Chapter 1

High Quality Graphics in R

Wolfgang Huber, with contributions from Trevor Martin and Susan Holmes

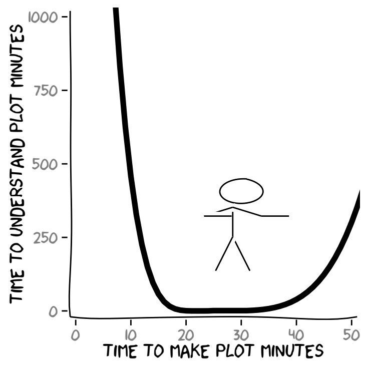


Figure 1.1: An elementary law of visualisation.

There are two important types of data visualization. The first enables a scientist to effectively explore the data and make discoveries about the complex processes at work. The other type of visualization should be a clear and informative illustration of the study's results that she can show to others and eventually include in the final publication.

Both of these types of graphics can be made with plotting functions in R. In fact, R offers different approaches for making graphics, each with their its advantages and limitations. *fixme: Review this when chapter is finished:* base R graphics were historically first, simple and procedural. Complex plots can quickly get messy to program. A more high-level approach – *grammar of graphics*, plots are built in modular 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. *lattice graphics* – for showing more than 2 variables at a time, with lattice graphics, we can attempt visualisation of data to up to 4 or 5 dimensions. In the end of the chapter, we cover some specialized forms of plotting such as maps and ideograms, still building on the base concept of the grammar of graphics.

1.1 Goals for this chapter

- Review the basics of *base* R plotting
- Understand the logic behind the *grammar of graphics* concept
- Introduce *ggplot2*'s `qplot` function
- Show how to build complex plots from the ground up using *ggplot2*'s `ggplot` function

- See how to plot 1D, 2D, 3-5D data, and understand faceting
- Become good at rapidly exploring data sets by visualization
- Create beautiful and intuitive plots for scientific presentations and publications

1.2 Built-in R Plotting

R has built-in plotting functions that can be used to quickly and easily visualize data.

The most basic of these is the `plot` function. In Figure 2 it is used to show data from an enzyme-linked immunosorbent assay (ELIZA) assay. An ELIZA assay uses antibodies and the resulting colour change created by them to identify an enzyme and quantify its activity, in this case 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 0.0488   0.017
## 2   1 0.0488   0.018
## 3   1 0.1953   0.121
## 4   1 0.1953   0.124
## 5   1 0.3906   0.206
## 6   1 0.3906   0.215

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

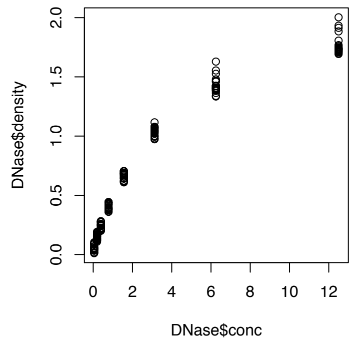


Figure 1.2: Plot of concentration vs. density for an ELISA assay of DNase.

This basic plot can be customized, for example by changing the plotting symbol and axis labels as shown in Figure 3 by using the parameters `xlab`, `ylab` and `pch` (plot character). The information about the labels 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")
```

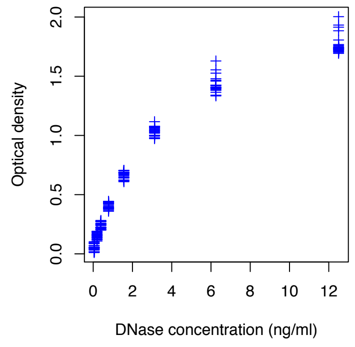


Figure 1.3: Same data as in Figure 2 but with better axis labels and a different plot symbol.

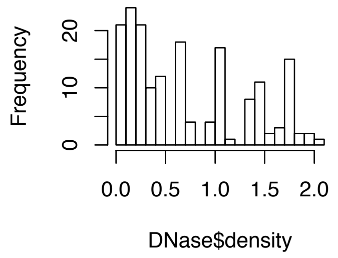
Besides scatterplots, we can also use built-in functions to create histograms and boxplots (Figure 4).

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

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

Boxplots are convenient to show multiple distributions next to each other in a compact space, and they are universally preferable to the barplots with error bars often seen in biological papers. We will see more about plotting univariate distributions in Section 0.7.

These plotting functions are great for quick interactive exploration of data; but we run quickly into their limitations if we want to create more sophisticated displays of data. In this chapter, we are going 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. But first let us load up an example dataset.



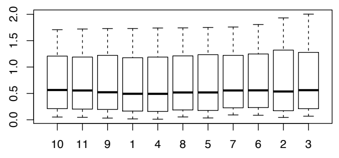


Figure 1.4: Histogram of the density from the ELISA assay, and boxplots of these values stratified by the assay run. The boxes are ordered along the axis in lexicographical order because the runs were stored as text strings. We could use R's type conversion functions to achieve numerical ordering.

1.3 An Example Dataset

To properly testdrive the *ggplot2* functionality, we are going to need a data set 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 data set 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. Eventually, by further and further specification, the different cell types and tissues arise that are needed for a full organism. The aim of the experimentOhnishi et al. (2014) was to see how and when the first *fixme: second?* of these *symmetry breaking* events occurs. 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
```

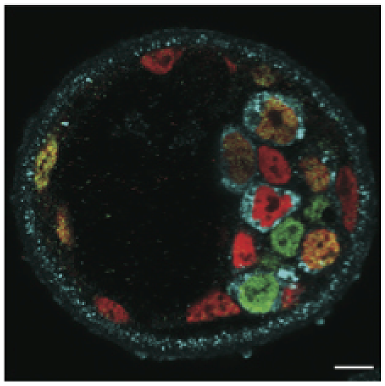


Figure 1.5: Single-section immunofluorescence image of the E3.5 mouse blastocyst stained for Serpinh1, a marker of primitive endoderm (blue), Gata6 (red) and Nanog (green). Scale bar: 10 μm. *fixme: need to get a similar (and perhaps even better) image from Takashi; this one is from the NCB paper.*

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

Let's have a look what informations is available about the samples[1].

```
head(pData(x), 1)

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

[1] The notation #CAB2D6 is a hexadecimal representation of the RGB coordinates of a colour; more on this in Section 0.10.2.

The information provided is a mix of information about the cells (i.e., the age, size and genotype of the embryo from which they were obtained) and technical information (scan date, raw data file name). Moreover, in the paper the authors divided the cells into 8 biological groups (`sampleGroup`), based on age, genotype and lineage, and they defined a colour scheme to represent these groups (`sampleColour`)[2].

[2] In this chapter we'll use the spelling *colour* (rather than *color*). This is to stay consistent with the spelling adopted by the R package *ggplot2*, which we'll use a lot. Other packages, like *RColorBrewer*, use the other spelling. In some cases, function or argument names are duplicated to allow users to use either choice, although you cannot rely on that.

```
groupSize <- table(x$sampleGroup)
groupSize

##
##           E3.25 E3.25 (FGF4-KO)      E3.5 (EPI)
##              36              17              11
##  E3.5 (FGF4-KO)       E3.5 (PE)      E4.5 (EPI)
##               8              11               4
##  E4.5 (FGF4-KO)       E4.5 (PE)
##              10               4
```

By convention, time in the development of the mouse embryo is measured in days, and reported as, for instance, `E3.5`. 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) differentiate into different cell lineages, called pluripotent epiblast (EPI) and primitive endoderm (PE),

1.4 ggplot2

The *ggplot2* package is a package created by Hadley Wickham that implements the idea of *grammar of graphics* – a concept created by Leland Wilkinson in his eponymous bookWilkinson and Wills (2005). Comprehensive documentation for the packageWickham (2009) can be found on its website. The online documentation includes great 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.

To load the package we run:

```
library("ggplot2")
```

We start by plotting the number of samples for each of the 8 groups using *ggplot2*'s `qplot` function in Figure 6. Using this function is similar to using the R's built-in `plot` function – but it can be combined with the techniques we learn in this chapter to create striking plots.

```
qplot(x = names(groupSize),
      y = as.vector(groupSize),
      geom = "bar", stat = "identity")
```

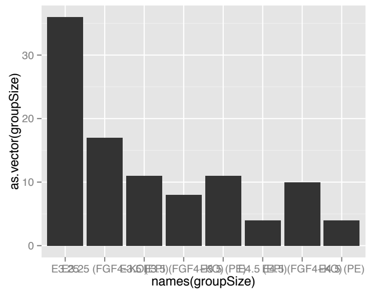


Figure 1.6: Making our first *ggplot2* figure using `qplot`.

We just wrote our first "sentence" using the grammar of graphics. We specified the *data* via the arguments `x` and `y`; we specified that we want each data item to be represented by bars – a type of *geometric object* (other possible geometric objects include point and lines, we will see these later). And finally we set the `stat` parameter to "identity" to indicate that we want to see the data as is; the name of this parameter is short for *statistic*, which is what we call any function of data. The identity statistic is trivial as it just returns the data themselves, but there are other more interesting statistics, such as binning, smoothing, averaging, or other operations that summarize 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.

The resulting plot, Figure 6, is not bad, but there are several potential improvements. Instead of R expressions for axis labels, we would like to use more descriptive English expressions. And we may want to colour the bars to help us quickly see which one corresponds to which group. This might be particularly useful if we use the same colours for the same message in several plots. To solve these problems, let us first reconstruct the colour scheme used in `pData(x)`; we rely on the fact that all the samples from the same group were assigned the same colour.

```
groupColour <- vapply(split(x$sampleColour, x$sampleGroup),
     FUN = unique, FUN.VALUE = "")
groupColour

##            E3.25 E3.25 (FGF4-KO)      E3.5 (EPI)
##       "#CAB2D6"       "#FDBF6F"       "#A6CEE3"
##   E3.5 (FGF4-KO)       E3.5 (PE)      E4.5 (EPI)
##       "#FF7F00"       "#B2DF8A"       "#1F78B4"
##   E4.5 (FGF4-KO)       E4.5 (PE)
##       "#E31A1C"       "#33A02C"
```

Now we can add a `fill` parameter and descriptive axis labels, as in Figure 7:

```
qplot(x = names(groupSize),
      y = as.vector(groupSize),
      geom = "bar", stat = "identity",
      xlab = "Groups", ylab = "Number of Samples",
      fill = names(groupSize)) +
   scale_fill_manual(values = groupColour, name="Colour code")
```

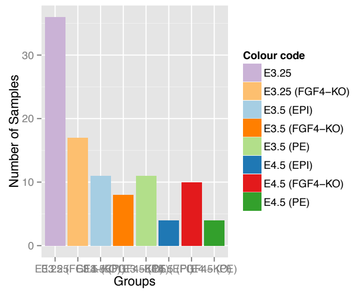


Figure 1.7: Similar to Figure 6, but with coloured bars and better axis descriptions.

Let's dissect the above "sentence". The added axis labels are straightforward (`xlab`, `ylab`). Then we stated that we want the bars to be coloured (filled) based on `names(groupSize)` (which in this case co-incidentally also is the x data, but that need not be so). But then we didn't just want *ggplot2* to use the default colours, but provide our own mapping from group names to colours. How did we do this? In this case, not by adding an additional parameter to the `qplot` function, but by using the + operator and "adding" to the graphics object returned by `qplot` the information about our prefered colour assignment. This information is encapsulated by the object returned by the `scale_fill_manual` function, and comprises the names-to-colour mapping (as a named vector) and a title.

This object-oriented way of doing things –taking a graphics object already produced in some way and then further refining it– can be more convenient and easy to manage then what normally is the alternative –providing all the instructions needed upfront, to the single function that creates the graphic. We can quickly try out different visualisation ideas without having to rebuild our plots each time from scratch, but rather store the basic object and then modify it in different ways.[3]

[3] The + operator plays well with R's preference for pass-by-value semantics, where you normally don't modify an object in place, but rather, every function call on an object treats its inputs as read-only, although it might return a modified version of its input. In R, the + operator is also just such a function, which takes two arguments (the expressions to its left and right) and returns a single result.

As you can see from our colour legend, we have chosen our colours well so that, for example, the mutant genotypes have more firey colours (orange, red), the PE lineage samples have a greenish colour and the EPI ones a blueish colour. In this way, we can easily look for trends in our data later at a glance since we have already grouped the samples by colour.

One more thing that we need to fix in Figure 7 is the readability of the bar labels. Right now they are running into each other — a common problem when you have descriptive names. Let's rotate the text so that it is more readable as shown in Figure 8.

```
qplot(x = names(groupSize),
      y = as.vector(groupSize),
      geom = "bar", stat = "identity",
      xlab = "Groups", ylab = "Number of Samples",
      fill = names(groupSize)) +
 scale_fill_manual(values = groupColour, name="Colour code") +
theme(axis.text.x = element_text(angle = 90, hjust = 1))
```

We have added on another clause, produced by the `theme` function, which indicates that we like x-axis text set at an angle of 90 degrees and right justified (`hjust`; the default would be to center it).

Now we have a great looking plot that clearly conveys our data on the number of samples in each group and also the types of groups that we are comparing.

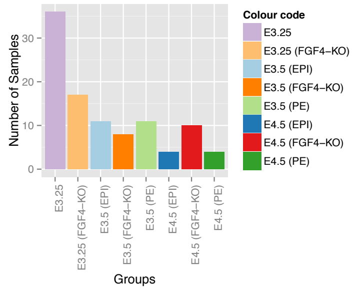


Figure 1.8: Fixing the barplot names through rotation of the labels.

1.5 The Grammar of Graphics

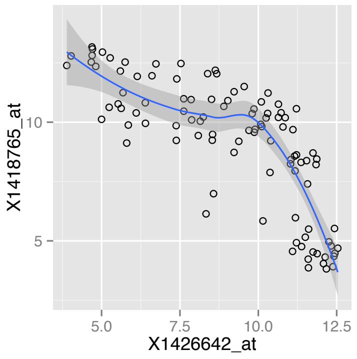


Figure 1.9: A scatterplot with three layers that show different statistics of the same data: points, a smooth regression line, and a confidence band.

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

1. a dataset
2. a choice of geometric object that serves as the visual representations of the data – for instance, points, lines, rectangles, contours
3. a description 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. a statistical summarisation rule
5. a coordinate system
6. a facet specification, i. e. the use of several plots to look at the same data

In the examples above, Figures 6–8, 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, and the statistical summary was the trivial one (i. e., none). We did not make use of a facet specification in the plots above, but we'll see examples later on (Section 0.9).

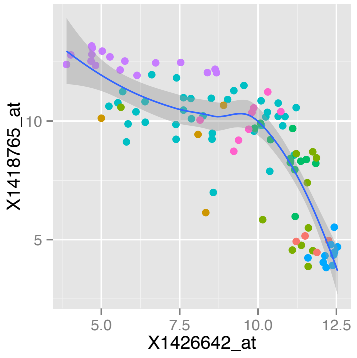


Figure 1.10: As Figure 9, but in addition with points coloured by the sample group (as in Figure 7). We can now see that the expression values of the gene Timd2 (whose mRNA is targeted by the probe 1418765_at) are consistently high in the early time points, whereas its expression goes down in the EPI 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 cell-to-cell variability.

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 layersWickham (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 <- data.frame(t(exprs(x)), pData(x))
ggplot( dftx, aes( x = X1426642_at, y = X1418765_at)) +
  geom_point( shape = 1 ) +
  geom_smooth( method = "loess" )
```

We assembled a copy of the expression data (`exprs(x)`) and the sample annotation data (`pData(x)`) all together into the *data.frame* `dftx` – since this is the data format that *ggplot2* functions most easily take as input (more on this in Sections 0.7.1 and 0.7.8).

We can further enhance the plot by using colours – since each of the points in Figure 9 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( colour = sampleColour), shape = 19 ) +
  geom_smooth( method = "loess" ) +
  scale_colour_discrete( guide = FALSE )
```

Question **1.5.1** *In the code above we defined the* `colour` *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* `colour` *aesthetics for all layers, i. e., move it into the argument list of* `ggplot`*? What happens if we omit the call to* `scale_colour_discrete`*?*

Question **1.5.2** *Is it always meaningful to summarize scatterplot data with a regression line as in Figures 9 and 10?*

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[4].

[4] Note that here were need to use the original feature identifiers (e. g., "1426642_at", without the leading "X"). These 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 `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 them correctly.

```
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, a geometric object, and labels (through the scale parameters). Most geometric objects implicitly call a suitable default statistical summary of the data, and vice versa. For example, if you are using `geom_histogram`, *ggplot2* implicitly bins your data and displays the results in barplot (`geom_bar`) format. Thus, you could equivalently plot your histogram by calling `geom_bar` with `stat_bin`.

```
dfx <- as.data.frame(exprs(x))
p1 <- ggplot(dfx, aes(x = ‘20 E3.25‘)) +
               geom_histogram(binwidth = 0.2)
p2 <- ggplot(dfx, aes(x = ‘20 E3.25‘)) +
```

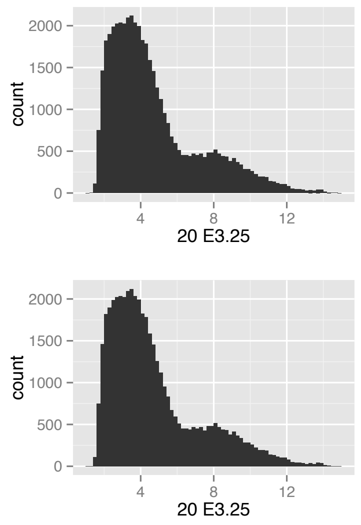


Figure 1.11: Two different ways of creating the same histogram using the grammar of graphics.

```
                geom_bar(stat = "bin", binwidth = 0.2)
library("gridExtra")
grid.arrange(p1, p2, nrow = 2)
```

grid.arrange is a convenient helper function to arrange multiple plots (in this case, stored in the objects p1 and p2) in a figure.

Let's come back to the barplot example from above and see how it is done in the ggplot way.

```
pb <- ggplot(data.frame(
                 name = names(groupSize),
                 size = as.vector(groupSize)),
              aes(x = name, y = size))
```

For now we have simply created a plot object pb and have not generated a plot yet. In fact we cannot make a plot yet,

```
pb

## Error:  No layers in plot
```

because we haven't specified what geometric object we want to use for our plot. All that we have in our pb object so far are the data and the aesthetics.

Now we can literally add on the other components of our plot through using the + operator:

```
pb <- pb + geom_bar(stat = "identity") +
   aes(fill = name) +
   scale_fill_manual(values = groupColour, name = "Colour code") +
   theme(axis.text.x = element_text(angle = 90, hjust = 1))  +
   xlab("Groups") + ylab("Number of Samples")
pb
```

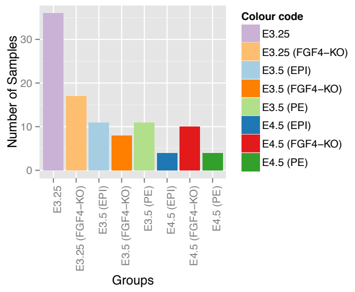


Figure 1.12: Using ggplot to create our barplot.

Thus we recreate our previous qplot result using the ggplot approach. This modular approach allows us a lot of freedom in creating figures and setting parameters. For example we can switch our plot to polar coordinates to create a popular 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
```

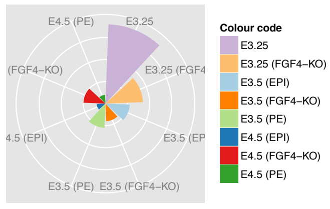


Figure 1.13: Using ggplot to create a barplot with polar coordinates.

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

1.6 1D visualisations

A fairly common task in biological data analysis is the comparison between several samples of univariate measurements. In this section we'll explore some possibilities for visualizing and comparing such samples. As an example, we'll use the intensities of a set of four genes Fgf4, Sox2, Gata6, Gata4 and Gapdh[5]. On the array, they are represented by

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

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

1.6.1 Data tidying I – matrices versus data.frame

Data analysis, and in particular visualisation, often involves a lot of shuffling around of data, until they are in the right shape and format for the algorithm or plotting routine. *ggplot2* likes its data in *data.frame* objects, and more specifically, in the long format[6]. The reasons behind this choice are well explained in Hadley Wickham's paper on *tidy data*Wickham (2014). Essentially the long format table is a very universal representation that can accommodate pretty much any data type and easy to programmatically interface to. On the other hand, for a specific data type, it may not always be the most efficient way of storing data, and it cannot easily transport rich metadata (i. e., data about the data)[7]. For instance, our example dataset x is stored as an object in Bioconductor's *ExpressionSet* class, which has multiple components, most importantly, the matrix `exprs(x)` with 45101 rows and 101 columns. The matrix elements are the gene expression measurements, and the feature and sample associated with each measurement are implied by its position (row, column) in the matrix; in contrast, in the long table format, the feature and sample identifiers need to be stored explicitly with each measurement. Besides, x has additional components, including the *data.frame*s `fData(x)` and `pData`, which provide various sets metadata about the microarray **f**eatures and the **p**henotypic information about the samples. To extract data from this representation and convert them into a *data.frame*, we use the function `melt`, which we'll explain in more detail below.

[6] More about this below.

[7] In other words, simple tables or *data.frame*s cannot offer all the nice features provided by object oriented approaches, such as encapsulation, abstraction of interface from implementation, polymorphism, inheritance and reflection.

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

##         probe  sample value
## 1 1420085_at 1 E3.25  3.03
## 2 1418863_at 1 E3.25  4.84
## 3 1425463_at 1 E3.25  5.50
## 4 1416967_at 1 E3.25  1.73
## 5 1420085_at 2 E3.25  9.29
## 6 1418863_at 2 E3.25  5.53
```

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

```
genes$gene <- names(probes)[ match(genes$probe, probes) ]
```

1.6.2 Barplots

A popular way to display data such as in our *data.frame* `genes` is through barplots. See Fig. 14.

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

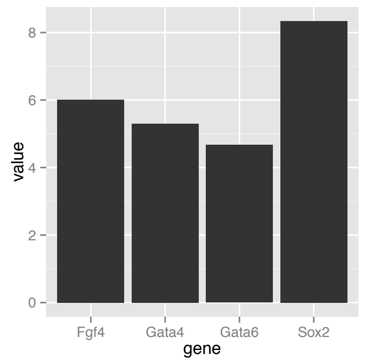


Figure 1.14: Barplots showing the means of the distributions of expression measurements from 4 probes.

In Figure 14, 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 summarisation 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[8].

[8] In fact, if the mean is an appropriate summary, such as for highly skewed distributions, or data sets 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",
               mult = 1, width = 0.25)
```

Here, we see again the principle of layered graphics:fig: 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 **c**onfidence **l**imits) 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 coloured the bars in lighter colours for better contrast.

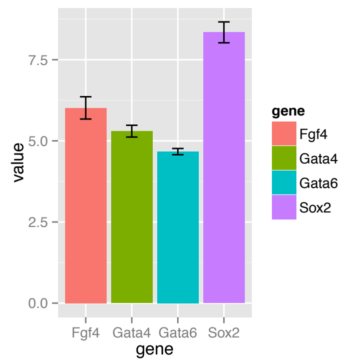


Figure 1.15: Barplots with error bars indicating standard error of the mean.

1.6.3 Boxplots

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

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

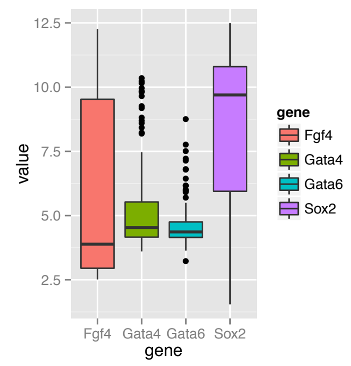


Figure 1.16: Boxplots.

Compared to the barplots, this takes a similar amount of space, but is much more informative. In Figure 16 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.

1.6.4 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()
```

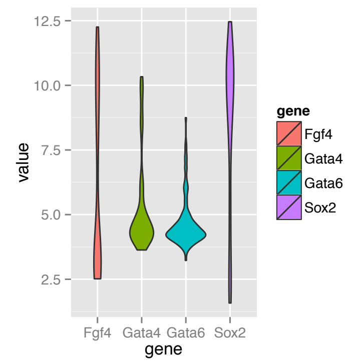


Figure 1.17: Violin plots.

1.6.5 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 overlapWilkinson (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 18. 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.

A fun alternative is provided by the package *beeswarm*. It works with base R graphics and is not directly integrated into *ggplot2*'s data flows, so we can either use the base R graphics output, or pass on the point coordinates to `ggplot` as follows.

```
library("beeswarm")
bee <- beeswarm(value ~ gene, data = genes, spacing = 0.7)
```

```
ggplot(bee, aes( x = x, y = y, colour = x.orig)) +
  geom_point(shape = 19) + xlab("gene") + ylab("value") +
  scale_fill_manual(values = probes)
```

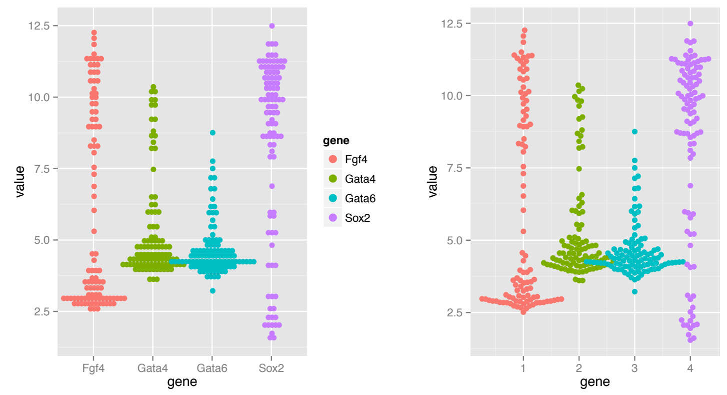


Figure 1.18: Left: dot plots, made using `geom_dotplot` from *ggplot2*. Right: beeswarm plots, with layout obtained via the *beeswarm* package and plotted as a scatterplot with `ggplot`.

The plot is shown in the right panel of Figure 18. The default layout method used by `beeswarm` is called *swarm*. It places points in increasing order. If a point would overlap an existing point, it is shifted sideways (along the x-axis) by a minimal amount sufficient to avoid overlap.

As you have seen in the above code examples, some twiddling with layout parameters is usually needed to make a dot plot or a beeswarm plot look good for a particular dataset.

1.6.6 Density plots

Yet another way to represent the same data is by lines plots of the density plots

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

Density estimation has a number of complications, and you can see these in Figure 19. 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; if the window is made bigger, pronounced features of the density 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 19 can become especially problematic if the sample sizes for the curves differ.

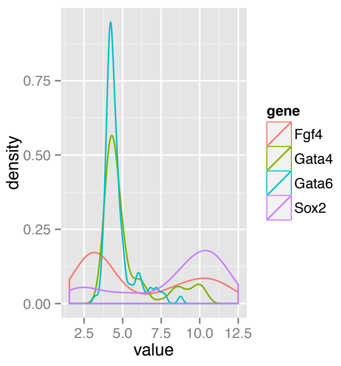


Figure 1.19: Density plots.

1.6.7 ECDF plots

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

$$F(x) = P(X \leq x), \tag{1.1}$$

where x takes all values along the real axis. The density of X is then the derivative of F, if it exists[9]. 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 simply

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

$$F_n(x) = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}_{x \leq x_i}. \tag{1.2}$$

An important property is that even for limited sample sizes n, the ECDF F_n is not very far from the CDF, F. 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 discussed above.

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

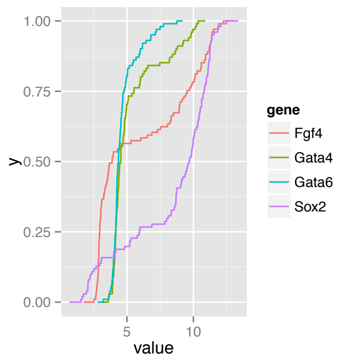


Figure 1.20: Empirical cumulative distribution functions (ECDF).

1.6.8 Data tidying II - Wide vs long format

Let us revisit the `melt` command from above. In the resulting *data.frame* `genes`, each row corresponds to exactly one measured value, stored in the column `value`. Then there are additional columns probe and sample, which store the associated covariates. Compare this to the following *data.frame* (for space reasons we print only the first five columns):

```
as.data.frame(exprs(x)[probes, ])[, 1:5]

##              1 E3.25 2 E3.25 3 E3.25 4 E3.25 5 E3.25
## 1420085_at      3.03    9.29    2.94    9.72    8.92
## 1418863_at      4.84    5.53    4.42    5.98    4.92
## 1425463_at      5.50    6.16    4.58    4.75    4.63
## 1416967_at      1.73    9.70    4.16    9.54    8.71
```

This *data.frame* has several columns of data, one for each sample (annotated by the column names). Its rows correspond to the four probes, annotated by the row names. This is an example for a data table in *wide format*.

Now suppose we want to store somewhere not only the probe identifiers but also the associated gene symbols. We could stick them as an additional column into the wide format *data.frame*, and perhaps also throw in the genes' ENSEMBL identifier for good measure. But now we immediately see the problem: the *data.frame* now has some columns that represent different samples, and others that refer to information for all samples (the gene symbol and identifier) and we somehow have to "know" this when interpreting the *data.frame*. This is what Hadley Wickham calls *untidy data*[10]. In contrast, in the tidy *data.frame* `genes`, we can add these columns, yet still know that each row forms exactly one observation, and all information associated with that observation is in the same row.

[10] There are many different ways for data to be untidy.

In tidy dataWickham (2014),

1. each variable forms a column,
2. each observation forms a row,
3. each type of observational unit forms a table.

A potential drawback is efficiency: even though there are only 4 probe – gene symbol relationships, we are now storing them 404 times in the rows of the *data.frame* `genes`. Moreover, there is no standardisation: we chose to call this column `symbol`, but the next person might call it `Symbol` or even something completely different, and when we find a *data.frame* that was made by someone else and that contains a column `symbol`, we can hope, but have no guarantee, that these are valid gene symbols. Addressing such issues is behind the object-oriented design of the specialized data structures in Bioconductor, such as the *ExpressionSet* class.

1.7 2D visualisation: Scatter Plots

Scatter plots are useful for visualizing treatment–response comparisons (as in Figure 3), associations between variables (as in Figure 10), or paired data (e. g., a disease biomarker in several patients before and after treatment). We use the two dimensions of our plotting paper, or screen, to represent the two variables.

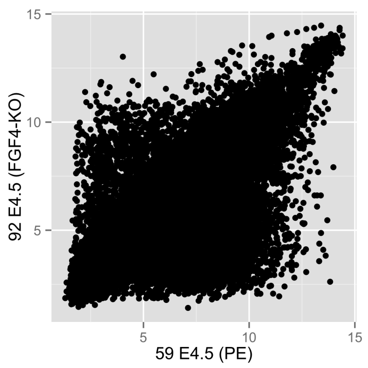


Figure 1.21: Scatterplot of 45101 expression measurements for two of the samples.

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 *data.frame* `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 14. 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 22).

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

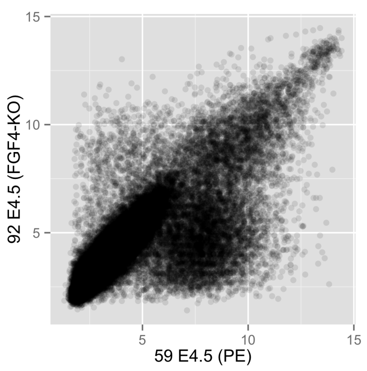


Figure 1.22: As Figure 21, but with semi-transparent points to resolve some of the overplotting.

This is already better than Figure 21, but in the very density 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 23.

```
scp + geom_density2d()
```

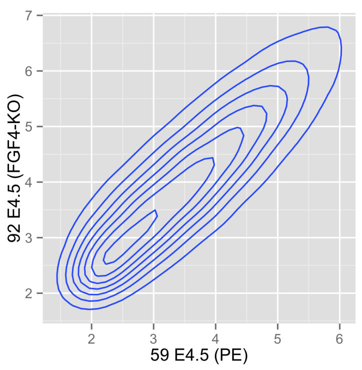


Figure 1.23: As Figure 21, but rendered as a contour plot of the 2D density estimate.

However, we see in Figure 23 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 24, 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 24.

```
library("RColorBrewer")
colourscale <- scale_fill_gradientn(
    colours = 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") +
  colourscale + coord_fixed()
```

We used the function `brewer.pal` from the package *RColorBrewer* to define the colour 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 shapes in Section 0.8.1 and about colours in Section 0.10.

The density based plotting methods in Figure 24 are more visually appealing and interpretable than the overplotted point clouds of Figures 21 and 22,

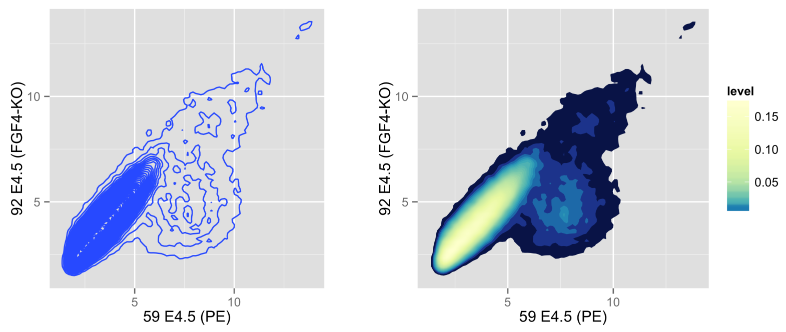


Figure 1.24: Left: as Figure 23, but with smaller smoothing bandwidth and tighter binning for the contour lines. Right: with colour filling.

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 binningCarr et al. (1987).

```
library("hexbin")
scp + stat_binhex() + coord_fixed()
scp + stat_binhex(binwidth = c(0.2, 0.2)) + colourscale +
  coord_fixed()
```

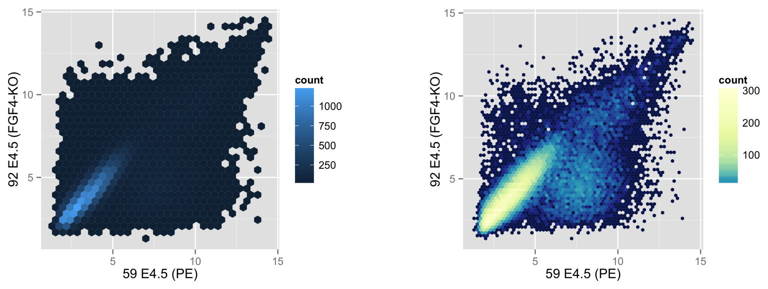


Figure 1.25: Hexagonal binning. Left: default parameters. Right: finer bin sizes and customized colour scale.

1.7.1 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[11]. Moreover, the graph dimensions also depend on the presence or absence of additional decorations, like the colour scale bars in Figure 25.

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

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 bankingCleveland et al. (1988).

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

```
library("ggthemes")
sunsp <- data.frame(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 26. 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 26. 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.

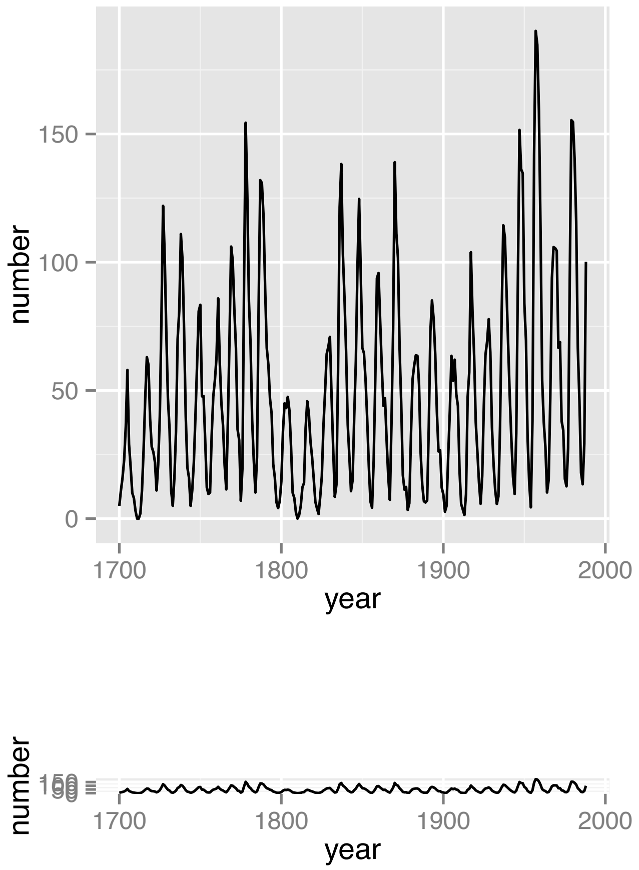


Figure 1.26: The sunspot data. In the upper panel, the plot shape is roughly quadratic, a frequent default choice. In the lower panel, a technique called *banking* was used to choose the plot shape.

1.8 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 colours. The `geom_point` geometric object offers the following aesthetics (beyond `x` and `y`):

- `fill`
- `colour`
- `shape`
- `size`
- `alpha`

They are explored in the manual page of the `geom_point` function. `fill` and `colour` refer to the fill and outline colour of an object; `alpha` to its transparency level. Above, in Figures 22 and following, we have used colour 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 most often we are better off with one or two additional dimensions and mapping them to a choice of the available aesthetics.

1.8.1 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[12]

[12] The first line, `mutate` is not strictly necessary – it's just some data wrangling to make the plots look better.

```
dftx <- mutate(dftx, lineage = factor(sub("^$", "no", lineage),
  levels = c("no", "EPI", "PE", "FGF4-KO")))

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

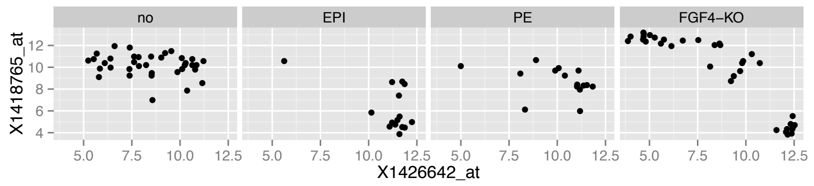


Figure 1.27: An example for *faceting*: the same data as in Figure 9, but now split by the categorical variable `lineage`.

The result is shown in Figure 27. 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 28.

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

Another useful function is `facet_wrap`: if the faceting variables 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.

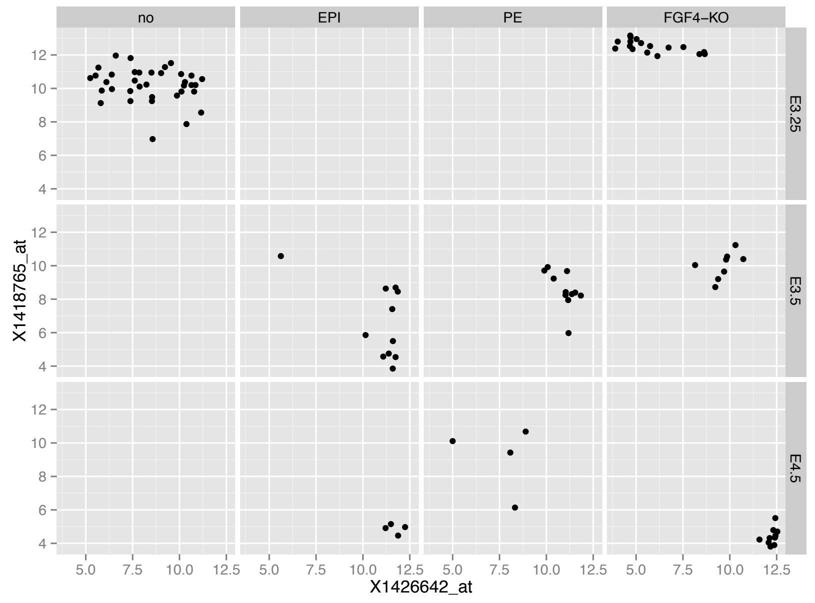


Figure 1.28: *Faceting*: the same data as in Figure 9, split by the categorical variables `Embryonic.day` (rows) and `lineage` (columns).

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

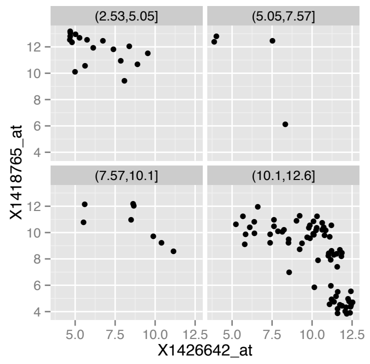


Figure 1.29: *Faceting*: the same data as in Figure 9, split by the continuous variable `X1450989_at` and arranged by `facet_wrap`.

We see in Figure 29 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`.

Axes scales In Figures 27–29, 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 14–18

1.8.2 plotly, webgl

fixme: Mention them as they are cool.

1.9 Colour

An important consideration when making plots is the colouring that we use in them. Most R users are likely familiar with the built-in R colour scheme, used by base R graphics, as shown in Figure 30.

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

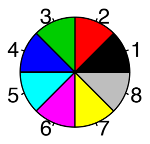


Figure 1.30: Basic R colours.

These colour choices date back from 1980s hardware, where graphics cards handled colours by letting each pixel either fully use or not use each of the three basic colour 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[13] The colours in Figure 30 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 colours used by some of the more modern visualisation oriented packages (including *ggplot2*) are much better already, but sometimes we want to make our own choices.

[13] Thus the 8^{th} colour should be white; in R, whose basic infastructure was put together when more sophisticated graphics display were already available, this was replaced by grey, as you can see in Figure 30.

In Section 0.8 we saw the function `scale_fill_gradientn`, which allowed us to create the colour gradient used in Figures 24 and 25 by interpolating the basic colour palette defined by the function `brewer.pal` in the *RColorBrewer* package. This package defines a great set of colour palettes, we can see all of them at a glance by using the function `display.brewer.all` (Figure 31).

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

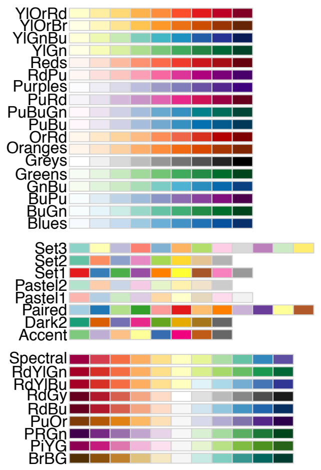


Figure 1.31: RColorBrewer palettes.

We can get information about the available colour palettes from the *data.frame* `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 33.

To obtain the colours from a particular palette we use the function `brewer.pal`. Its first argument is the number of colours 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 colours (for example so we can plot a heatmap with continuous colours) we can use the `colorRampPalette` function command to interpolate any of the *RColorBrewer* presets – or any set of colours:

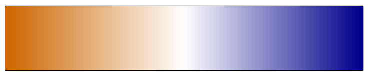


Figure 1.32: A quasi-continuous colour palette derived by interpolating between the colours `darkorange3`, `white` and `darkblue`.

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

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

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

1.9.1 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 particularly versatile and produces good-looking output is the function `pheatmap` from the eponymous package. 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* colour palette from *RcolorBrewer* in conjuction with the `colorRampPalette` function to interpolate the 11 colour into a smooth-looking palette (Figure 33).

```r
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 = groupColour,
    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
)
```

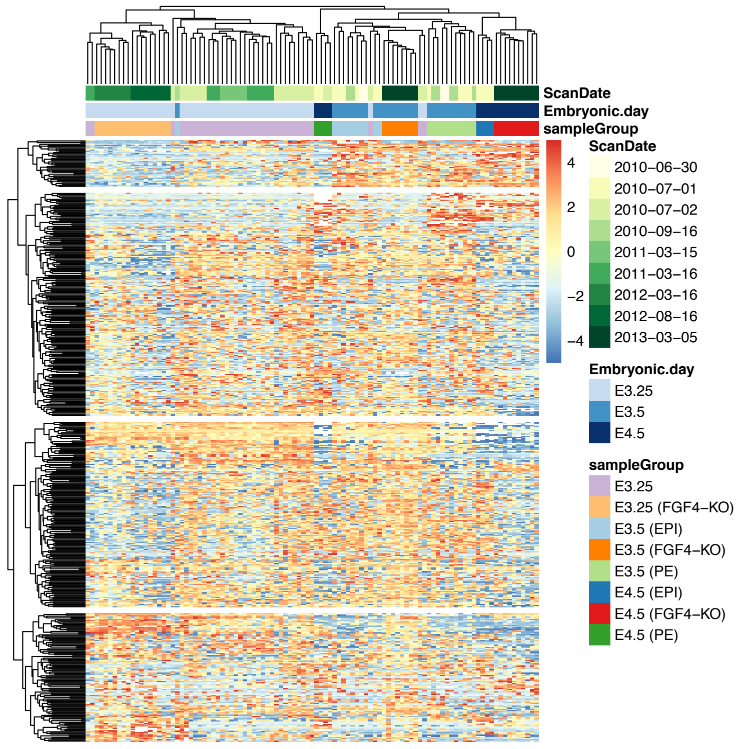


Figure 1.33: A heatmap of relative expression values, i. e., $\log_2$ fold change compared to the average expression of that gene (row) across all samples (columns). The colour scale uses a diverging palette, whose neutral midpoint is at 0.

Let us take a minute to deconstruct the 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 coloured bars on top of the heatmap. There is also a similar `annotation_row` argument, which we haven't used here, for coloured bars at the side. `annotation_colors` is a list of named vectors by which we can override the default choice of colours 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 visualisations, it's worth studying them.

1.9.2 Colour spaces

Colour perception in humansvon Helmholtz (1867) is three-dimensional[14]. 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 0.5, where we printed out the contents of `groupColour`:

[14] Physically, there is an infinite number of wave-lengths of light, and an infinite number of ways of mixing them.

```
groupColour[1]

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

Here, CA is the hexadecimal representation for the strength of the red colour channel, B2 of the green and D6 of the green colour 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.

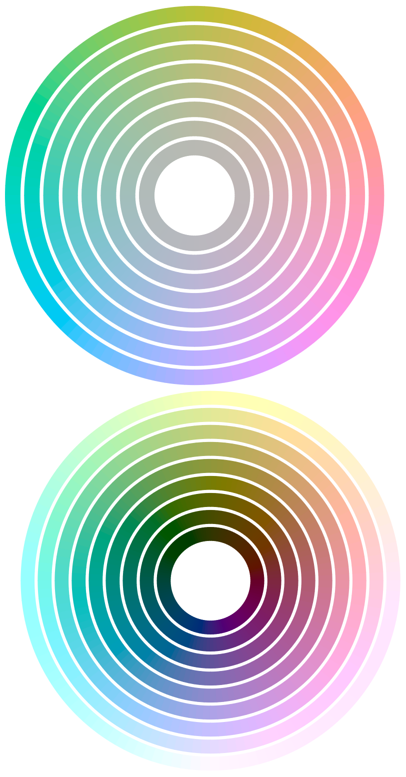


Figure 1.34: Circles in HCL colorspace. Upper panel: The luminosity L is fixed to 75, while the angular coordinate H (hue) varies from 0 to 360 and the radial coordinate $C = 0, 10, \ldots, 60$. Lower panel: constant chroma $C = 50$, H as above, and varying luminosity $L = 10, 20, \ldots, 90$.

The R 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[15] 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 coloured areas look bigger than dark ones. Our attention also tends to get drawn to loud colours, and fixing the value of chroma makes the colors equally attractive to our eyes.

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

There are many ways of choosing colours from a colour wheel. *Triads* are three colours chosen equally spaced around the colour wheel; for example, $H = 0, 120, 240$ gives red, green, and blue. *Tetrads* are four equally spaced colours around the colour wheel, and some graphic artists describe the effect as "dynamic". *Warm colours* are a set of equally spaced colours close to yellow, *cool colours* a set of equally spaced colours close to blue. *Analogous colour* sets contain colours from a small segment of the colour wheel, for example, yellow, orange and red, or green, cyan and blue. *Complementary colours* are colours diametrically opposite each other on the colour wheel. A tetrad is two pairs of complementaries. *Split complementaries* are three colours 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).

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 colours with low to moderate chromatic content are usually more pleasant.

1.10 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 colour scales. In the plots of this chapter that involved the microarray data, we used the logarithmic transformation[16] – not only in scatterplots like Figure 21 for the x and y-coordinates, but also in Figure 33 for the colour 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: $\log(ax) = \log a + \log x$.

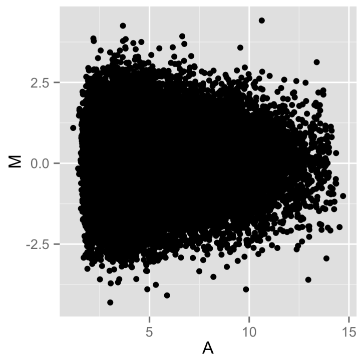
[16] We used it implicitly since the data in the *ExpressionSet* object x already come log-transformed.

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 35, it is easy to take away the impression that the distribution of `M` depends on `A`, with higher variances for low `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`.

```r
A <- exprs(x)[,1]
M <- rnorm(length(A))
qplot(A, M)
qplot(rank(A), M)
```

Question **1.10.1** *Can the visual artefact be avoided by using a density- or binning-based plotting method, as in Figure 25?*

Question **1.10.2** *Can the rank transformation also be applied when choosing colour scales e. g. for heatmaps? What does* histogram equalization *in image processing do?*



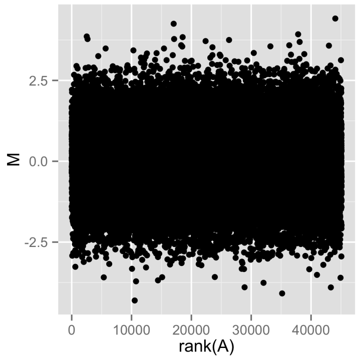


Figure 1.35: The effect of rank transformation on the visual perception of dependency.

1.11 Saving Figures

Just as important as plotting figures is saving them for later use.

ggplot2 has a built-in plot saving function called `ggsave`, which if run by itself defaults to saving the last plot you made with the size of the graphics device that it was/is open in.

```
ggsave("myplot1.png")
```

Or you can specify a particular plot that you want to save, say, the sunspot plot from earlier.

```
ggsave("myplot2.pdf", plot = sp)
```

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 in principle limited by the resolution of the original file. And you don't want the figures in your talks or papers look poor because of pixelisation artefacts!

1.12 Biological data with ggbio

fixme: Expand this section... do something more interesting

A package that can be really useful for biological data is an offshoot of *ggplot2* made for biology-specific plots called *ggbio*. One example is plotting and highlighting **ideograms**[17]

[17] The term ideogram generally means a graphic that represents a concept, specifically in biology we usually are referring to plots of the chromosome, like in Figure 36.

Load the ideogram cytoband information for the hg19 build of the human genome

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

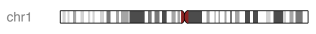


Figure 1.36: Chromosome 1 of the human genome: ideogram plot.

1.13 Recap of this chapter

- You have had an introduction to the base plotting functions in R. They are widely used and can be convenient for quick data exploration.
- You should now be comfortable making beautiful, versatile and easily extendable plots using *ggplot2*'s `qplot` or `ggplot` functions.
- 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 wats
- Now you are prepared to explore *ggplot2* and plotting in general on your own.

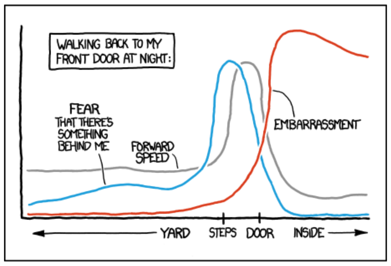
1.14 Exercises

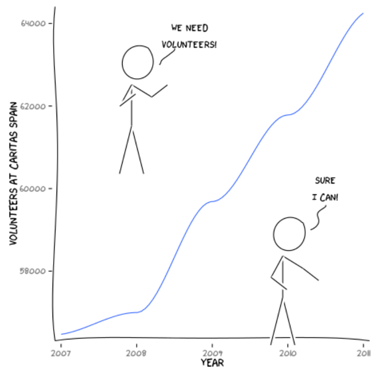
Exercise **1.1 (themes)** *Explore how to change the visual appearance of plots with themes. For example:*

```r
qplot(1:10,1:10)
qplot(1:10,1:10) + theme_bw()
```

Exercise **1.2 (colour names in R)** *Have a look at* `http://research.stowers-institute.org/efg/R/Color/Chart`

Exercise **1.3 (ggxkcd)** *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`.


WALKING BACK TO MY FRONT DOOR AT NIGHT:
FEAR THAT THERE'S SOMETHING BEHIND ME
FORWARD SPEED
EMBARRASSMENT
YARD
STEPS
DOOR
INSIDE



1.15 Session Info

- R version 3.2.0 (2015-04-16), `x86_64-apple-darwin14.3.0`
- Locale: `en_US.UTF-8/UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8`
- Base packages: base, datasets, graphics, grDevices, grid, methods, parallel, stats, stats4, utils
- Other packages: affy 1.46.0, annotate 1.46.0, AnnotationDbi 1.30.1, beeswarm 0.2.0, Biobase 2.28.0, BiocGenerics 0.14.0, boot 1.3-16,

clue 0.3-49, cluster 2.0.1, colorspace 1.2-6, DBI 0.3.1, dplyr 0.4.1, extrafont 0.17, Formula 1.2-1, fortunes 1.5-2, genefilter 1.50.0, geneplotter 1.46.0, GenomeInfoDb 1.4.0, ggbio 1.16.0, ggplot2 1.0.1, ggthemes 2.1.2, gplots 2.17.0, gridExtra 0.9.1, gtools 3.5.0, hexbin 1.27.0, Hiiragi2013 1.4.0, Hmisc 3.16-0, IRanges 2.2.3, KEGGREST 1.8.0, knitr 1.10.5, lattice 0.20-31, MASS 7.3-40, mouse4302.db 3.1.2, org.Mm.eg.db 3.1.2, pheatmap 1.0.2, RColorBrewer 1.1-2, reshape2 1.4.1, RSQLite 1.0.0, S4Vectors 0.6.0, showtext 0.4-2, survival 2.38-1, sysfonts 0.5, xkcd 0.0.4, XML 3.98-1.2, xtable 1.7-4

- Loaded via a namespace (and not attached): acepack 1.3-3.3, affyio 1.36.0, assertthat 0.1, BiocInstaller 1.18.2, BiocParallel 1.2.2, biomaRt 2.24.0, Biostrings 2.36.1, biovizBase 1.16.0, bitops 1.0-6, BSgenome 1.36.0, caTools 1.17.1, codetools 0.2-11, dichromat 2.0-0, digest 0.6.8, evaluate 0.7, extrafontdb 1.0, foreign 0.8-63, formatR 1.2, futile.logger 1.4.1, futile.options 1.0.0, gdata 2.16.1, GenomicAlignments 1.4.1, GenomicFeatures 1.20.1, GenomicRanges 1.20.4, GGally 0.5.0, graph 1.46.0, gtable 0.1.2, highr 0.5, httr 0.6.1, KernSmooth 2.23-14, labeling 0.3, lambda.r 1.1.7, latticeExtra 0.6-26, lazyeval 0.1.10, magrittr 1.5, munsell 0.4.2, nnet 7.3-9, OrganismDbi 1.10.0, plyr 1.8.2, png 0.1-7, preprocessCore 1.30.0, proto 0.3-10, RBGL 1.44.0, Rcpp 0.11.6, RCurl 1.95-4.6, reshape 0.8.5, rpart 4.1-9, Rsamtools 1.20.4, rtracklayer 1.28.4, Rttf2pt1 1.3.3, scales 0.2.4, showtextdb 1.0, splines 3.2.0, stringi 0.4-1, stringr 1.0.0, tools 3.2.0, VariantAnnotation 1.14.1, XVector 0.8.0, zlibbioc 1.14.0

Bibliography

Daniel B Carr, Richard J Littlefield, WL Nicholson, and JS Littlefield. Scatterplot matrix techniques for large N. *Journal of the American Statistical Association*, 82(398):424–436, 1987.

W. S. Cleveland, M. E. McGill, and R. McGill. The shape parameter of a two-variable graph. *Journal of the American Statistical Association*, 83:289–300, 1988.

Ross Ihaka. Color for presentation graphics. In Kurt Hornik and Friedrich Leisch, editors, *Proceedings of the 3rd International Workshop on Distributed Statistical Computing*. Vienna, Austria, 2003.

R. A. Irizarry, B. Hobbs, F. Collin, Y. D. Beazer-Barclay, K. J. Antonellis, U. Scherf, and T. P. Speed. Exploration, normalization, and summaries of high density oligonucleotide array probe level data. *Biostatistics*, 4(2):249–264, 2003.

J. Mollon. Seeing colour. In T. Lamb and J. Bourriau, editors, *Colour: Art and Science*. Cambridge Unversity Press, 1995.

Y. Ohnishi, W. Huber, A. Tsumura, M. Kang, P. Xenopoulos, K. Kurimoto, A. K. Oles, M. J. Arauzo-Bravo, M. Saitou, A. K. Hadjantonakis, and T. Hiiragi. Cell-to-cell expression variability followed by signal reinforcement progressively segregates early mouse lineages. *Nature Cell Biology*, 16(1):27–37, 2014.

H. von Helmholtz. *Handbuch der Physiologischen Optik*. Leopold Voss, Leipzig, 1867.

Hadley Wickham. *ggplot2: elegant graphics for data analysis*. Springer New York, 2009. ISBN 978-0-387-98140-6. URL `http://had.co.nz/ggplot2/book`.

Hadley Wickham. A layered grammar of graphics. *Journal of Computational and Graphical Statistics*, 19(1):3–28, 2010.

Hadley Wickham. Tidy data. *Journal of Statistical Software*, 59(10), 2014.

L. Wilkinson. Dot plots. *The American Statistician*, 53(3):276, 1999.

L. Wilkinson and G. Wills. *The Grammar of Graphics*. Springer, 2005.