Plotting Complex Figures Using R

Version 1.1

Licence

Table of Contents

Introduction

One of the biggest attractions to the R programming language is that built into the language is the functionality required to have complete programmatic control over the plotting of complex figures and graphs. Whilst the use of these functions to generate simple plots is very easy, the generation of more complex multi-layered plots with precise positioning, layout, fonts and colours can be challenging. This course aims to go through the core R plotting functionality in a structured way which should provide everything you would need to create plots of any kind of data you may encounter in the future. Although the course is mostly restricted to the plotting functions of the core R language we will also introduce a few very common add-in packages which may prove to be useful.

Graphing Libraries in R

In this course we will deal mostly with the plotting functions built into the core R language. These cover all of the main graph types you're likely to use and will be available on any installation of R. These libraries are not the only way to do plotting in R though – there are other graphing libraries which can be imported and each of these has advantages and disadvantages. The most common external library is `ggplot`, and this isn't covered in this course – we have a whole separate course just on `ggplot` if you'd like to learn about that.

The R Painter's Model

Plotting in R operates on what is known as a 'painter's model'. Complex plots in core R are not drawn in a single operation, but require several different plotting layers to be laid down on top of each other. Newly added layers are placed over the top of whatever has already been drawn, and by default will obscure anything which was plotted underneath them. The addition of new layers is a one-way operation – once a layer has been drawn there's no way to remove or modify it, so if you find you've made a mistake in a lower layer the only way to correct it is to start over again from the bottom of the stack.
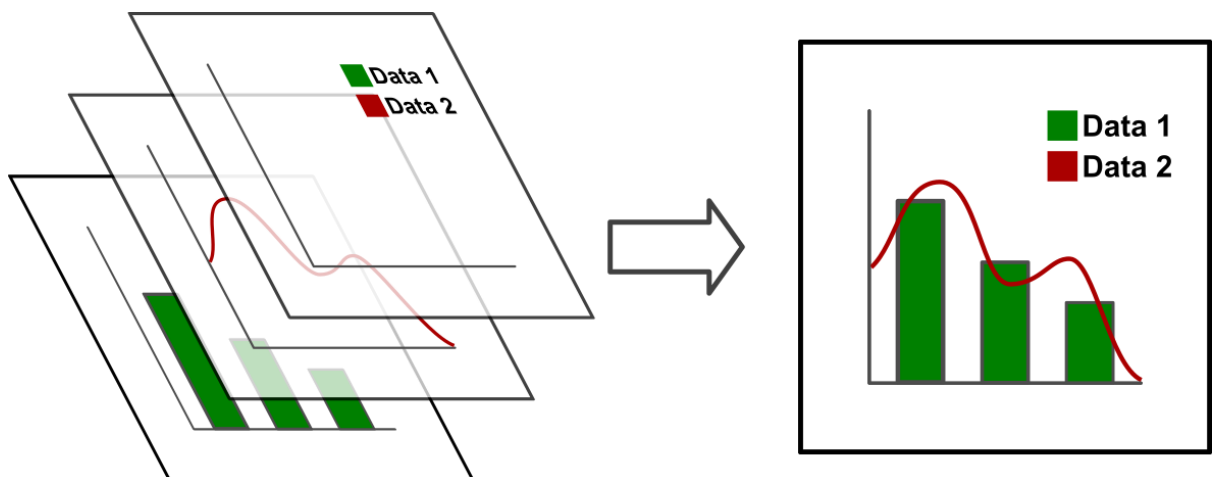
The power of this approach is that you can define your own complex graph types by combining simple graph types in any combination, and there is very little that you can't do in terms of plotting using the core R plotting functions.

Core Graph Types and Extensions

There are a number of built-in plotting functions in R, but in general they split into 2 main types. When drawing a complex plot the usual process is that you start by drawing a complete plot area. As well as drawing a graph this also sets the scaling used for the x and y coordinates which are then used for layers subsequently placed on top. Most commonly, the first plot drawn defines the entire plot area and the axis scales used by subsequent layers.

Core Graph Types

The R language defines a number of different plotting functions which draw different styles of plot. Each of the functions below defines a primary plot which draw a plot area (axes, scales, labels etc) as well as representing some data. All of these functions have many options which can dramatically change the appearance of the plot, and we'll look at these later so even if there isn't a plot which immediately looks the way you want, you can probably find one which can be adapted.
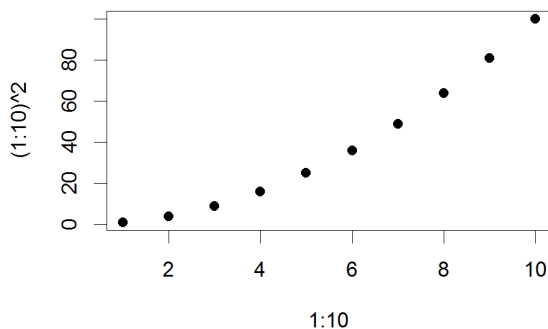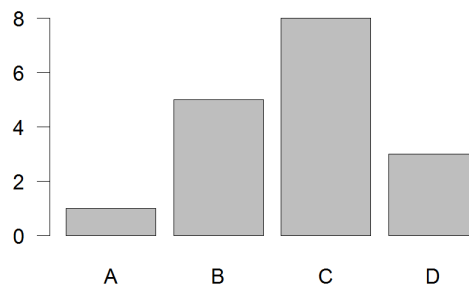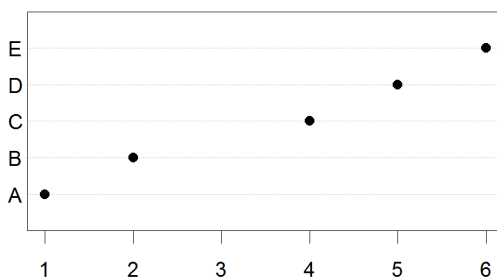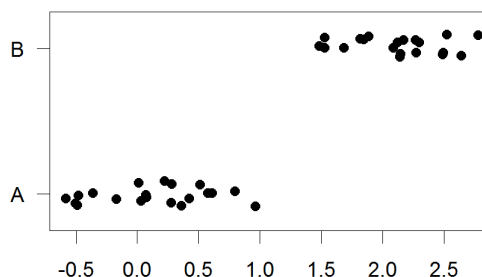
Histogram
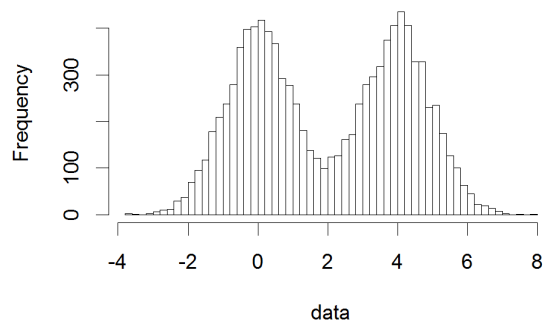
Boxplot
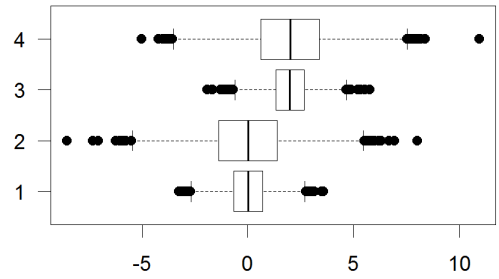
Pie Chart
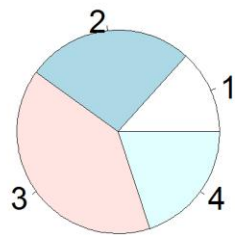
Smooth Scatter
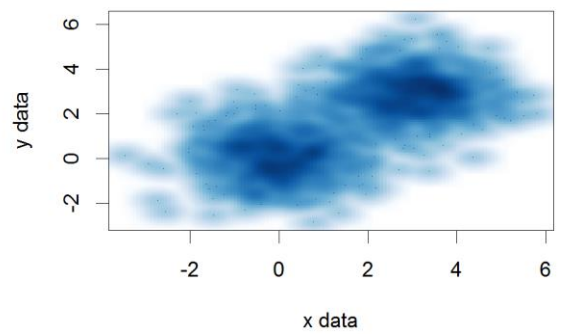
Control of Graph Appearance

Internal Graph Options

When drawing a plot it is very common to alter the default appearance to show exactly what you want to see. As with all other R functions you can therefore modify the action of the function by passing in additional options as key=value pairs. Whilst each plot type will have some plot specific options, a lot of the options you can pass end up being common to most plot types. You can see the full set of options for a given function on its help page (run `?functionname`), but we'll go through some of the common options below.

Axis Scales

Most plot types will auto-scale their axes to accommodate the range of data you initially supply.   There are cases though where you need to manually specify the scale, either to expand or contract it to better show the data, or, if you're going to add more layers afterwards to ensure that the axis range is sufficient to accommodate all of the data you're later going to plot – remember that you can't modify a layer once you've drawn it.

Adjustment of the range of axes is pretty universally provided by the options `xlim` and `ylim`, both of which require a vector of 2 values (min,max) to specify the range for the axis. Rather than specifying these values manually you can also use the output of the `range` function to calculate it. Remember that the scales are set by the data you initially pass to the base plot, so if you want to add further layers with a wider range later then you need to ensure that the ranges you set when drawing the initial plot are wide enough to accommodate all of the data you later intend to plot.

Axis Labels

Axis labels for the x and y axes can be set using the `xlab` and `ylab` options. These are different to the labels used to specify category labels for specific datasets.

Plot title / subtitle

Whilst titles can be added to plots as an overlay, you can normally add a main title using the `main` option, and a subtitle using the `sub` option.

Plotting character

For plots which use a specific character to indicate a data point you can choose from a range of different point shapes to use. These are set using the `pch` (point character) option and take a single number as its value. The shapes which correspond to each number are shown below.

Plot Characters

✕ 4	◆ 9	◹ 14	● 19	△ 24
+ 3	✳ 8	⊗ 13	◆ 18	◇ 23
△ 2	⊠ 7	⊞ 12	▲ 17	▢ 22
○ 1	▽ 6	✡ 11	● 16	○ 21
□ 0	◇ 5	⊕ 10	■ 15	● 20
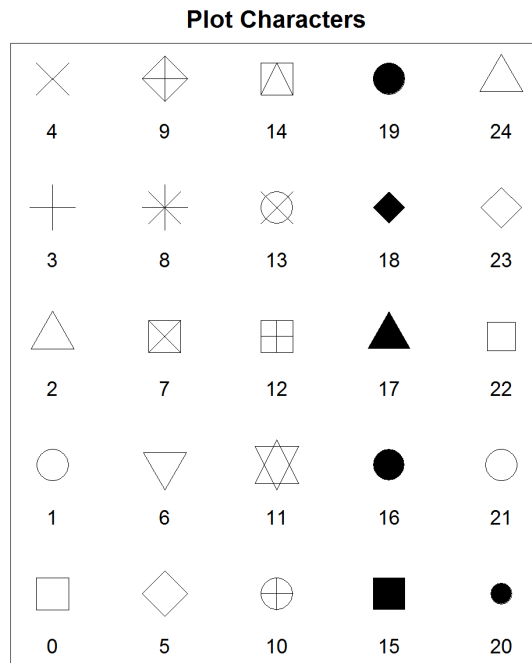
The size of the plot characters can be changed using the `cex` (character expansion) option. This is a value where 1 is the default size and other values indicate the proportion of the default size you want to use.

cex sizes

0.25	0.5	0.75	1	2	4	8
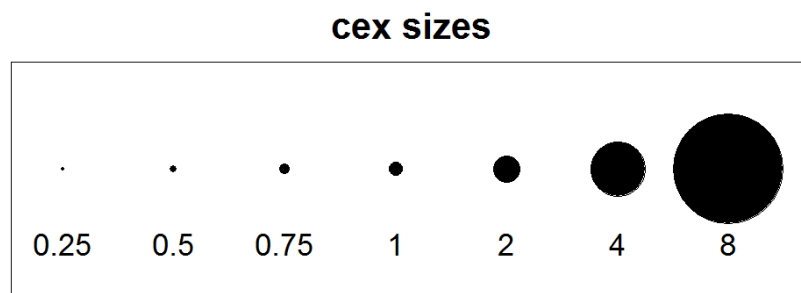
Plot Specific Options

For each plot type there are a range of specific options which can be added to the required arguments to change the appearance of the plot. In the section below we will go through the different plot types, describe the format(s) of data they accept and what the most common supplementary options they use are.

plot()

The `plot` function is the generic x-y plotting chart type. Its default appearance is as a scatterplot, but by changing the way the points are plotted and joined you can make it into a line graph or a point and line graph.

Scatter plot   **Line graph**   **Dot and line graph**
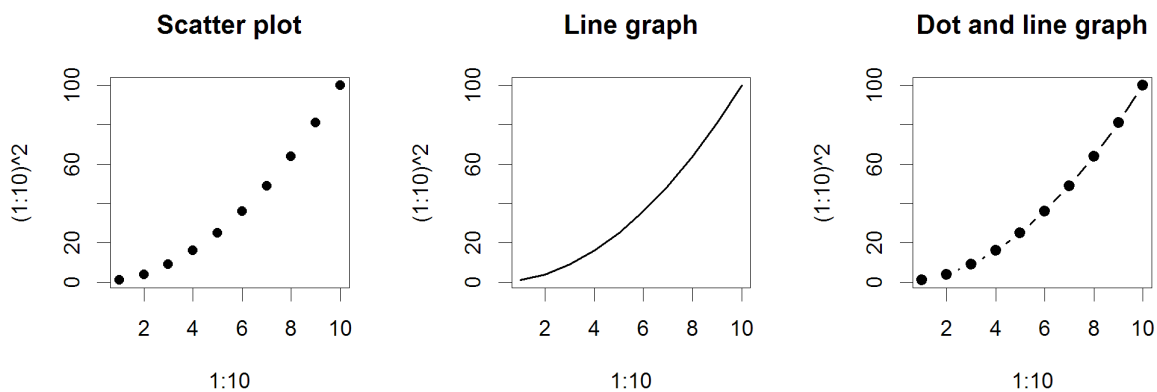
Options:

- `plot` is hugely flexible and can take in almost any kind of data. Most custom data types will override plot to do something sensible with the data. For the common uses of `plot` you'd pass in either 2 vectors of values (x and y), or a data frame or matrix with 2 columns (x and y)
- The `type` option determines what sort of plot you get
  - `type="l"` gives you a line plot
  - `type="p"` gives you a scatter plot (the default)
  - `type="b"` gives you a dot and line plot
  - For line plots `lwd` sets the line width (thickness)
  - For line plots `lty` sets the line type
    1. Solid line
    2. Dashed line
    3. Dotted line
    4. Dot and dash line
    5. Long dash line
    6. Long then short dash line

barplot()

`barplot` is used to construct bar graphs. By varying the options applied you can create standard bar graphs, stacked bar graphs, or have groups of side by side bars. The graphs can be constructed either horizontally or vertically.

Bar Chart   **Stacked Bar Chart**   **Grouped Bar Chart**
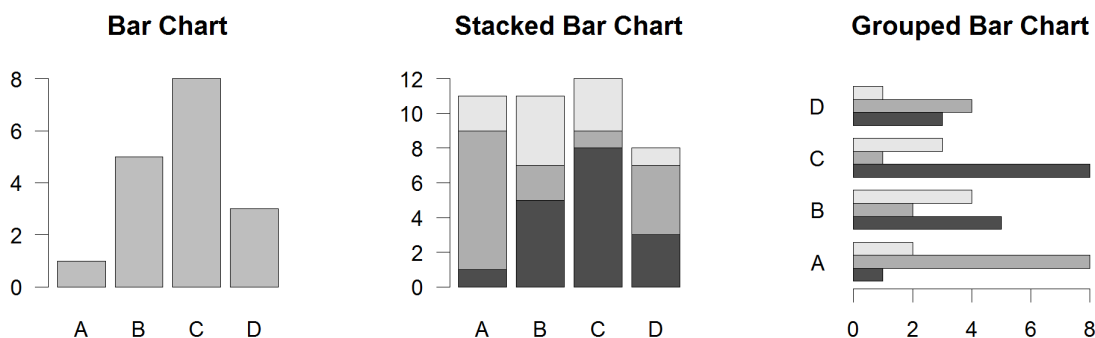
Options:

- barplot can either be given a simple vector of values for a single data series, or it can be given a matrix of values for sets of linked data series
- The names.arg option is used to set the category names. These will also be taken from the names associated with the data vector if it has them
- If you pass a matrix of data then you can set the beside argument to TRUE to have the data series grouped together on a common baseline, rather than presented as a stacked chart.
- If you want your bars to run horizontally then you set horiz to TRUE.
- You can alter the spacing between bars using the space argument. This can either be a single value to say what spacing separates all bars, or it can be one value per bar. If you have grouped bars then you can pass a vector of two values, the first for the between group space and the second for the within group space.

dotchart()
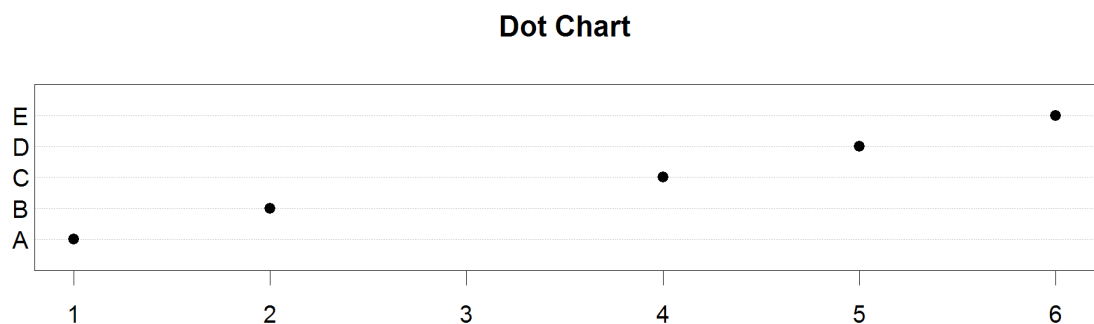
Dot Chart

Options:

- dotchart will normally be given a simple vector of values for a single data series. You can pass it a matrix in which case it generates a facetted set of independent dot charts.
- The labels option is used to set the category names. These will also be taken from the names associated with the data vector if it has them.

stripchart()

Stripcharts are a way to show all individual measures for a set of related data points along with summarised values such as a mean. They are a better way of representing small datasets whose distribution doesn't conform well enough to a standard distribution to be able to be accurately summarised by a boxplot or similar graph.

Stripchart
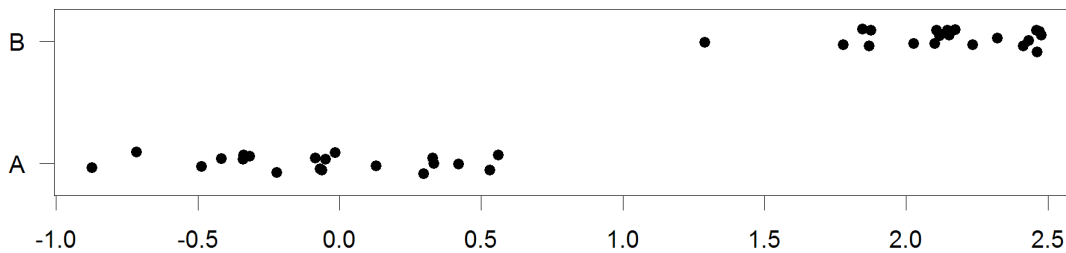
Options:

- Stripchart's data is normally passed in as a list of numeric vectors containing the different series you want plot. This can be a data frame if the sets are the same size (since data frames are lists anyway). You can also use the `data ~ factor` notation to split a single dataset by a factor.
- You can use the method option to say what happens when multiple data points are the same. The default is that they plot over the top of each other, but you normally want to use `method="jitter"` to spread them out horizontally. You can specify the numeric `jitter` argument to say how spread you want them to be.
- By default the plots are horizontal. You can make them vertical by using the `vertical` argument.
- Group names are taken from the names of the list slots if present, but you can specify them manually using `group.names`.

hist()

The `hist` function draws histograms which bin your data and then count the occurrences of data points within each bin. It is a natural way to summarise the distribution of a single continuous dataset.

Histogram
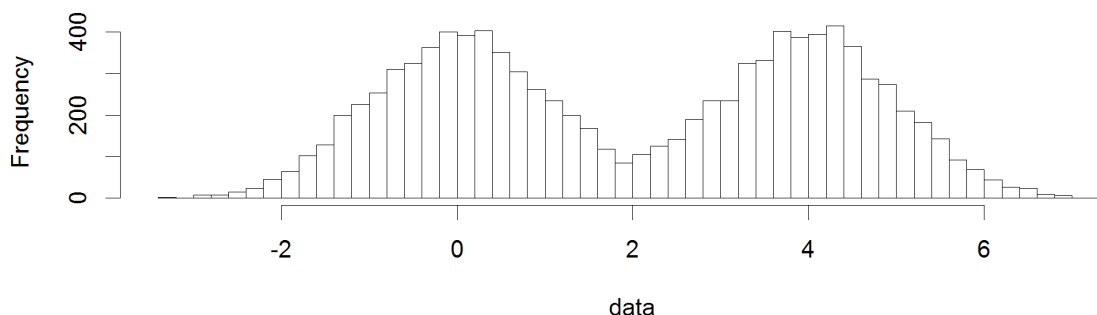
Options:

- The `hist` function only takes a single vector of values for its data.
- To control the number of categories into which the data is broken you use the `breaks` argument. Most simply this is a single value indicating the total number of categories but you can also provide a vector of values between which the different categories will be constructed. If you're adventurous you can also pass in a function which will calculate the breakpoint vector rather than passing the vector in directly.

- By default the plot shows the frequency with which data falls into each category, and the y axis is the number of observations. You can set the `probability` argument which turns the y values into proportions of the data, such that the sum of the values is 1.
- If you want to put text above each of the bars then you can pass a character vector in the `labels` argument.

boxplot()

A boxplot is a traditional way to summarise the major properties of a uni-modal distribution. It plots out the median, interquartile range and makes an estimate of the ends of the distribution and shows individual outliers which fall beyond this. Whilst it is a useful shortcut it has mostly been superseded by more accurate representations such as the violin plot or beanplot.
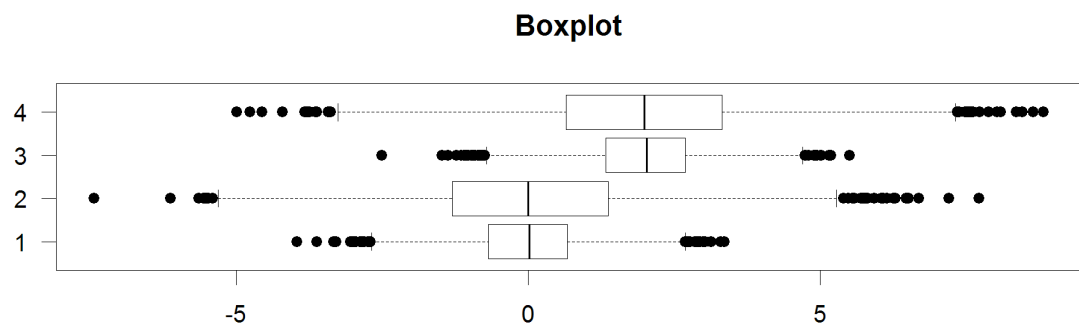
Options:
- Data for `boxplot` can either be a single numeric vector, a list of numeric vectors or a formula of the form `values ~ factor`.
- The sensitivity of the whiskers on the plot is controlled by the `range` argument which specifies the maximal value for these in terms of a multiple of the inter-quartile range. Commonly used values would be 1.5 for outliers, 3 for extreme outliers, and 0 for the limits of the data values.
- The widths of the boxes can be controlled by the `width` argument. You can make the widths proportional to the number of observations by using the `varwidth` option.
- Group names are taken from the supplied dataset but can be overridden with the `names` argument.
- The plot is vertical by default but can be plotted sideways by setting the `horizontal` argument.

pie()

The `pie` function draws a pie chart to summarise the categorical division of a dataset into subgroups.
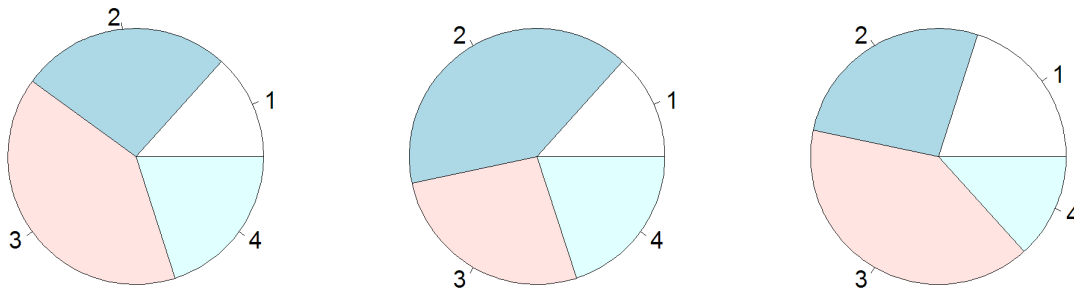
Options:
- Data must be a numeric vector of positive values.
- Labels are taken from the data names but can be overridden by the `labels` argument. If the labels are big then you might need to reduce the size of the pie by setting the `radius` argument to a lower value.
- Categories are drawn anti-clockwise by default but you can set `clockwise` to make them go the other way.
- You can change the angle at which the first category starts using the `angle` argument.

smoothScatter()

The `smoothScatter` plot is a variant of the usual x-y plot function and is useful in cases where there are too many points in the plot to be able to accurately represent each one. In the `smoothScatter` plot instead of plotting individual points shaded areas are constructed to illustrate the density of points falling into each part of the plot allowing for an intuitive visualisation of very large datasets.
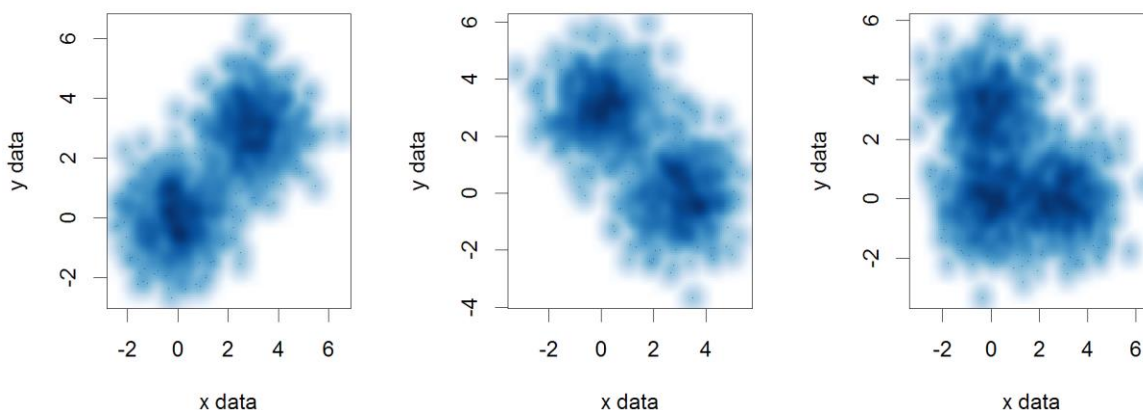
Options:
- Data can be a pair of vectors (of the same length), a list of length 2, or a matrix with 2 columns.
- You can use the `nbin` argument to specify the number of bins the data range is split into for the density calculation, this can be a vector of length 2 if you want to set it separately for x and y.
- For colour density you can use the `colramp` argument to provide a function which takes an integer value as an argument and returns that number of colours. You can also provide a custom function to map densities to colours using the `transformation` option.

Common Par Options

As well as the options passed to the individual plotting functions there is also a global set of plotting options managed by the par function. These control high level attributes such as margins and spacing, font sizes and orientation and line styles.

The `par` function manages all of these parameters which are held in a global list. You can edit the values stored in this list which will affect all of the plots you draw within the same plotting session.

Editing the `par` values is done with the `par` function itself. You can pass in key=value pairs to set any of the `par` parameters to a new value. When you do this the return value is a list containing the values which were replaced. This allows you to re-run par on the output of the previous run to restore all parameters to their original values.

For example, if we wanted to set character expansion (`cex`) to 1.5 we could first check what the current value was:

```
par()$cex
[1] 1
```

We could set a new value, and store the old one.

```
par(cex=1.5) -> old.par
par()$cex
[1] 1.5
```

We can then later restore the original value.

```
par(old.par)
par()$cex
[1] 1
```

Margins and spacing

Margins in the plot area are set using the `mar` or `mai` parameters, which can be set either from `par` or within the plotting function. The only difference between these different parameters is the units they use (`mar` = number of lines, `mai` = inches) you can set the conversion between `mar` and `mai` using the `mex` parameter.

When setting the margins you supply a vector of 4 values which represent the **bottom**, **left**, **top** and **right** margin sizes you would like. If you decrease your margins you need to be careful that you don't push plot elements (axis labels for example) off the plot area.

If you increase your margins too much you will sometimes find that there is insufficient space left for the required plot area, and you'll get an error like:

```
Error in plot.new() : figure margins too large
```
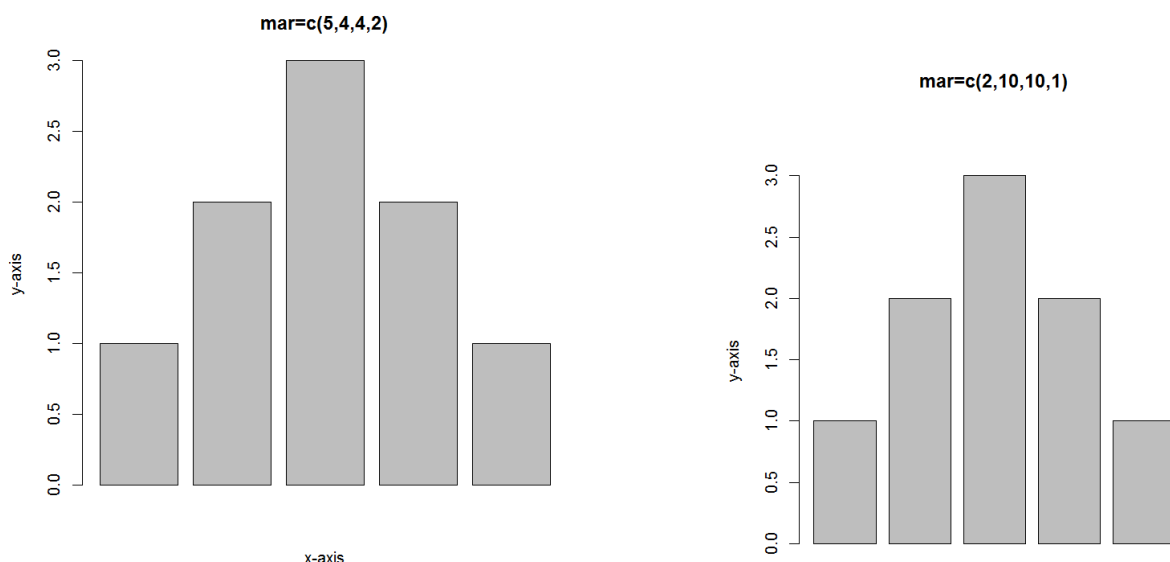
Fonts and Labels

There are various options you can set which will affect the appearance of fonts and labels within your plots.

The simplest change you can make is to the size at which the various labels within the plot will be drawn. This can be changed using one of the `cex` (character expansion) parameters. The main `cex` parameter is a proportional value relative to 1 (the default size) and affects all text and plot characters. You can do more targeted changes by altering `cex.axis` (axis tick labels) `cex.lab` (axis titles), `cex.main` (plot title) or `cex.sub` (plot subtitle). You should also note that, confusingly, there is a `cex` parameter which can be passed to the `plot` function which only affects the plotting characters, and doesn't affect the fonts, which are only changed by altering this parameter through `par`.
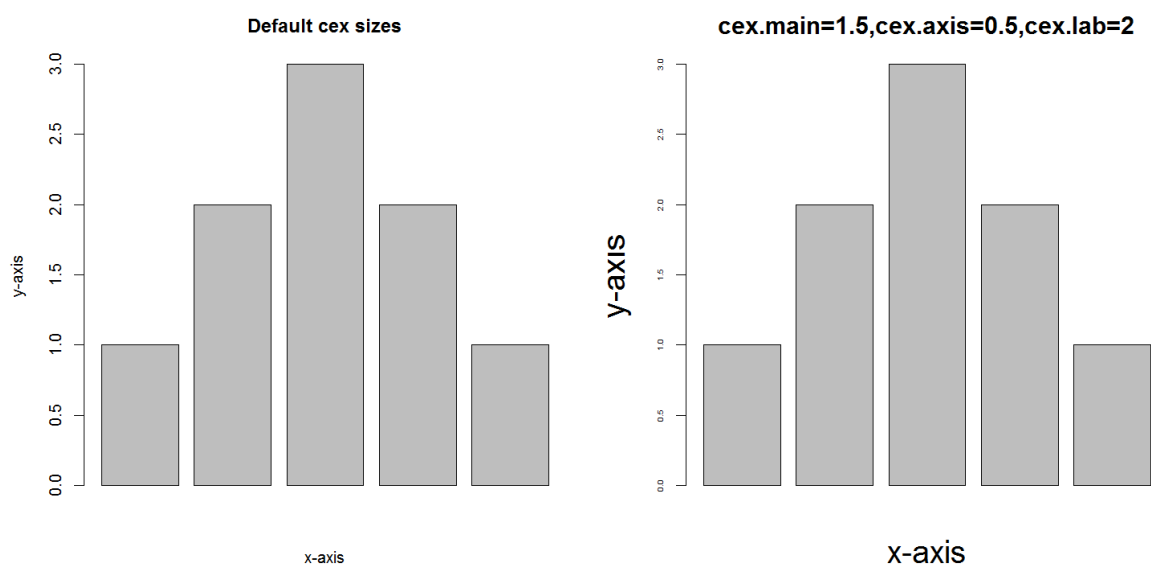
As well as changing the size of your labels you can also change the font which is used to draw them. R offers 4 standard font styles which can be assigned to the different plot elements. These are:

1. Plain text (the default)
2. Bold text
3. Italic text
4. Bold italic text

These can be assigned to the `font` parameter to change all fonts in a plot, or to one of its sub-parameters, `font.axis` (axis tick labels), `font.lab` (axis titles), `font.main` (plot title) or `font.sub` (plot subtitle).

In addition to changing the style of the font you can also change between 3 standard font families using the `family` parameter. These are `sans` (plain sans-serif font – the default and what you should nearly always use), `serif` (a serif font similar to Times Roman – shouldn't generally be used for plots), or `mono` (monospaced font where all letters are the same width. Can be useful if including DNA or Protein sequence tags in a plot).

Finally you can choose the orientation of your labels by using the `las` parameter. This takes a numeric value whose meaning is:

0. Always parallel to the axis (the default)
1. Always horizontal (usually the easiest to read)
2. Always perpendicular to the axis
3. Always vertical
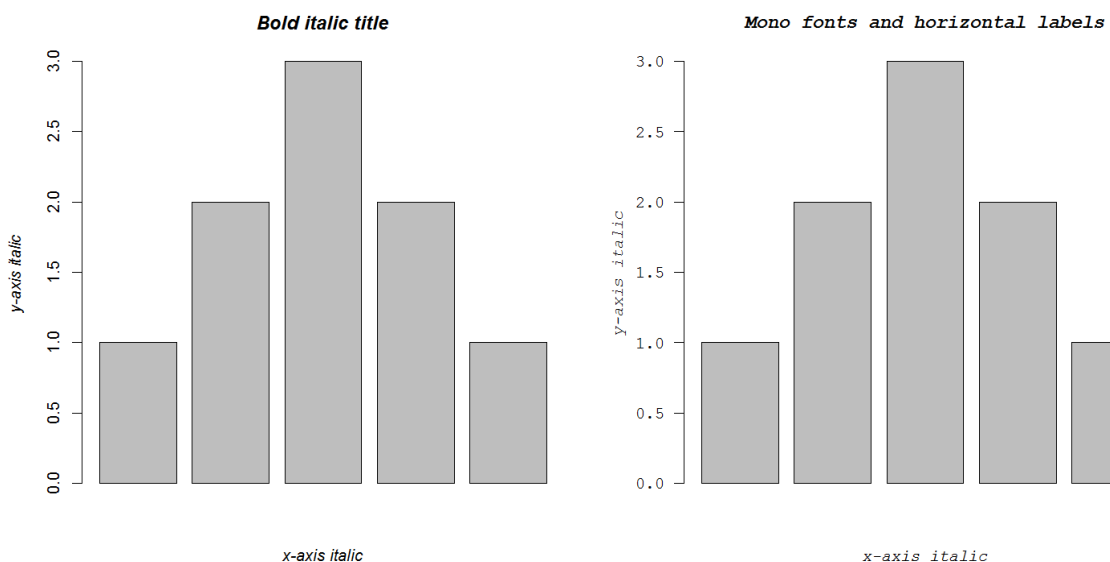
Multi-panel plots

One of the most common things to change in the `par` system is the overall layout of the plot area. Rather than having your `plot` command draw a graph which takes up the entire plot area you can divide this into a grid of sub-areas where each new `plot` command moves to the next sub-area until the entire plot area is full, at which point the next plot wipes the whole area and starts over again.

You can set the plot area to be divided in this way using the `mfrow` parameter. This takes a vector of two values (rows, columns) to say how you want the division to occur.
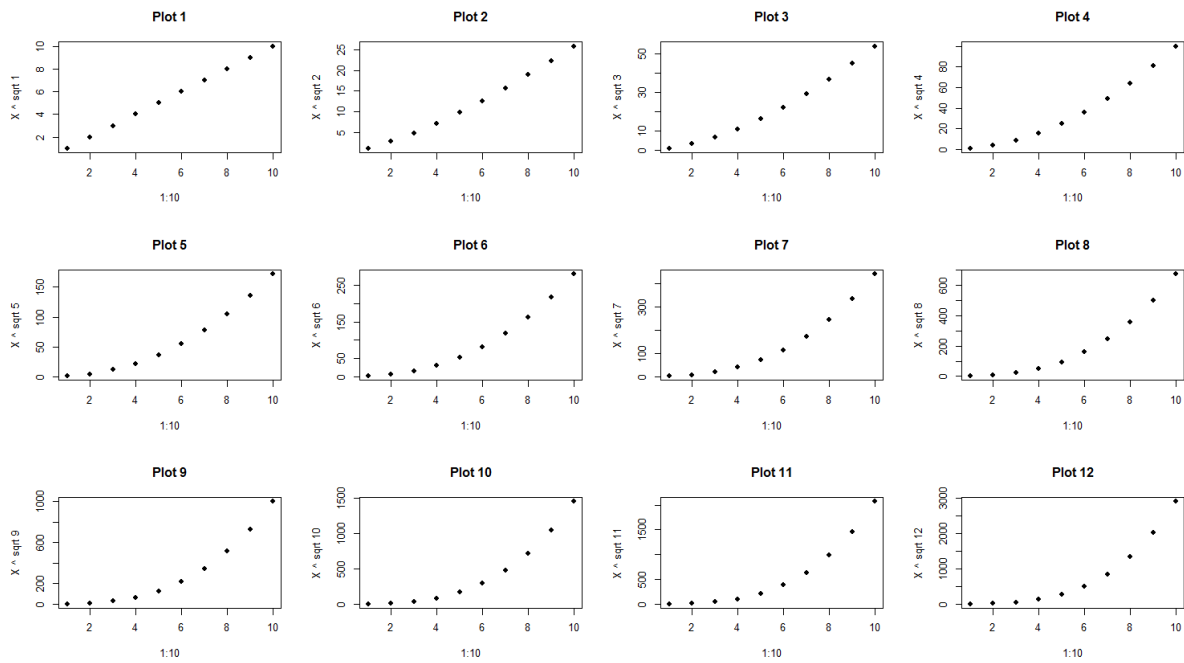
Using Colour

Most plot types in R allow you some flexibility over the use of colour. Most commonly you will provide either a custom vector of colours to use, or you can use a common colour scheme to which your samples will be matched.

In general there are two ways of specifying colours, either you can describe an individual colour, or you can provide a function which can generate a custom set of colours dynamically.

Specifying colours

Colours can be manually specified either through a direct RGB hexadecimal string or through the use of a controlled set of colour names defined within the language. You can see the predefined colours by running `colors()`. If you want to manually define a colour then you can do this using hexadecimal notation where you can specify either 3 or 4 values in a single string. The first 3 required values are the amount of red, green and blue in hex values between 0 (`00`) and 255 (`FF`). The optional fourth value would be the alpha (transparency) value.

For example:

- Pure red would be `#FF0000`
- Pure blue would be `#0000FF`
- Dark green would be `#007F00`
- Dark purple would be `#CC00CC`
- Semi-transparent yellow would be `#FFFF0044`

To convert to hexadecimal you can use `as.hexmode()` in R.

Built-in colour schemes

Unless it's for a very simple application most of the time it doesn't make any sense to try to define your own colour palettes to use in R. Defining a good palette with good colour separation and equal visual impact is not a trivial task and it's better to use a pre-defined palette rather than make up your own.

There are some colour palettes built into the core R language. These are generally semi-quantitative schemes which can be useful but maybe aren't the best schemes available. The built-in colour palette functions are:

- `rainbow`
- `heat.colors`
- `cm.colors`
- `terrain.colors`
- `topo.colors`

Each of them simply takes the number of colours you want to generate as an argument.

Rainbow Colours

Heat Colours

CM Colours

Terrain Colours

Topo Colours
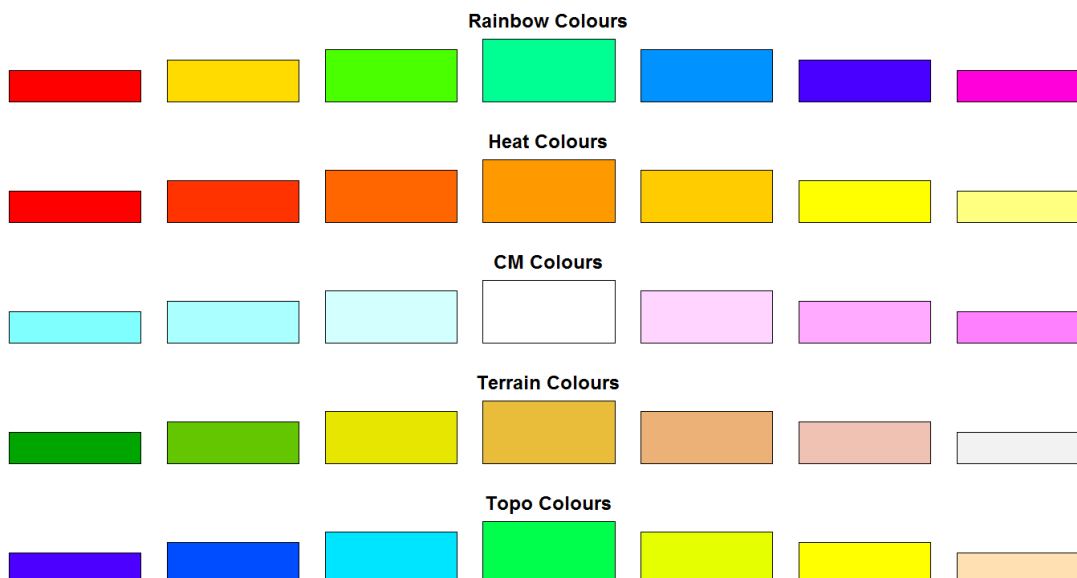
External colour libraries

In addition to the colour functions built into R there are some very good external colour packages which are usually the best place to go for either pre-defined palettes, or for functions to easily generate your own palettes.

Color Brewer

One of the most popular colour palette packages is `RColorBrewer`. The ColorBrewer project (http://colorbrewer2.org/) originated to make colour schemes for maps, but the palettes it produced are now used in lots of applications. It really just consists of a set of pre-defined palettes from which you can select the one you like, and choose how many colours from it to extract. You can view all of the palettes available with:

```
display.brewer.all()
```
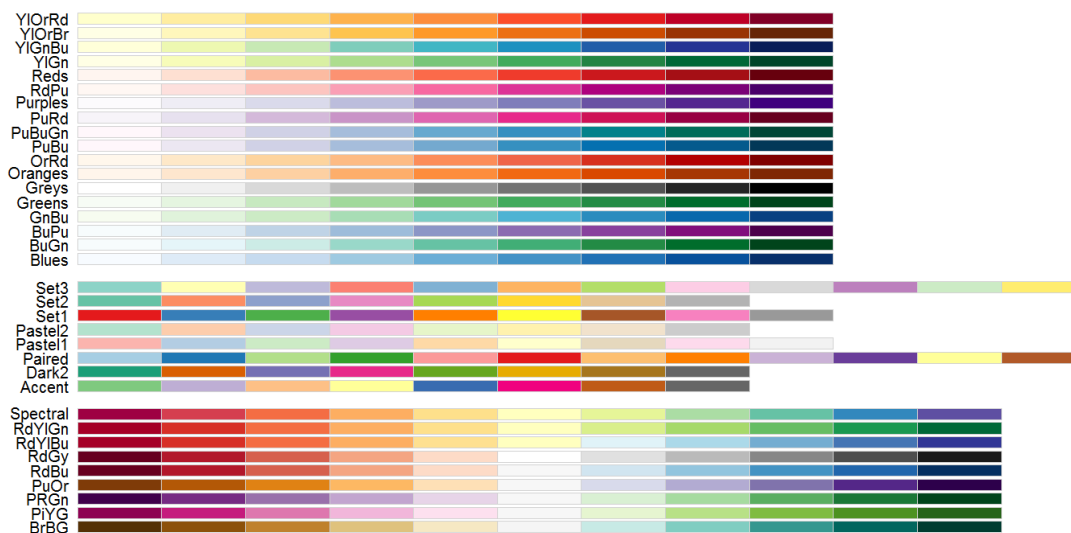
It's particularly good for categorical palettes (the middle block in the image above).

To extract colours from the palettes provided you simply use the `brewer.pal` function to say how many colours you want and which palette you want to take them from.

```
brewer.pal(5,"Set1")
[1] "#E41A1C" "#377EB8" "#4DAF4A" "#984EA3" "#FF7F00"
```

colorRamps

Whilst `RcolorBrewer` is very good for categorical palettes, if you want to generate quantitative colour schemes which smoothly transition between different colours then the functions provided within `colorRamps` may be a better choice. The package provides two functions which are useful for different tasks.

- `colorRamp` takes a vector of key colours and returns a function into which you can pass a vector of values between 0 and 1 and you will get back a matrix of RGB values for colours which correspond to that value.
- `colorRampPalette` takes a vector of key colours and returns a function into which you can pass an integer number and it will return you that number of colours sampled evenly from across the colour palette which was created.

```
colorRamp(c("red","white","blue")) -> palette
palette(seq(from=0,to=1,by=0.1))
       [,1] [,2] [,3]
 [1,]   255    0    0
 [2,]   255   51   51
 [3,]   255  102  102
 [4,]   255  153  153
 [5,]   255  204  204
 [6,]   255  255  255
 [7,]   204  204  255
 [8,]   153  153  255
 [9,]   102  102  255
[10,]    51   51  255
[11,]     0    0  255


colorRampPalette(c("red","white","blue"))(5)
[1] "#FF0000" "#FF7F7F" "#FFFFFF" "#7F7FFF" "#0000FF"
"3838FF" "#0000FF"
```

Colorspace

Another package which provides an easy way to construct and select palettes is the `colorspace` package.

This provides a single function `choose_palette()` which opens up a graphical tool which allows you to select the type of palette you want and adjust some options to create something which you think is suitable. When you've finished with it, it will return a function to generate the palette you selected to your program.

This tool can only be used interactively – you can't put it into a non-interactive R script since it requires your intervention in the selection, but it's a nice way to see what you're getting.
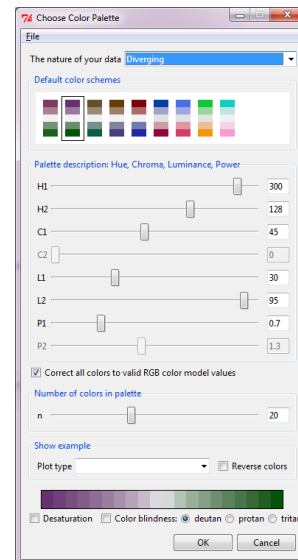
Applying colour to plots

Most plot types will have an argument to pass in the colours you want to use (it's usually `col`). To fill this argument you can generally pass in one of two things:

1. You can pass a vector of colours which will then be applied to the corresponding data

2. In some plot types you can pass in a vector of factors which will relate to the data you provided so that the groups will be created automatically and coloured according to the global palette.

The first option is by far the simplest and can be used in conjunction with the palette generation methods described above.

```
c(1,6,2,4,7) -> data
brewer.pal(length(data),"Set1") -> colours
barplot(data,col=colours)
```
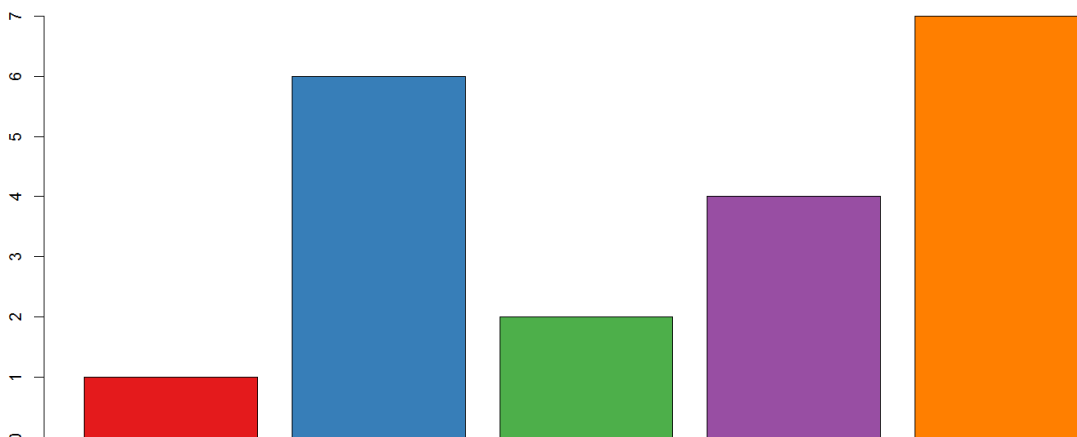
```
group.data
```

```
     Sex Height Weight
1    Male    194  107.0
2  Female    155   76.5
3  Female    159   78.5
4    Male    201  103.5
5    Male    188   94.0
6  Female    117   52.5
7    Male    197   97.5
8  Female    127   60.5
9  Female    130   65.0
10   Male    215  105.5
11 Female    130   72.0
12   Male    205  110.5
13   Male    193   98.5
14   Male    167   82.5
15 Female    171   84.5
16 Female    171   75.5
17   Male    226  118.0
```

```
plot(group.data$Height,group.data$Weight,
    col=group.data$Sex,
    pch=19,
    cex=3)
```
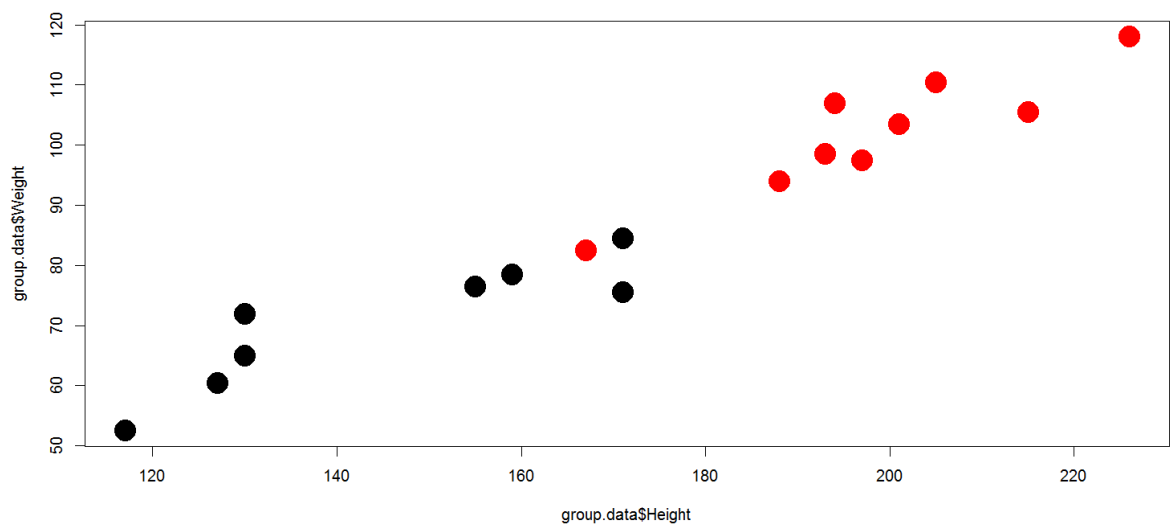

If you use the factor method of assigning colours then you get whichever colours are in the global palette and the ordering of factors is whatever is specified in your factor vector.

If you want to change the colours in the global palette you can pass a new vector of colours into the palette function to set this. This is a global setting and will apply to all plots which use the palette from this point on.

```
palette(brewer.pal(9,"Set1"))
```

If you want to change the ordering of your different factors within a factor vector then you can alter this using the levels function.

```
levels(group.data$Sex)
[1] "Female" "Male"
> levels(group.data) <-c("Female","Male")
> levels(group.data$Sex)
[1] "Female" "Male"
```

Dynamic use of colour

The uses of colour described above would be what you'd use to do categorical colouring, but often we want to use colour for a more quantitative purpose. In some graph types you can specify a colour function and it will dynamically generate an appropriate colour for each data point, but it can be useful to do this more manually in other plot types.

Colouring by point density

One common use of dynamic colour is to colour a scatterplot by the number of points overlaid in a particular area so that you can get a better impression for where the majority of points fall. R has a built in function `densCols` which can take in the data for a scatterplot and will calculate an appropriate colour vector to use to accurately represent the density. This has a built in colour scheme which is just shades of blue, but you can pass in your own colour generating function (such as one generated by `colorRampPalette`) to get whatever colouring you prefer.

```
plot(dens.cols.data,pch=19,main="No density colouring")
plot(dens.cols.data,
     col=densCols(dens.cols.data,
                  colramp = colorRampPalette(
                             c("blue2","green2","red2","yellow")
                                             )
                 )
     ,pch=19,
     main="With custom density colouring"
)
```

**No density colouring**   **With custom density colouring**
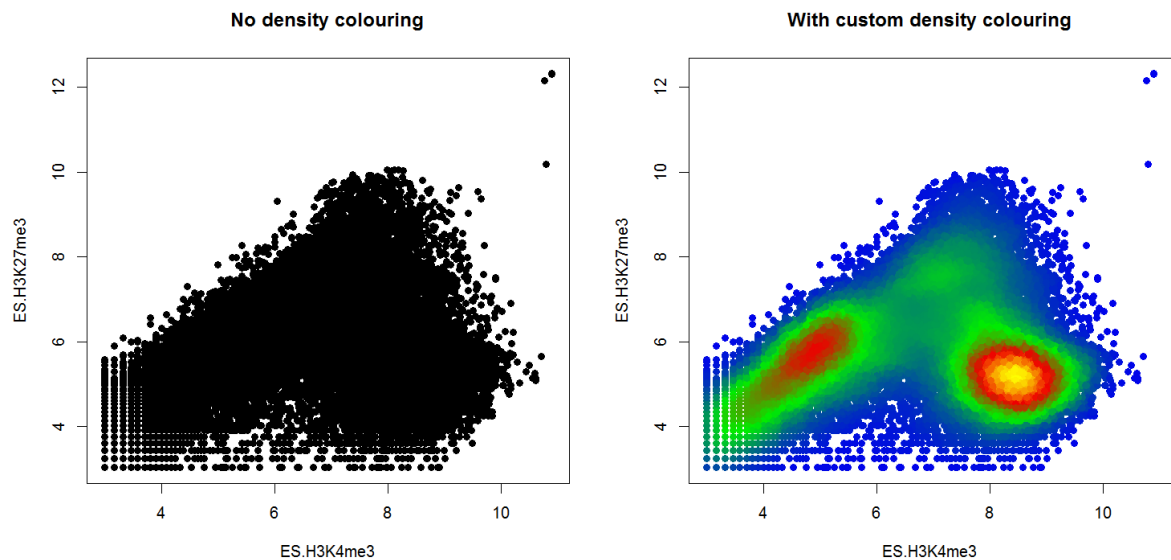
Mapping colours to quantitative values

Whilst colour in general isn't a great way to represent quantitative data it can be really useful to use the colour of points in a plot as an extra quantitative variable. This will then require a way to map colours from a palette to values in a quantitative set. There is no built in function within R to do this, but it's fairly easy to write your own small function to do this. The example below is pretty simplistic in that it scales the colours over the whole range which might be a problem if you had large outliers, but it should be easy to modify if you need something more complex than this.

```r
map.colours <- function(value,range,palette) {

  proportion <- (value-range[1])/(range[2]-range[1])
  index <- round((length(palette)-1)*proportion)+1


  return(palette[index])
}
```

You could then use it like:

```r
diffs = abs(dens.cols.data$ES.H3K4me3 - dens.cols.data$ES.H3K27me3)

map.colours(
  diffs,
  range(diffs),
  colorRampPalette(c("black","red"))(100)
) -> quant.colours

plot(dens.cols.data,
     col=quant.colours,
     pch=19,
     main="Using colour to show differences")
```

Using colour to show differences
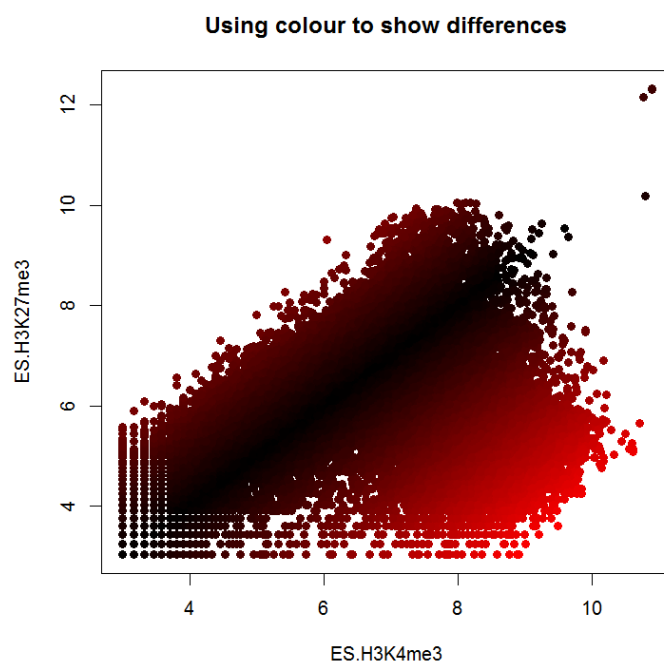
Complex Graph Types
Plot Overlays

After you have drawn your initial plot area and have set the scaling of your x and y axes you can then overlay more layers of data or annotation on top of the original chart. Whilst it is possible to draw another complete plot over the top of an initial plot (we'll do that later), it is much more usual to use a separate set of functions which are designed to add more data to the top of an existing plot area.

points()

The `points` function allows you to plot individual scatterplot points over the top of an existing graph. It can be used to highlight subsets on a normal scatterplot or overlay individual data points on a graph type which summarises data.
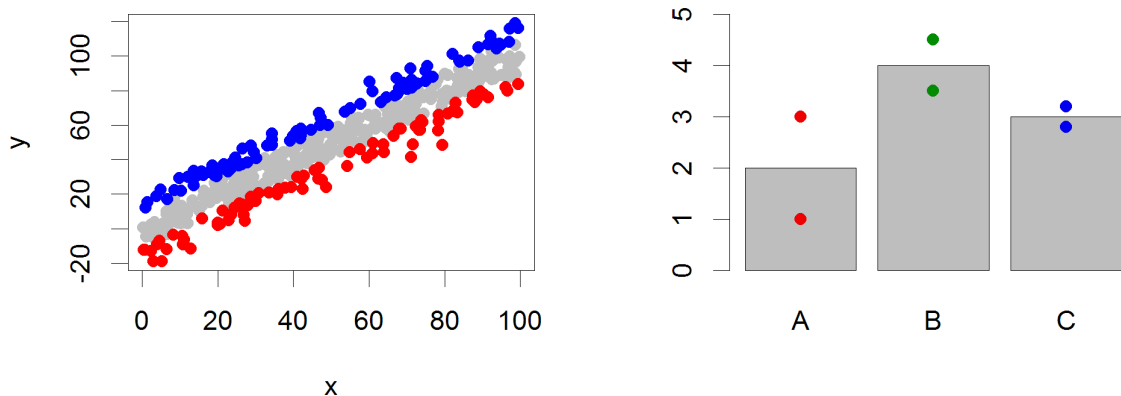
Options:
- Data is a pair of vectors (of the same length) giving the x and y positions of the points.

lines() / arrows() / segments() / abline()

These related functions allow you to place lines over an existing plot. The `lines` function is mostly used to add an extra dataset consisting of multiple x,y segments. The `arrows` function draws individual arrows onto plots, and can be modified to add error bars (which would be arrows with flat heads), and `abline` is used to draw a single line (usually a trend line or regression) to a plot.
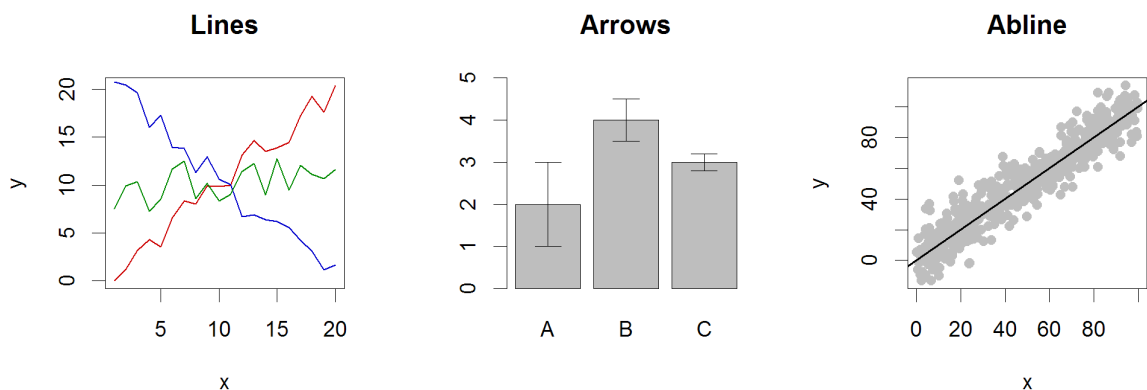
Options:

- For `lines` the data is a pair of vectors (of the same length) giving the x and y positions of the points in the lines to plot.
- For `arrows` you specify separate vectors for `x0`, `y0`, `x1` and `y1` but you can plot multiple datasets at once. If plotting on a `barplot` then the return value from the `barplot` function will give you the centre positions of the bars which you can use in any overlays.
- For `abline` you specify the intercept (`a`) and slope (`b`), or you can pass in an object which defines a line, such as the output of a correlation.
- For `lines` you can set `type` in the same way as used for plot
- For `arrows` you can specify the `length` and `angle` of the arrowhead. You can also use the `code` to specify which type of arrowhead to draw (1=arrow at start, 2=arrow at end, 3=arrow both ends)

polygon()

The `polygon` function allows you to draw shaded areas on plots. It is often used to illustrate confidence intervals in complex plots, or can be used to provide background shading, or other types of highlight.
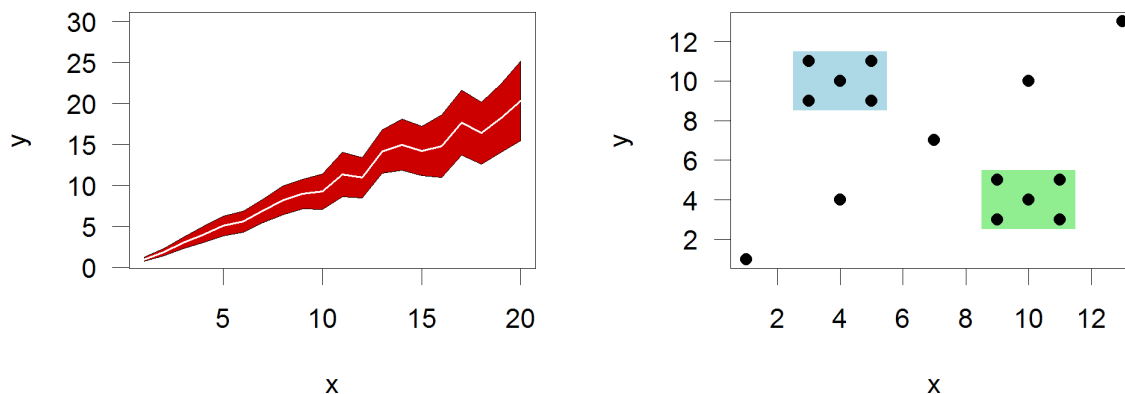
Options:

- The data for `polygon` is a set of two vectors containing the `x` and `y` coordinates of the set of points which define the boundary of the polygon.

text()

The `text` function allows you to place arbitrary text over the plot. It isn't generally used for normal graph components such as axis labels, which are normally options to the initial plot, or figure legends, which have their own function. It can be used to label points of interest or include other non-standard notation onto a plot (p-values for example).
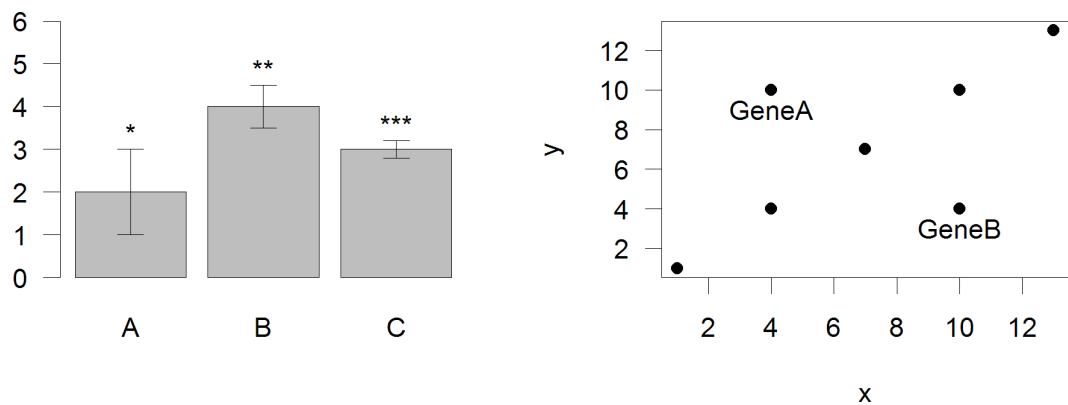
Options:

- The data for `text` are three vectors, one for the `x` coordinates, one for `y` coordinates and one for the `labels` themselves.
- Since you often don't want the text placed directly over the point you plotted you can adjust the label position either by changing x and y directly, or by specifying `adj` which is a 2 value vector indicating the offset to apply to x and y values, or by specifying `pos` where R will calculate an offset (1=below, 2=left, 3=above, 4=right).

axis()

Plot axis are normally provided by the function which originally draws the plot, you don't need to add them explicitly. There are times, however when having the ability to draw individual axes can be useful. This would include cases where you wanted to have multiple x or y-axes, or if you wanted the axis to be on a different scale to the data.
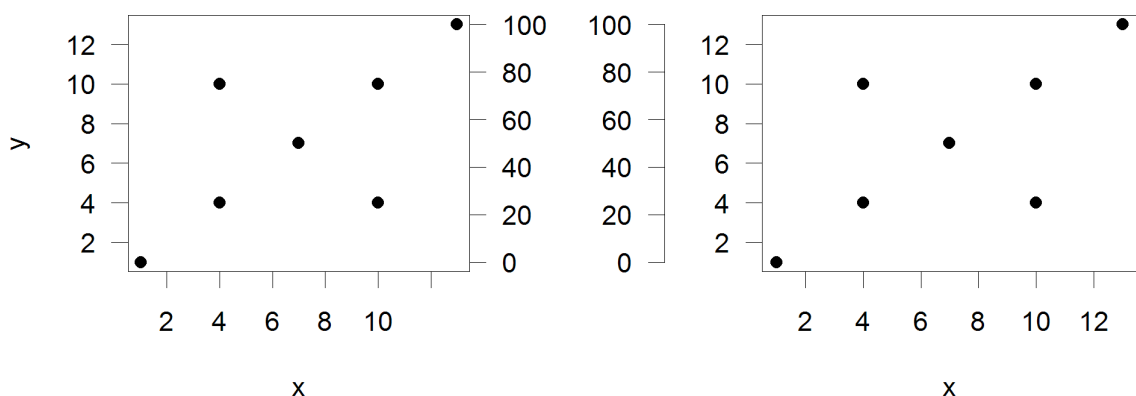
Options:

- The position relative to the plot is set with `side` (1=below, 2=left, 3=above, 4=right)
- The positions of the tick marks are set with `at`. The scaling is set by the last plot drawn so if you want a secondary axis you must draw a new plot area with `new=TRUE` set in your `par` options to change the plotting scale.
- To move the axis back from its default position next to the plot area uses the `lines` option to say how much space to leave before drawing the axis.
- If you want to add text to the new axis then you can use the `mtext` function to add this. This is a separate function, and not a parameter to the axis command.

legend()

The `legend` function is used to add a standard data legend to a plot. It can either be pinned to one of the corners of the plot, or placed at any arbitrary location in the graph.
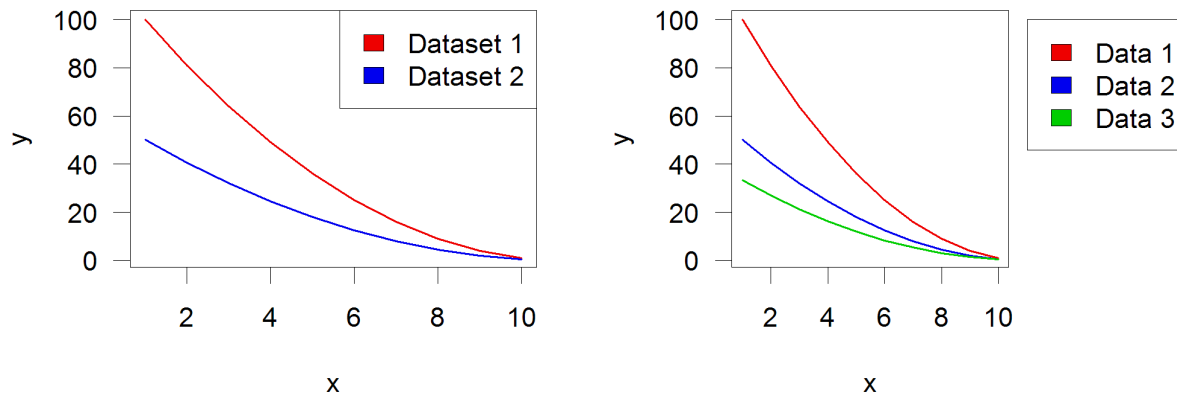
Options:

- The position of the legend can either be set with a compound text string such as `topright` or `bottomleft` or with specific `x` and `y` coordinates.
- If you want the legend to be positioned outside the main plot area an still be visible then you need to set `xpd=NA`
- Labels are set with the `legend` argument.
- If you want shaded boxes next to the labels then pass a vector of colours to the `fill` argument.

title()

Most commonly you will add titles, subtitles and axis labels when you first create your plot, but you can use the `title` function to add these after plotting. Using the `title` function gives you a little more control over the placement of titles than may be available in the main options of the individual plot types. It would also allow for adding titles to whole sets of multi-panel plots.
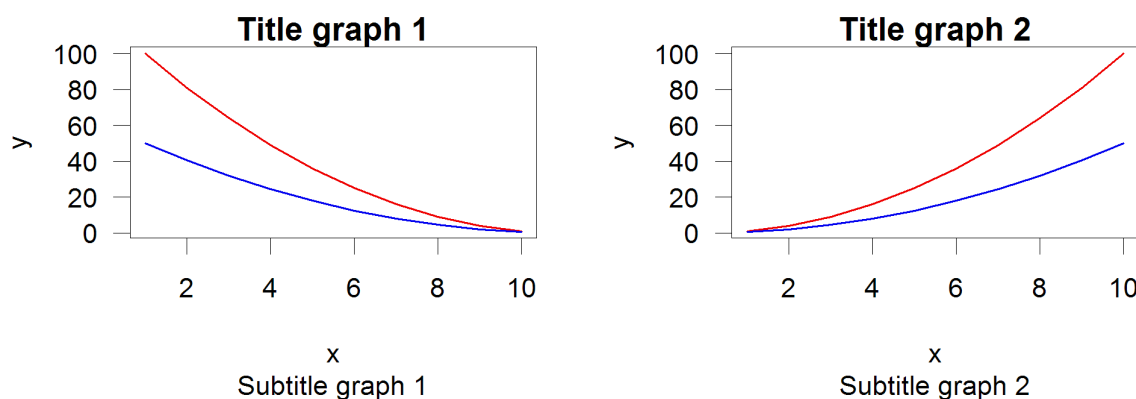
Options:

- Two title strings can be passed into the function, one for the `main` title (above the plot) and another for the `sub` title (below the plot).
- If a multi-panel plots is being constructed then you can pass `outer=TRUE` to have the title applied to the outer margins of the plot which are common to all of the sub-plots.

Multi axis plots

When adding an overlay to an existing plot you are only adding additional graphical elements, not generating a new plot area. The coordinate system for the plot is set by the first plotting command and is then retained through the addition of extra layers.

In some cases though you want to be able to re-configure the scaling of the plot area, so that you can overlay plots on different scales. Drawing this kind of multi-scale plot brings up a couple of problems – firstly how to change the plotting scale within a single plot area, and secondly how to add additional y-axes to be able to represent the different scales used for different lines within the graph.

Changing the plotting coordinate system

The mechanism you can use to change the coordinate system for a plot is to simply repeat the drawing of the base plot using a new dataset to reset the ranges of the x and y axes. Normally when you do this you would clear the previous plot and start a new one, however if you set `par(new=TRUE)` then the previous plot area will not be cleared and the new plot will be overlaid. If you only apply this parameter then you will also overwrite the axes and borders of the original plot, so for the secondary plot it is usual to set `xaxt="n"` and `yaxt="n"` so as to not re-draw the x and y axes. It is then possible to add a secondary axis later by simply calling the `axis` function with `side=4` to place it on the right using the new scale.

```
plot(0:10,(0:10)*100,type="l",lwd=2,col="red",ylab="",xlab="Values",main="Multi Scale",yaxt="n")
axis(side=2,col="red",lwd=2)
par(new=TRUE)
plot(0:10,rev(0:10),type="l",lwd=2,col="blue",xaxt="n",yaxt="n",xlab="",ylab="")
axis(side=4,col="blue",lwd="2")
```
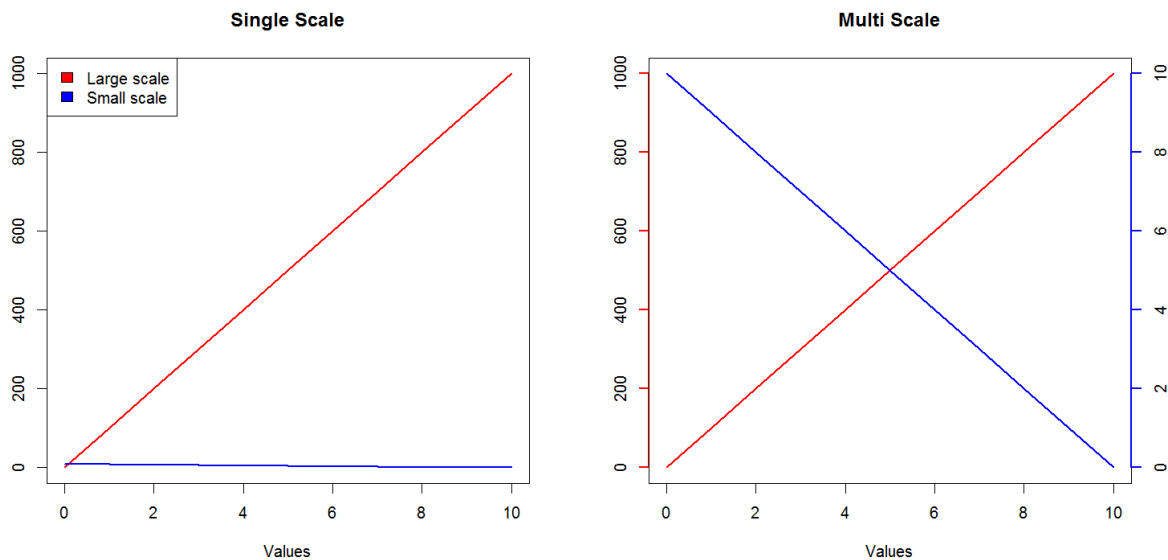
Adding secondary axes

In the example above we simply used the `axis` function to add a new axis to the right side of the plot for the second scale. In some circumstances though we want the axis to appear as a secondary axis on the left. In this case you'll need to firstly adjust your `par(mar)` setting so that you leave a larger margin on the left of the plot to be able to accommodate a secondary axis. You can then do the plot as shown above, but now you can keep the axis on the left (`side=2`) and then use the `lines` option to say how far back to position it.
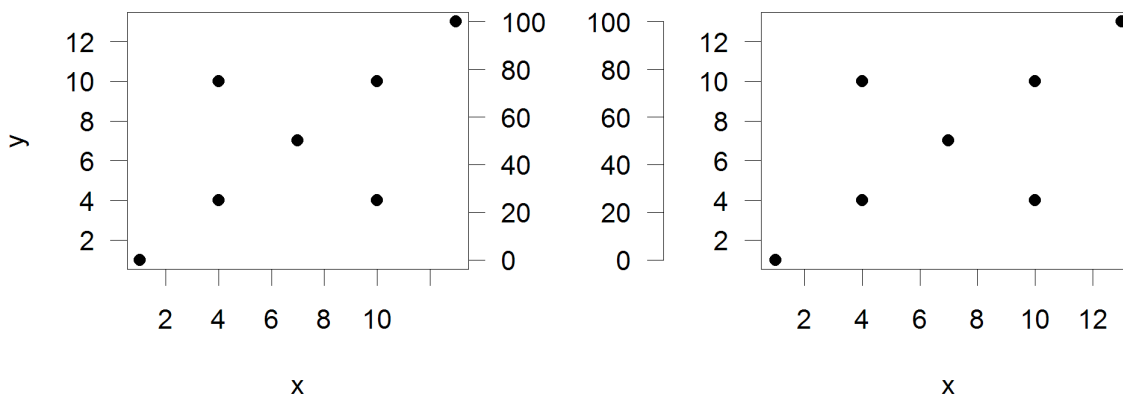
Drawing outside the plot area

By default most items plotted in R must be contained within the defined plot area. This would mean that you could plot a subsequent dataset which had values outside the original range for the plot, and it wouldn't escape the boundaries of the plot to draw over other parts of your figure.

There are occasions though when you want to be able to draw outside the plot area. The most common of these is to place a data legend alongside the plot instead of within it, but there may be other cases where this is useful too.

The variable you can adjust to control this is the `par xpd` variable. This can take one of 3 values. The default is `FALSE` and constrains drawing to the plot region. If you set it to `TRUE` you can draw over any part of the current figure, but can't overwrite onto other plots which might be on the same panel. If you set it to `NA` then you can draw anywhere.

You can then specify an x and y position outside of the plot area to position any new elements. The scaling for x and y is still taken from the plot area.

```
par(mar=c(4,4,4,10))
barplot(c(1,2,1.5,2.5,3),col=c("red","orange","yellow","green","blue"))
par(xpd=TRUE)
legend(paste("Sample",c("A","B","C","D","E")),fill=c("red","orange","yellow
","green","blue"),x=6.5,y=2.5)
text("Legend",x=7.6,y=2.7)
```
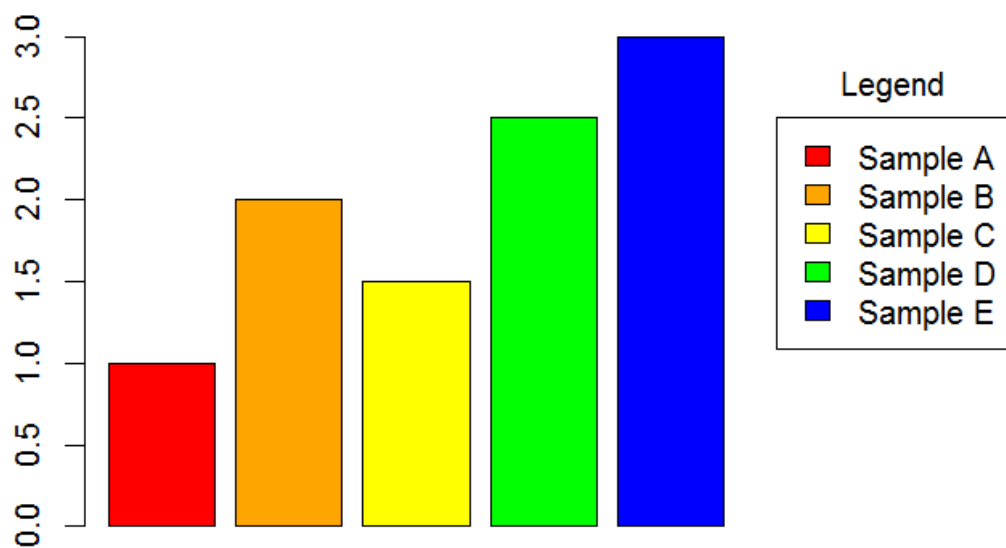
Common extension packages

In addition to the plotting functions there are a couple of stand-alone packages which act in a very similar manner to the core plotting functions, but which add some very useful graph types which are otherwise not available, or not very user friendly. In particular the packages `beanplot` and `heatmap.2` are worth looking at.
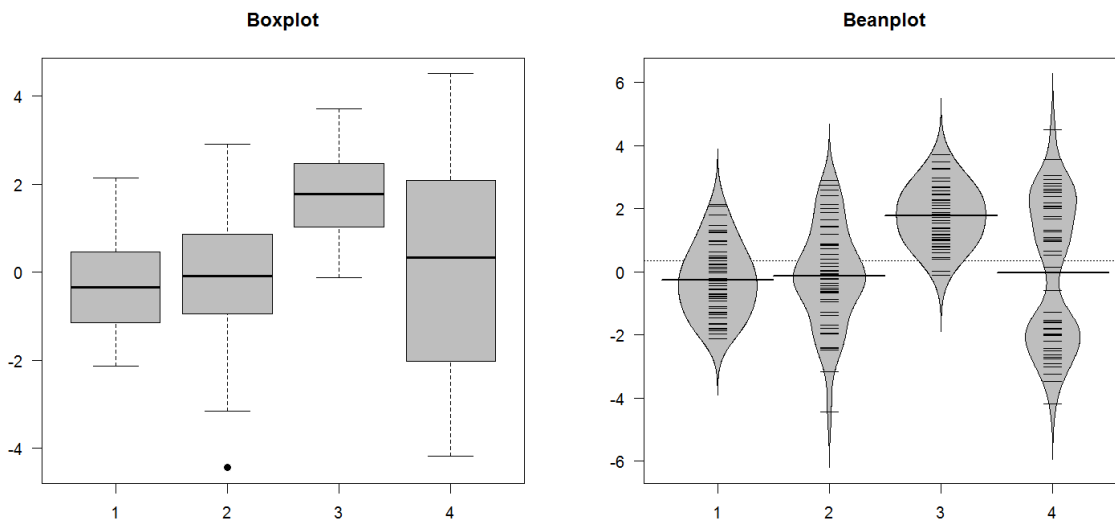
Beanplot

A beanplot is a graph type used to summarise the distribution of a set of values. The nature of the plot means that, much like a boxplot, it is easy to summarise several data series in a single plot, however, unlike a boxplot, the beanplot allows to you see the full range of the distribution of values, and makes no assumption about the data being normally distributed. It is therefore generally a better option than a boxplot in most cases.

Beanplots are provided by the `beanplot` package, and the standard data arguments mirror those for a boxplot. You input data can be a single vector, a list, a data frame or a formula.

```
boxplot(bean.data, las=1,names=1:4,pch=19,col="grey")
beanplot(bean.data, las=1,names=1:4,col="grey")
```


In the example above you can see that the fourth data series is actually bimodal in nature. In the boxplot this simply looks like increased variance, whereas the greater detail in the beanplot allows you to see the details of the distribution.

There are a number of additional options which can be used with `beanplot` to alter the display, or to configure how data is shown.

Options:
- The `bw` parameter determines the function used to draw the smoothed outline around the bean data. Normally you can leave this alone, but if you have a dataset with large numbers of replicates in it (especially zeros) then you may find that the plot refuses to draw because it can't estimate a suitable curve. In these cases setting `bw=nrd0` (zero at the end) should allow the plot to proceed.

- The `what` parameter allows you to remove parts of the plot. The default `beanplot` presentation shows the outline of the data density, but also a horizontal line for the average value, a set of small 'bean' lines for each data point and a dashed line for the global average. The `what` parameter lets you set a vector of 4 values of 1 or 0 to say whether to draw the total average, beans, per-bean average and the density line.
- The `names` parameter allows you to explicitly name the groups in the plot if these aren't picked up automatically from the data.
- By using the `side` parameter you can say whether the data is plotted to both sides (the default) for just left or right. By using a subsequent over plot on the same scale you can then plot the other side to be able to make a direct comparison of two datasets.

Gplots

The `gplots` library was developed to try to overcome some of the shortcomings of some of the core plot types in R. It provides alternative versions of several of the core graph types and adds some new ones. There is one particular plot in `gplots` though which has largely supplanted the original core version and this is the `heatmap.2` plot.

Heatmap.2

The core R plot has a `heatmap` function which can use clustering and colouring to allow the display of huge datasets in a compact and efficient way. The core `heatmap` however has quite limited customisation and a more advanced `heatmap.2` was developed which has largely supplanted the core version for this graph type. The core version of the heatmap has a lot of features, including a histogram to show the colour scale, clustering on two axes and various trend lines superimposed over the main plot.
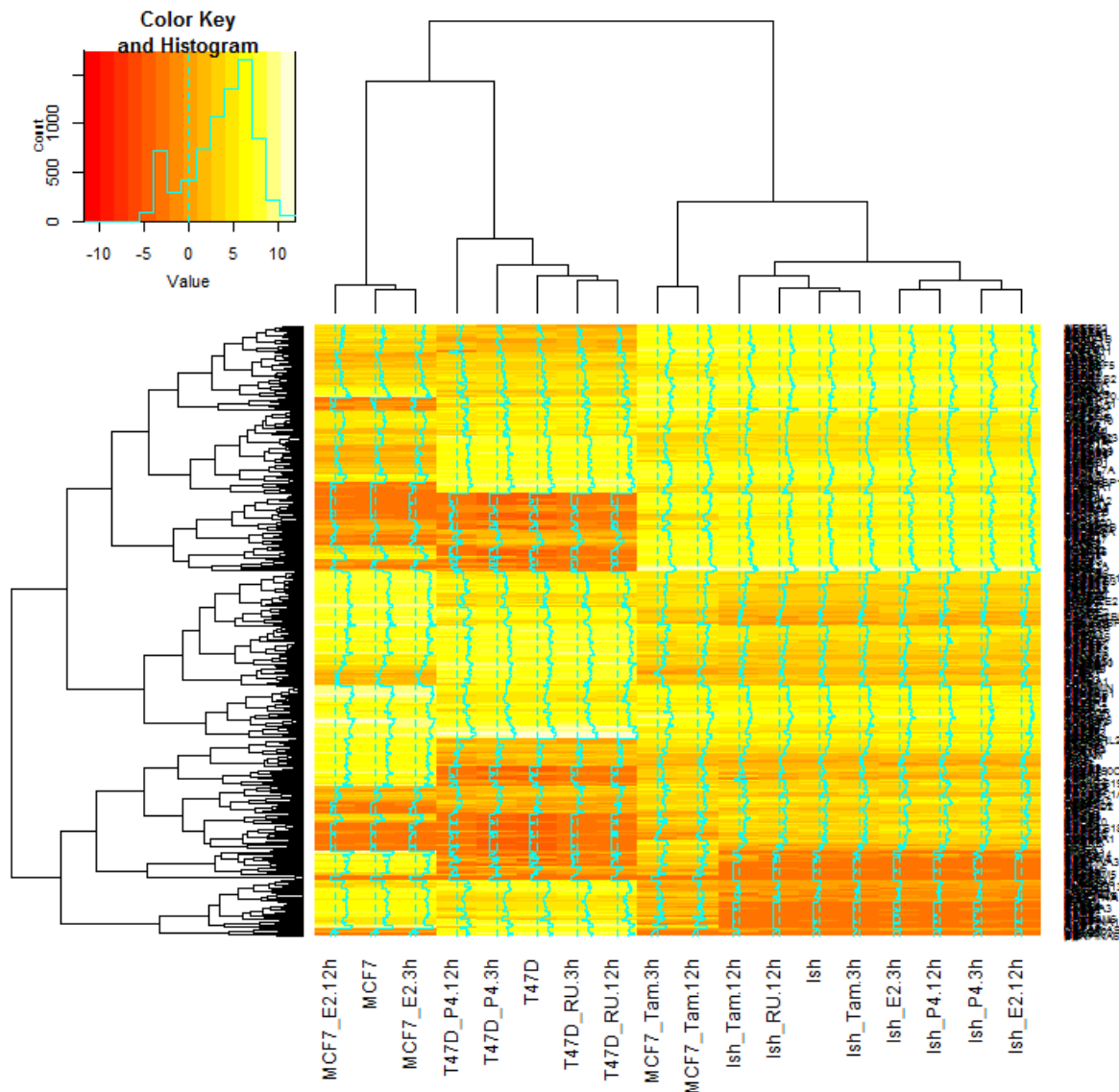
The advantage of using `heatmap.2` is that you can change a large number of the aspects of this plot.

- You can increase the amount of space for the labels on the x and y axes using the `margins` parameter (note, not the same as the `par mar` parameter. You pass in a 2 element vector with the margins for the bottom and right hand side respectively.
- You can remove the per column traces using `trace=none`
- You can remove the histogram colour key using `key=FALSE`. You can make it smaller using `keysize` and setting a size value relative to 1.
- You can change or remove the column and row labels using `labCol` and `labRow`. If you set them to and empty string then the labels will be removed.
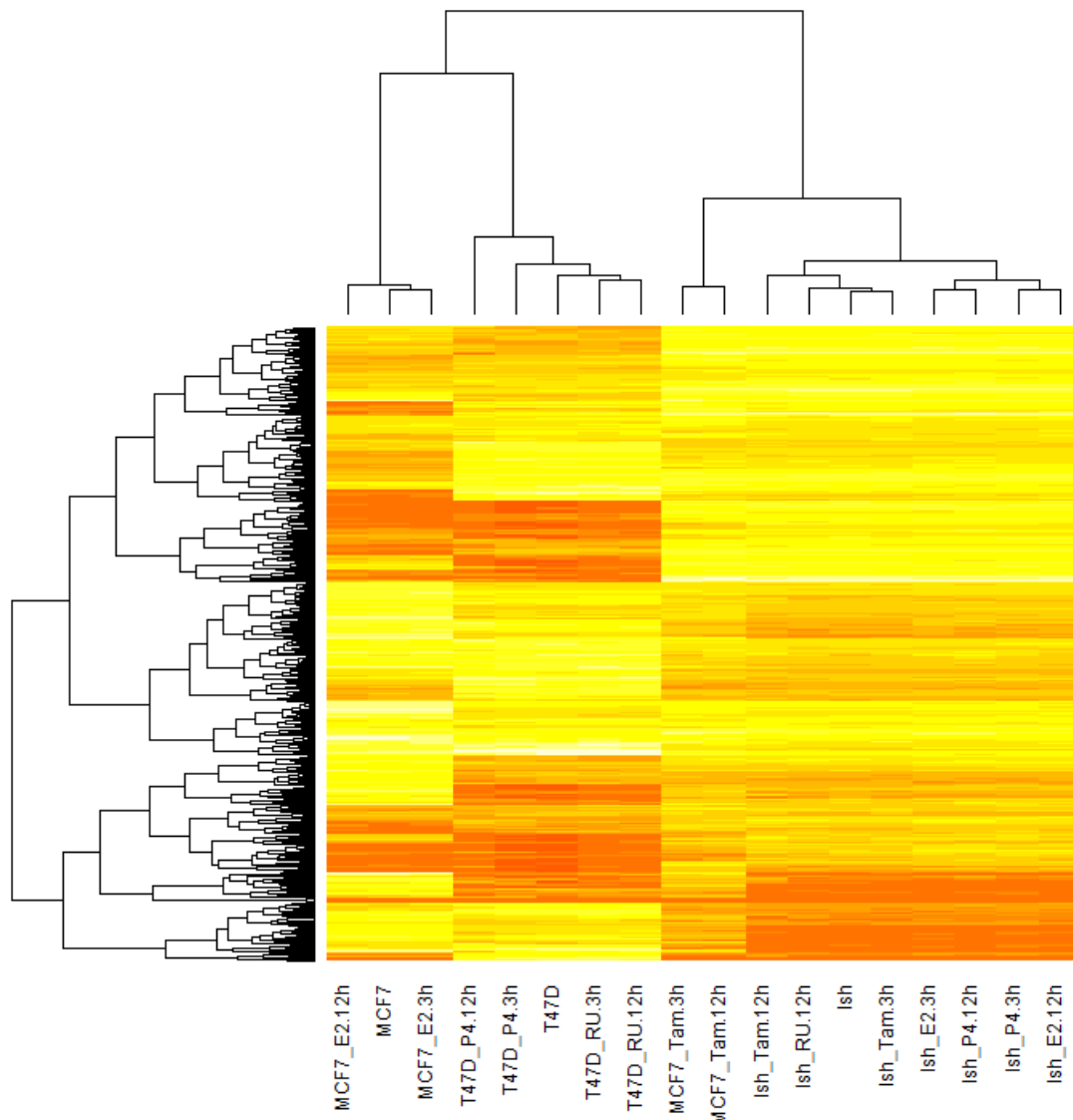
Heatmap colours

The default `heatmap` view will plot the data values supplied to it without any transformation or normalisation. The colours used therefore reflect the range of values in your data. If you have negative values then the range will be made symmetrical around zero. The default colour scheme is not hugely informative in this situation and you will often want to substitute your own.

To change the colour scheme you can use the `col` parameter to provide a function which can generate the colours you want. The easiest way to generate this function is with the `colorRampPalette` function discussed earlier.

The range and scaling of the colours is determined by the `breaks` parameter. This can either be a single value indicating how many divisions the range of data should be split into, or a vector of endpoint values used to define the different bins you want to use – this would be the way to compress the scale used.

If you don't specify exactly which breaks to use then you still might want to change whether the range of breaks is symmetrical about zero or not. The plot defaults to symmetrical ranges if the data contains

negative values, but is otherwise asymmetric. You can adjust this behaviour using the `symbreaks` parameter.

```
col=colorRampPalette(c("white","blue")),
breaks=50
```
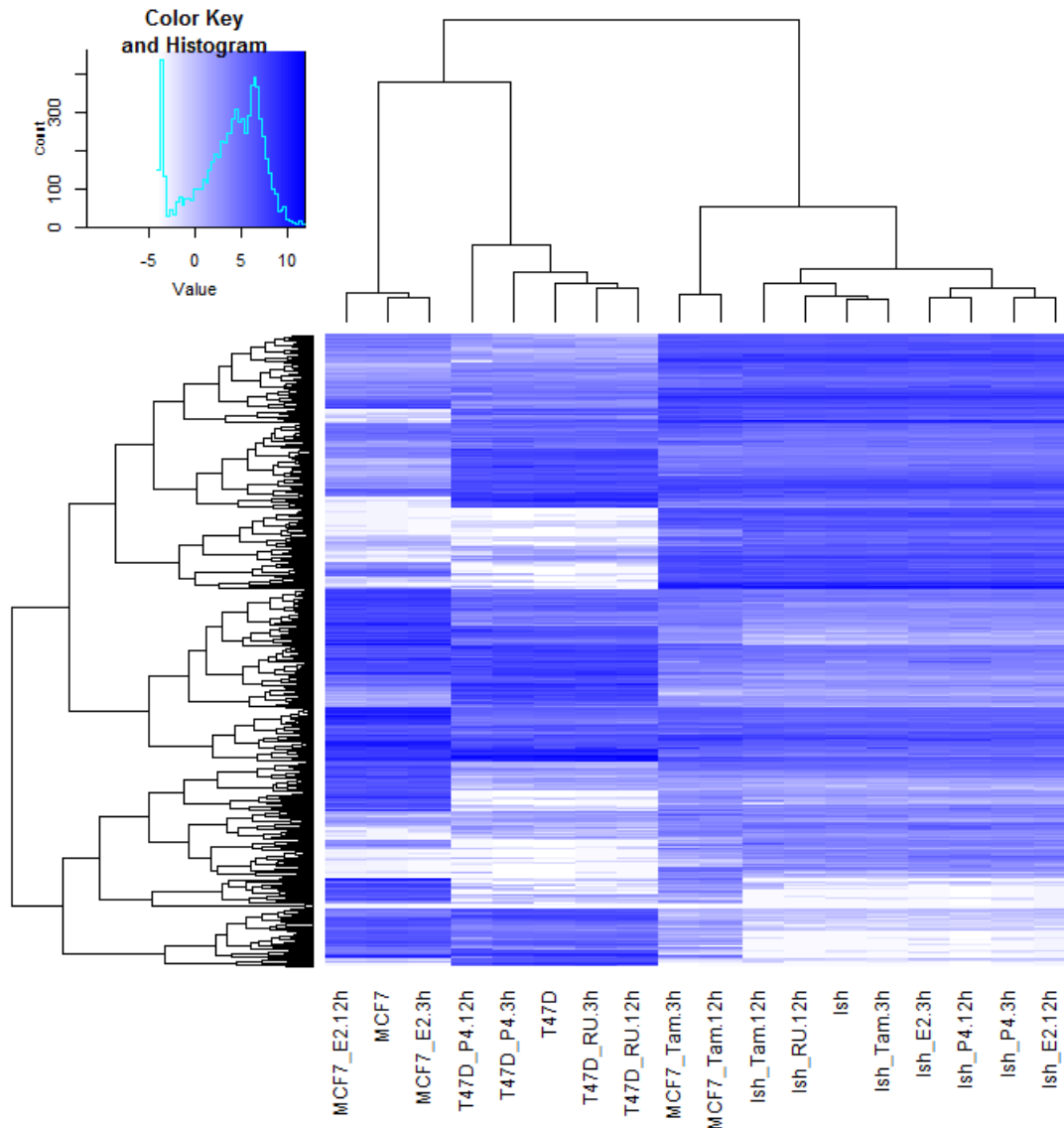

Clustering options

In general terms there are two common types of clustering you are likely to use:

1. Euclidean clustering where the similarity between two datasets is determined by the difference in their absolute data values.
2. Correlation clustering where the similarity between two datasets is determined by the pattern and direction of their changes.

In general you use Euclidean clustering when you want to focus on the importance of the absolute quantitated values. You would use correction clustering when you care about datasets having the same trends in their differences without actually caring about the magnitude of those changes.

`heatmap.2` defaults to Euclidean clustering, which won't be appropriate for many applications, so it's useful to know how to change this to correlation clustering. The parameter which controls this is `distfun` which controls the function used to calculate the distance between two data sets. This needs to be a function which returns a distance value. The function you would need to use the standard `cor` function (Pearson's correlation) in a heatmap would be:

```
function(x) as.dist(1-cor(t(x)))
```

When doing a correlation based clustering you should also generally normalise your row data to z-scores to remove the effect of different absolute values, and different variances on different rows. To do this you set `scale="row"` in the `heatmap.2` options. You should see that the scale on the histogram changes to show z-scores. When plotting z-score based data you should generally use a diverging, symmetrical colour scheme.
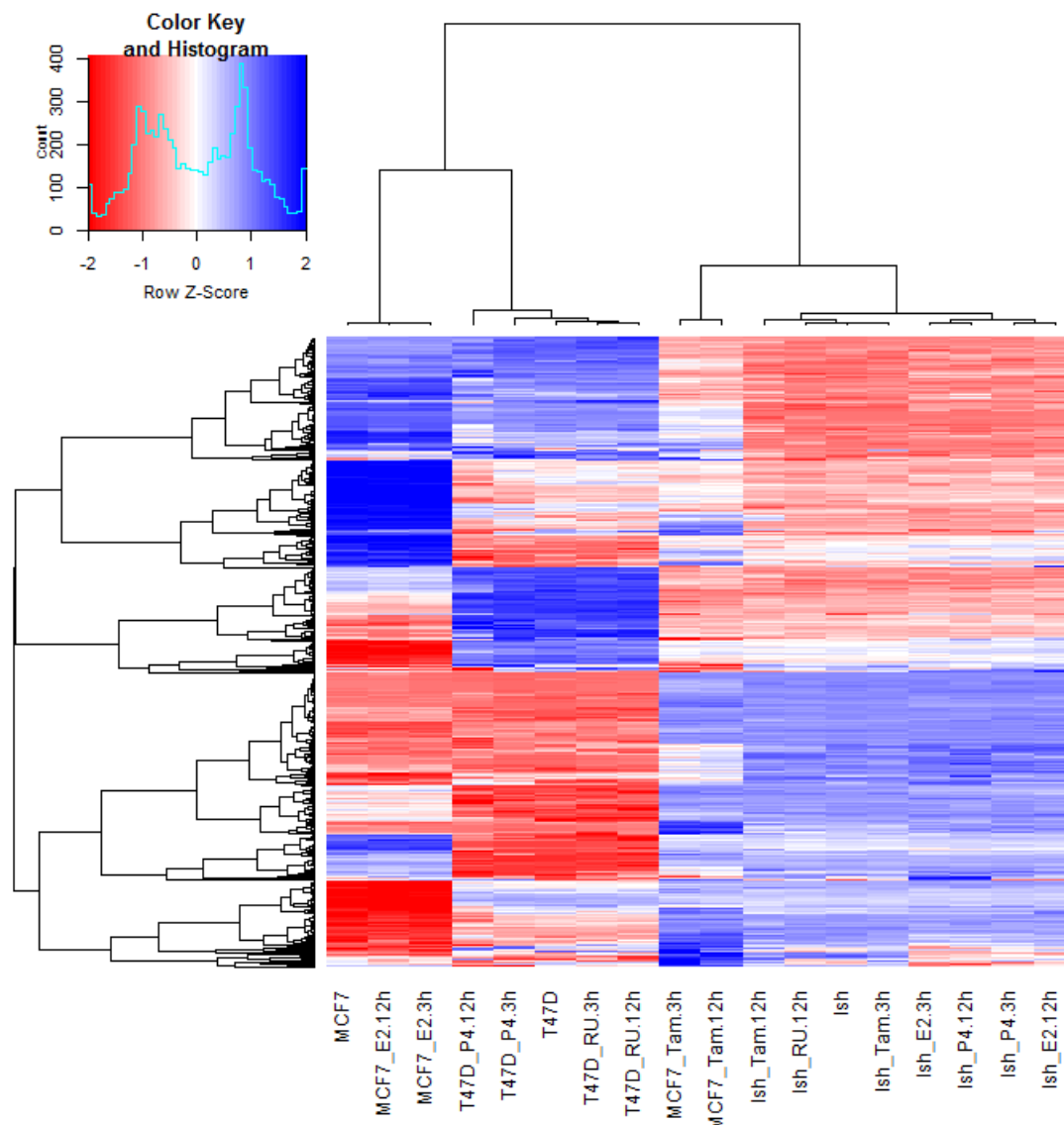
Exporting graphs

When working within R you can either plot your figures to an interactive plotting device which will allow you to see the results of your plot straight away, or you can send the figure straight into a file in one of the supported graphics formats.

Graphics Devices

To write graphics data to a file you need to set up a graphics device of the correct type. This will determine the format of the file generated as well as other parameters such as its size, scaling and resolution.

Once your device is set up then you can send plots to it using the same functions as for interactive plotting and the data will be written straight to the file.

When you've finished your plot you can call the `dev.off()` function which will close the device and the file it is attached to.

Graphics functions

Below is a list of the different functions available from within R.

File format	Function	Bitmap/Vector	Editable	Lossless	Recommended for
Bitmap	`bmp`	bitmap	no	yes	
Jpeg	`jpeg`	bitmap	no	no	
Png	`png`	bitmap	no	yes	Presentations / Web
Tiff	`tiff`	bitmap	no	yes	
SVG	`svg`	vector	yes	yes	Compositing figures
PDF	`pdf`	vector	yes	yes	Printing Immediately

When running the function to set up the device there are a few parameters you can set. These generally determine the size and resolution of the plot area.

- The `units` parameter says what the numbers for the page size will mean. You can pick from `px` (pixels), `in` (inches) or `mm` (millimetres).
- The `height` and `width` parameters set the plot size using the value from `units`.
- The `pointsize` parameter says what the default font size will be in plots. Setting this value higher will make all text larger.
- The `res` parameter sets the resolution of the image in pixels per inch. It will have an effect on the default interpretation other programs will make about the size at which the image will be displayed, and will also affect the `pointsize` parameter since this is related to the resolution.