of complementaries, with one partner split equally to each side, for example, $H = 60,\ 240 - 30,\ 240 + 30$. This is useful to emphasize the difference between a pair of similar categories and a third different one. A more thorough discussion is provided in the references (Mollon, 1995; Ihaka, 2003).

[21] Physically, there is an infinite number of wavelengths of light and an infinite number of ways of mixing them, so other species, or robots, can perceive less or more than three colors.

Figure 3.38: Circles in HCL colorspace. Upper panel: The luminosity L is fixed to 75, while the angular coordinate H (hue) varies from 0 to 360 and the radial coordinate $C = 0, 10, \ldots, 60$. Lower panel: constant chroma $C = 50$, H as above, and varying luminosity $L = 10, 20, \ldots, 90$.

[22] CIE: Commission Internationale de l'Éclairage – see e. g. Wikipedia for more on this.

Lines vs areas

For lines and points, we want that they show a strong contrast to the background, so on a white background, we want them to be relatively dark (low lightness L). For area fills, lighter, more pastell-type colors with low to moderate chromatic content are usually more pleasant.

3.11 Data transformations

Plots in which most points are huddled up in one area, with a lot of sparsely populated space, are difficult to read. If the histogram of the marginal distribution of a variable has a sharp peak and then long tails to one or both sides, transforming the data can be helpful. These considerations apply both to `x` and `y` aesthetics, and to color scales. In the plots of this chapter that involved the microarray data, we used the logarithmic transformation[23] – not only in scatterplots like Figure 3.24 for the x and y-coordinates, but also in Figure 3.37 for the color scale that represents the expression fold changes. The logarithm transformation is attractive because it has a definitive meaning - a move up or down by the same amount on a log-transformed scale corresponds to the same multiplicative change on the original scale: $\log(ax) = \log a + \log x$.

Sometimes the logarithm however is not good enough, for instance when the data include zero or negative values, or when even on the logarithmic scale the data distribution is highly uneven. From the upper panel of Figure 3.39, it is easy to take away the impression that the distribution of `M` depends on `A`, with higher variances for lower `A`. However, this is entirely a visual artefact, as the lower panel confirms: the distribution of `M` is independent of `A`, and the apparent trend we saw in the upper panel was caused by the higher point density at smaller `A`.

```
gg = ggplot(tibble(
  A = exprs(x)[, 1],
  M = rnorm(length(A))),
  aes(y = M))
gg + geom_point(aes(x = A))
gg + geom_point(aes(x = rank(A)))
```

▶ Question **3.11.1.** Can the visual artefact be avoided by using a density- or binning-based plotting method, as in Figure 3.28?

▶ Question **3.11.2.** Can the rank transformation also be applied when choosing color scales e. g. for heatmaps? What does **histogram equalization** in image processing do?

[23] We used it implicitly since the data in the *ExpressionSet* object `x` already come log-transformed.

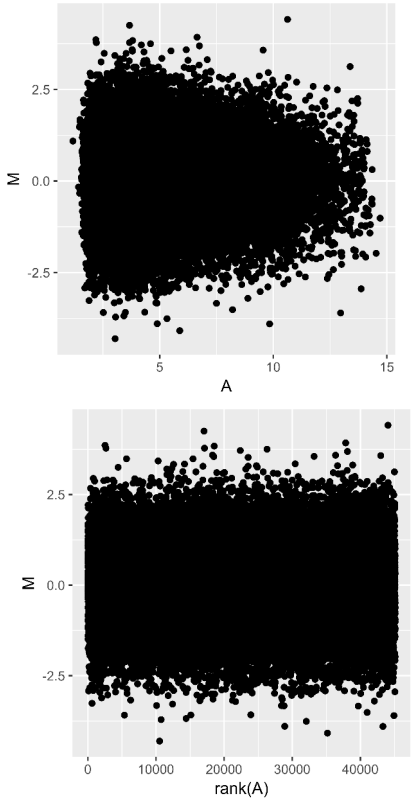


Figure 3.39: The effect of rank transformation on the visual perception of dependency.

3.12 Mathematical symbols and other fonts

We can use mathematical notation in plot labels, using a notation that is a mix of R syntax and LaTeX-like notation (see `help("plotmath")` for details):

```
volume = function(rho, nu)
           pi^(nu/2) * rho^nu / gamma(nu/2+1)

ggplot(tibble(nu    = 1:15,
  Omega = volume(1, nu)), aes(x = nu, y = Omega)) +
geom_line() +
xlab(expression(nu)) + ylab(expression(Omega)) +
geom_text(label =
"Omega(rho, nu)==frac(pi^frac(nu, 2) ~ rho^nu, Gamma(frac(nu, 2) + 1))",
  parse = TRUE, x = 6, y = 1.5)
```

The result is shown in Figure 3.40. It's also easy to switch to other fonts, for instance the serif font Times (Figure 3.41).

```
ggplot(genes, aes( x = value, color = gene)) + stat_ecdf() +
  theme(text = element_text(family = "Times"))
```

In fact, the set of fonts that can be used with a standard R installation are limited, but luckily there is the package *extrafont*, which facilitates using fonts other than R's set of basic PostScript fonts. There's some extra work needed before we can use it, since fonts external to R first need to be made known to it. They could come shipped with your operating system, a with word processor or another graphics application. The set of fonts available and their physical location is therefore not standardized, but will depend on your operating system and further configurations. In the first session after attaching the *extrafont* package, you will need to run the function `font_import` to import fonts and make them known to the package. Then in each session in which you want to use them, you need to call the `loadfonts` function to register them with one or more of R's graphics devices. Finally you can use the `fonttable` function to list the available fonts. You'll need to refer to the documentation of the *extrafonts* package to see how to make this work on your machine.

▶ Question **3.12.1.** Use *extrafont* to produce a version of Figure 3.41 with the font "Bauhaus 93" (or another one available on your system).

▶ Question **3.12.2.** Have a look at the code producing Figure 3.1.

3.13 Genomic data

To visualize genomic data, in addition to the general principles we have discussed in this chapter, there are some specific considerations. The data are usually associated with genomic coordinates. In fact genomic coordinates offer a great organising principle for the integration of genomic data. You will probably have seen genome browser displays such as in Figure 3.43. Here we'll briefly show how to produce such plots programmatically, using your data as well as public annotation. We can only give a short glimpse, and we refer to resources such as Bioconductor for a fuller picture.

The main challenge of genomic data visualization is the size of genomes. We need visualizations at multiple scales, from whole genome down to the nucleotide level.

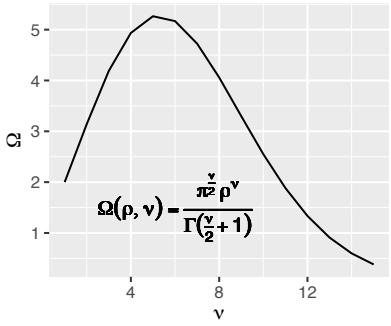


Figure 3.40: Volume Ω of the ν-dimensional sphere with radius $\rho = 1$, for $\nu = 1, \ldots, 15$.

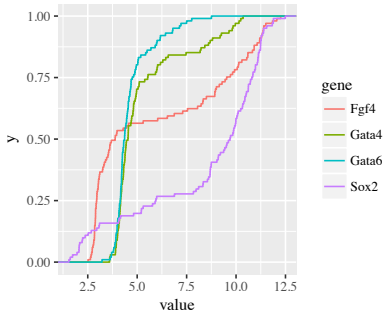


Figure 3.41: As Figure 3.22, with a different font.

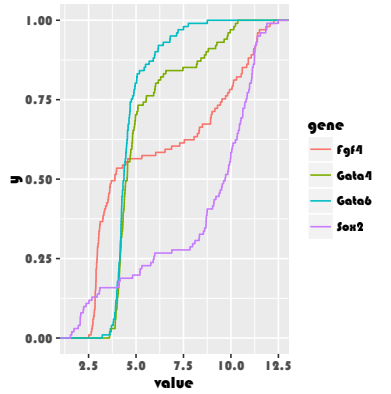


Figure 3.42: As Figure 3.22, with font "Bauhaus 93".

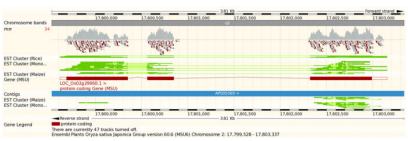


Figure 3.43: Screenshot from Ensembl genome browser, showing gene annotation of a genomic region as well as a read pile-up visualization of an RNA-Seq experiment.

It should be easy to zoom in and and out, and we may need differemt visualization strategies for the different size scales. It can be convenient to visualize biological molecules (genomes, genes, transcripts, proteins) in a linear manner, although their embedding in the 3D physical world can matter (a lot).

Let's start with some fun examples, an ideogram plot of human chromosome 1 (Figure 3.44) and a plot of the genome-wide distribution of RNA editing sites (Figure 3.45).

```
library("ggbio")
data("hg19IdeogramCyto", package = "biovizBase")
plotIdeogram(hg19IdeogramCyto, subchr = "chr1")
```

```
## Warning: 'panel.margin' is deprecated. Please use 'panel.spacing'
property instead
```

```
## Warning: 'panel.margin' is deprecated. Please use 'panel.spacing'
property instead
```

```
## Warning: 'panel.margin' is deprecated. Please use 'panel.spacing'
property instead
```

Figure 3.44: Chromosome 1 of the human genome: ideogram plot.

The `darned_hg19_subset500` lists a selection of 500 RNA editing sites in the human genome. It was obtained from the Database of RNA editing in flies, mice and humans (DARNED, `http://darned.ucc.ie`). The result is shown in Figure 3.45.

```
library("GenomicRanges")
data("darned_hg19_subset500", package = "biovizBase")
autoplot(darned_hg19_subset500, layout = "karyogram",
         aes(color = exReg, fill = exReg))
```

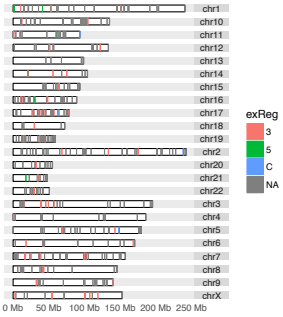


Figure 3.45: Karyogram with RNA editing sites. `exReg` indicates whether a site is in the coding region (C), 3'- or 5'-UTR.

▶ Question **3.13.1.** Fix the ordering of the chromosome in Figure 3.45 and get rid of the warning about chromosome lengths.

▶ Answer **3.13.1.** The information on chromosome lengths in the hg19 assembly of the human genome is (for instance) stored in the `ideoCyto` dataset. In the following, we also use the function `keepSeqlevels` to subset and reorder the chromosomes. See Figure 3.46

```
data("ideoCyto", package = "biovizBase")
dn = darned_hg19_subset500
seqlengths(dn) = seqlengths(ideoCyto$hg19)[names(seqlengths(dn))]
dn = keepSeqlevels(dn, paste0("chr", c(1:22, "X")))
autoplot(dn, layout = "karyogram",
         aes(color = exReg, fill = exReg))
```

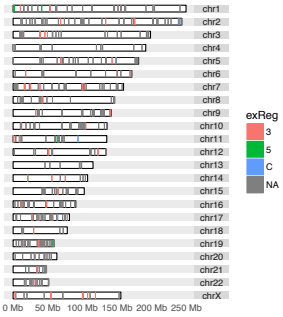


Figure 3.46: Improved version of Figure 3.45.

What type of object is `darned_hg19_subset500`?

```
darned_hg19_subset500[1:2,]
```

```
## GRanges object with 2 ranges and 10 metadata columns:
##       seqnames              ranges strand |       inchr       inrna
##          <Rle>           <IRanges>  <Rle> | <character> <character>
##   [1]     chr5 [86618225, 86618225]      - |           A           I
##   [2]     chr7 [99792382, 99792382]      - |           A           I
##              snp        gene      seqReg       exReg      source
```

```
##       <character> <character> <character> <character> <character>
##   [1]        <NA>        <NA>           0        <NA>    amygdala
##   [2]        <NA>        <NA>           0        <NA>        <NA>
##           ests      esta      author
##       <integer> <integer> <character>
##   [1]         0         0    15342557
##   [2]         0         0    15342557
##   -------
##   seqinfo: 23 sequences from an unspecified genome; no seqlengths
```

It is a *GRanges* object, that is, a specialized class from the Bioconductor project for storing data that are associated with genomic coordinates. Its first three columns are obligatory: `seqnames`, the name of the containing biopolymer (in our case, these are names of human chromosomes), `ranges`, the genomic coordinates of the intervals (in this case, the intervals all have length 1, as they each refer to a single nucleotide), and the DNA strand from which the RNA is transcribed. You can find out more on how to use this class and its associated infrastructure in the documentation, e. g., the vignette of the *GenomicRanges* package. Learning it is worth the effort if you want to work with genome-associated datasets, as it allows for convenient, efficient and safe manipulation of these data and provides many powerful utilities.

To Do

- Along chromosome plots, annotation + data, with sequence as the integrating principle (Gviz).
- HilbertViz

3.14 Recap of this chapter

- You should now be comfortable making beautiful, versatile and easily extendable plots using *ggplot2*'s `ggplot`.
- You should have a basic understanding of the grammar of graphics: the main word classes and the basic vocabulary.
- You understand the importance of choosing the right colors, proportions, and the right geom objects.
- Don't be afraid of setting up your data for faceting – this is a great quick way to look at many different ways to slice the data in different ways.
- Now you are prepared to explore *ggplot2* and plot your data on your own.

3.15 Further reading

The most useful books about *ggplot2* is the second edition of Wickham (2016) and the ggplot2 website. There are a lot of *ggplot2* code snippets online, which you will find through search engines after some exercise (stay critical, not everything online is

true, or up-to-date). Of course, the foundation of the system is based on Wilkinson (2005) and the ideas by Tukey (1977); Cleveland (1988).

3.16 Exercises

▶ Exercise **3.1** (Themes). Explore how to change the visual appearance of plots with themes. For example:

```
ggcars = ggplot(mtcars, aes(x=hp, y=mpg)) + geom_point()
ggcars
ggcars + theme_bw()
ggcars + theme_minimal()
```

▶ Exercise **3.2** (Color names in R). Have a look at `http://research.stowers-institute.org/efg/R/Color/Chart`

▶ Exercise **3.3** (xkcd). On a lighter note, you can even modify *ggplot2* to make plots in the style of the popular webcomic XKCD. You do this through manipulating the font and themes of ggplot2 objects. See `http://stackoverflow.com/questions/12675147/how-can-we-make-xkcd-style-graphs-in-r`.