# A guide to data visualization using BoutrosLab.plotting.general

Christine P'ng & Jeff Green

October 17, 2023

# Contents

| Rank | Aspect judged |
|---|---|
| 1 | Position along a common scale |
| 2 | Position on identical but nonaligned scales |
| 3 | Length |
| 4 | Angle, Slope |
| 5 | Area |
| 6 | Volume, Density, Colour saturation |
| 7 | Colour hue |

Table 1: Tasks ordered from most to least accurate. Adapted from Cleveland & McGill.

## 1.0 Introduction

Why does data visualization matter? After all, data visualization takes work. Datasets are often complex, having millions of data points and many dimensions. Decisions have to be made regarding the best way of highlighting the main message. Aesthetic considerations require knowledge of colour theory, typography, composition, and more. Why do people go through all the trouble?

The reason is that it is often faster and easier for a person to process data visually. Noticing relationships between data points in a spreadsheet requires much more careful attention compared to viewing the same data points in a chart. Graphs can be used to first find trends in the data, and later to illustrate and highlight messages in data for communication to others[? ?].

### 1.1 Stages of figure creation

When it comes to making figures, there are four main steps:

1. Understand the data and know what the plot is supposed to convey
2. Determine which chart-type is best suited to the data
3. Design the figure to communicate clearly and be aesthetically pleasing
4. Evaluate if other people can understand it

Understanding a dataset is especially important when a dataset is large and not all of it is relevant for plotting. For example, if a dataset has five dimensions, but only two are relevant to convey the main message, it could be appropriate to only display the relevant dimensions. In addition, knowing what the data is trying to convey is helpful for choosing the correct chart-type and properly emphasizing the data.

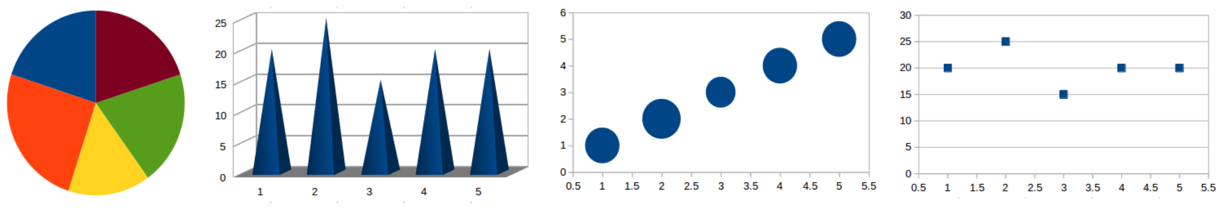


Figure 1: Different charts displaying identical data

Different chart-types are suited for different data-types, and emphasize different kinds of relationships. Using the correct chart-type will improve the readability of the data. It is important to note that some methods of visually encoding data are easier to interpret than others (Table 1). For example, pie charts are difficult to interpret because people struggle to translate angles and areas into exact numerical values. Instead, a barchart which compares lengths on a common scale would be easier to read. Display methods which are more accurately interpreted should be selected over methods which are relatively difficult to accurately interpret[? ?].

Figures should be designed to enhance a viewer's ability to compare data and draw appropriate conclusions[?] . This can be done through careful arrangement of a figure, such as placing related elements near each other, highlighting important items, selecting appropriate colours and fonts, and more. Poorly-designed figures can distort the truth and lead viewers to draw inaccurate conclusions[?] .

To check if a figure is effective, it is helpful to request an interpretation of the figure from someone who does not know what the intended message is. This "test" (which may need to be conducted multiple times) can reveal how clear a figure is.

### 1.2 What is BoutrosLab.plotting.general?

BoutrosLab.plotting.general is a software package for generating publication-quality, customizable plots. It produces a variety of chart-types, ranging from common charts for simple datasets to novel charts for displaying highly-dimensional datasets.

This package is built on top of the `lattice` package[?] , which is an implementation of Trellis graphics[?] for R. Therefore, when plots are created in this package, parameters are passed to the appropriate `lattice` function and a Trellis object is created. Trellis graphics are graphs that can display a variable or the relationship between variables, conditioned on one or more other variables. Therefore, they are very useful for displaying data sets with many variables, such as is often the case in biological data sets.

## 2.0 Designing figures

Before creating figures, it is important to know something about design. Terrible design will miscommunicate data, whereas excellent design will guide viewers to understand the data[?] .

### 2.1 Typography

Typography is the art of arranging type to make written words understandable. This encompasses decisions related to which typeface to use, at which size, with what line spacing, and more.

When it comes to choosing fonts for a figure, it is advisable to use no more than one or two fonts. The distinction between a typeface and a font is often confused: a typeface is Arial, which is made up of a number of fonts, such as Arial bold, Arial roman, Arial italic, and other such fonts[?] . Thus, when selecting which fonts to use in a figure, one option is to use Arial roman for most text, and Arial bold for headers.

Another options for selecting different fonts is to choose very different fonts. This can be achieved by combining a serif and sans serif font with different font size, colour, boldness, *etc*[?] .

Typefaces can be divided into two classes: serif and sans serif. A serif is a small mark found at the tips of letter forms. Serif fonts have serifs (such as Times New Roman), while sans serif fonts do not have serifs (such as Arial). There are mixed opinions as to which kind is advisable for which occasion.

For figures, we advise using common fonts, such as Arial. Unusual fonts may give unintended impressions, and has the danger of distracting viewers from the message, especially when the vast majority of figures use very standard fonts. This is enforced by journals, which often have figure guidelines restricting font choice.

If there is sufficient cause to use an unusual font, ensure that the chosen font is legible and appropriate for the use-case.

### 2.2 Principles of design

The following principles of design are concepts to keep in mind when arranging elements of a design:[?]

- Contrast: used to attract the eye and organize the composition
- Repetition: creates unity and consistency

- Alignment: adds visual connection between elements and helps to unify and organize
- Proximity: group related items helps organization and movement

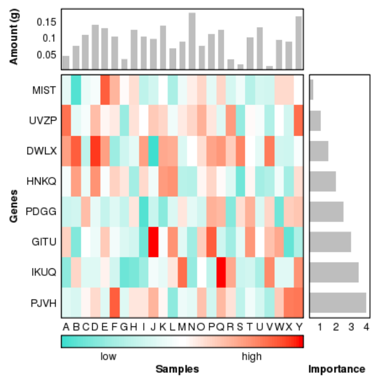


Figure 2: Multiplot example

Take this figure as an example. The bright red cells in the heatmap attract the eye first: they contrast the white and turquoise. This highlights the genes and samples with a high expression change. This bright red is repeated in the colour key underneath the heatmap, as well as in varying degrees of saturation throughout the heatmap to unify the figure. The surrounding barplots and colour key are properly aligned to the borders of the heatmap, and each barplot is in close proximity to the row or column it is associated with.

When designing a layout, it is important to understand which elements of the composition are most important: what *should* a viewer notice first? Good layout controls how the viewer's eye moves around the page. Therefore, the most important elements should stand out the most. This can be done by varying the colour, size, shape, and more. In addition, if a design is divided into thirds in both directions, the intersections of the dividing lines is where focal points should be located. This is called the rule of thirds[?] .

## 2.3 Understanding colour

Colour is a highly relative medium. Someone might say that red is an 'angry' colour. Another person seeing the same colour may believe instead that red is associated with being 'lucky'. There is no consensus on what a colour means.

### 2.3.1 Colour theory

Colour can be described in three ways:

- Hue is what the colour is called, for example 'red'
- Saturation is how vibrant the colour is: 'dull red' versus 'bright red'
- Value is how light or dark the colour is: 'light red' versus 'dark red'

The colour wheel is a way of arranging colours such that primary colours are equidistant, and the result of their mixing fills in the in-between segments (which are called the secondary and tertiary colours). This colour wheel shown is the one commonly used by artists, where the primary colours are red, yellow, and blue.

Red, orange, and yellow are considered the 'warm' colours, while green, blue, and purple are the 'cool' colours. The warm colours are thought to attract more attention (with red being the most attention-grabbing colour), while the cool colours tend to visually recede. This means that if equal amounts of warm and cool colours are in a design, the warm colours will dominate. This applies to data visualization because of two equal areas of a plot are coloured red and green, the red areas will appear larger[?] - although this only appears to be the case for high-saturation colours[?]

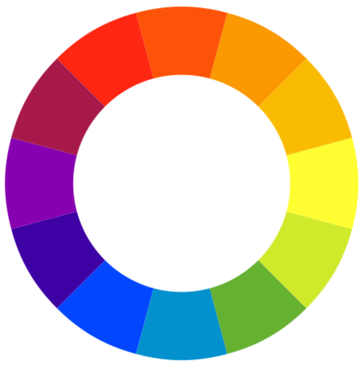


Figure 3: Colour wheel

Complementary colours are colours which are placed at opposite ends of the colour wheel (for example: red and green). These colour combinations achieve the highest hue-based contrast. A triad of colours is made up of three colours which are equidistant on the colour wheel. A double complement is composed of two sets of complementary colours. Analogous colours neighbour each other on the colour wheel.

### 2.3.2 Colour models

There are different ways of generating colour, and consequently colours displayed on different mediums often do not appear the same. For example, a colour scheme might be easily distinguishable on a computer screen, but not when printed.

Computer monitors generate colours using the RGB colour model, which is based on light (primary colours are red, green, and blue). In contrast, printers generate colours using the CMYK colour model, which is made with ink (primary colours are cyan, magenta, and yellow)[?] . The important thing to note is that the colours made in RGB and the colours made in CMYK may not appear the same. RGB creates a more vibrant spectrum of colours than CMYK: this is why printed material often appears duller than on a computer monitor. Therefore, if you expect your figure to be reproduced in print, use the CMYK colour model to have a more accurate idea of what the final image will look like.

CMYK and RGB are device-dependent colour models. This means that a colour specified using these colour models may appear different when created using different devices[?] . Some software can convert CMYK colours to RGB and vice versa, however different software use different algorithms to do this conversion[?] . The result is that a single colour, converted to a different colour model using two different software will produce two different colours. The same problem can be found when using different printers and even across operating systems with different monitor calibration[?] .

### 2.3.3 Colour blindness

Colour blindness is a decreased ability to distinguish colours, which affects a significant percentage of the population. The most common forms affect a person's ability to distinguish red and green.

When choosing colours, it is good practice to accommodate colour blind individuals. There are a number of ways to do this: one method is by not relying on colour hue alone, but also colour value and other elements. For example, a scatter plot might use different plotting characters in addition to different coloured dots.

Another option is to avoid colour schemes which mix colours with components of red and green. Alternatives include red-turquoise and green-magenta colour combinations[?] .

Using software to simulate colour blindness can be a good test of whether or not a given figure will be challenging to interpret for colour blind individuals[?] .

Choosing colour schemes which are greyscale-compatible is desirable both for people with colourblindness and for when figures are reproduced in black and white. There are two main ways to ensure a figure is decipherable in greyscale and it is advisable to use both. The first method is by choosing colours with large differences in value (lightness and darkness). This with create a variety of greys, whereas if all colours have the same value, they will appear to be the same grey. The second method is by varying the plotting characters or line types. For example, a scatterplot might use circles and triangles to divide different groups in addition to differently coloured points.

### 2.3.4 Simultaneous contrast

The perception of colour varies based on its surroundings. If a person stares intently at a red square for a time, then suddenly shifts his gaze to a white square, he will for a moment perceive green instead of red. This is called simultaneous contrast: the phenomenon when colours influence the perception of other colours to have greater contrast[?] .

This affects both value and hue. A blue surrounded by grey will cause the grey to appear more like a yellowish grey. The consequence is that there is some danger with using complementary colours: for example, if a figure contains a mix of blues and yellow of varying values, a light yellow surrounding a blue may cause it to appear darker and more saturated, such that it may be confused with the next darkest blue in the colour scheme[?] .

Therefore, when selecting colour schemes which are used in mixed arrangements, avoid complementary colours. This also avoids a perceptual “vibration” which occurs when complementary colours of similar values are neighbours[?] .

### 2.3.5 Data-appropriate

Use appropriate colours for the data at hand. For example, if the data progresses from low to high values, it may be intuitively visualized using a sequential colour scheme, which uses value differences. If the data increases in two directions (such as in the positive and negative directions), use a diverging colour scheme. If there is no intuitive magnitude differences in the data, as is the case with nominal and categorial data, use a qualitative scheme, which relies primarily on differences in hue[?] .

The rainbow colour map is often used in visualizations for sequential data, however it is not a good choice[?] . Although the colours in a rainbow *are* ordered based on wavelength of light, this ordering is not perceptual (unlike value-based ordering). In addition, this colour scheme progresses with uneven perceptual steps, as each hue change is a sharp change, while the progression within a hue is comparatively slow. This colour scheme should be avoided.

Sometimes the data suggests which colour should be used. For example, if a figure compared blue eyes and green eyes, the data points might be coloured blue and green.

In some fields of study, colour conventions are in place. These convention can be used to help make data quickly recognizable to viewers[?] , and at the very least, colour schemes should not reverse conventions, as this would likely lead to confusion.

Whether colour is even necessary at all should be considered[?] . Colour is a relative medium, difficult to interpret accurately, poses problems to people with colour blindness and may result in loss of clarity when images are reproduced in greyscale. It is easy to default to using colour in figures because of its aesthetic value, but it important to consider whether or not colour is the most effective method of communicating data.

## 2.4 File type

There are multiple ways of classifying file types. Understanding the distinctions will help with selecting the appropriate file type.

### 2.4.1 Raster *vs.* vector

Images can either be raster (also known as bitmap) or vector. Raster images are encoded as pixels, which is why they appear pixelated when zoomed in. On the other hand, vector images are represented by mathematical expressions, which means that they can be infinitely scaled without losing resolution.

Photographs are raster images. In order to print a large raster image, the image must have a high resolution (which increases the file size).

Logos are often made to be vector images. This is because they expect to be reproduced at different sizes.

### 2.4.2 Image format

There are many different kinds of image formats available, which will produce either raster or vector images. Some examples of vector images are PDFs and SVGs. Some common bitmap examples include JPEGs, TIFFs, PNGs and GIFs.

Bitmap images can be compressed using either lossless or lossy algorithms. Lossless algorithms decrease file sizes without compromising the quality of the image, whereas lossy algorithms decrease file size by sacrificing image quality. However, lossy algorithms are often able to create a smaller image size. JPEGs generally use a lossy compression algorithm. This means that each time you edit and save a JPEG image, some of the image quality is lost.

A second consideration is what type of colour mode is supported. JPEG, PNG, and GIF use the RGB colour model, which is how colour is generated on digital displays. If an image is intended to be printed, the TIFF format also supports the CMYK mode.

Sometimes an image is very simple and uses very few colours. GIFs are small files and only support 256 colours, making them ideal for these instances.

Sorted by increasing file size:

| File type | Compression | Mode | Colours |
|---|---|---|---|
| GIF | Lossless | RGB | 256 |
| JPEG | Lossy | RGB | millions |
| PNG | Lossless | RGB | millions |
| TIFF | Lossless | RGB, CMYK, etc | millions |

In summary: use TIFFs for print, use PNGs for smaller file size without quality loss, use JPEGs if some quality loss is acceptable, and use GIFs when only few colours are needed.

The file type used by BoutrosLab.plotting.general is determined by adding the extension to the file name. If no extension is specified, the package defaults to creating TIFF files.

## 2.5 Resolution

Image resolution is relevant to raster images which are not scalable. Pixel resolution is the type most commonly referred to when image resolution. This refers to the number of pixels in an inch, also called dots per inch (dpi) and describes the amount of detail an image holds.

Therefore, the image resolution is related to (but not the same as) the image size. If an image has a resolution of 72 dpi, this means that 72 pixels are stored in each inch. If the image is enlarged, the resolution will decrease because the pixel size apparently increases. Conversely, if the image is shrunken, there is option of increasing the resolution.

Different use-cases require different resolutions. Digital displays require lower resolutions than print displays. Most computer monitors are able to display a maximum of 72 dpi, which means that even if an image has a higher resolution, the display will only display a maximum of 72 dpi.

Standard print resolution is often cited as 300 dpi. However, line art and text often need higher resolution to be clear. Journals may specify what resolution is required for figures. BoutrosLab.plotting.general defaults to producing high resolution images of 1600 dpi.

## 3.0 Figure creation reviewed

Figure creation begins with understanding the data. The second step is designing the figure. Then the figure is created. Finally, the figure is assessed and improved.

Imagine a dataset comprising of two groups of subjects. One group runs for one mile, and the other group takes a nap. The heart rate for each subject is measured. How could this data be displayed?

One option is to plot each data point on a scatterplot. The two groups could be distinguished by colour and plotting character. Another option is to use the distribution of heart rates and create two boxplots. There are many ways of showing the same data, and depending on what the intended message is, different methods may be more appropriate.

Plots in BoutrosLab.plotting.general are created using a single function call. Depending on the function, mandatory parameters may be `formula`, `data` and/or `x`. These parameters are used to indicate the data to be plotted and which variables to display.

Additional parameters may be used to adjust font sizes, add background shading, change colour schemes, and more. These parameters are optional and used for customization. The full range of parameters available is conveniently viewable on the BoutrosLab.plotting.general API, together with extended example code of parameter usage.

Once a figure is made, it should be reproduced in its expected display format. For example, if a figure is expected to be printed in a journal, it should be assessed in a print format, and at the expected size. This allows viewers to check font legibility, colour differences, and other subtleties that may otherwise go unnoticed on screens.

## 4.0 Plotting Functions

The following chart-types are available in BoutrosLab.plotting.general:

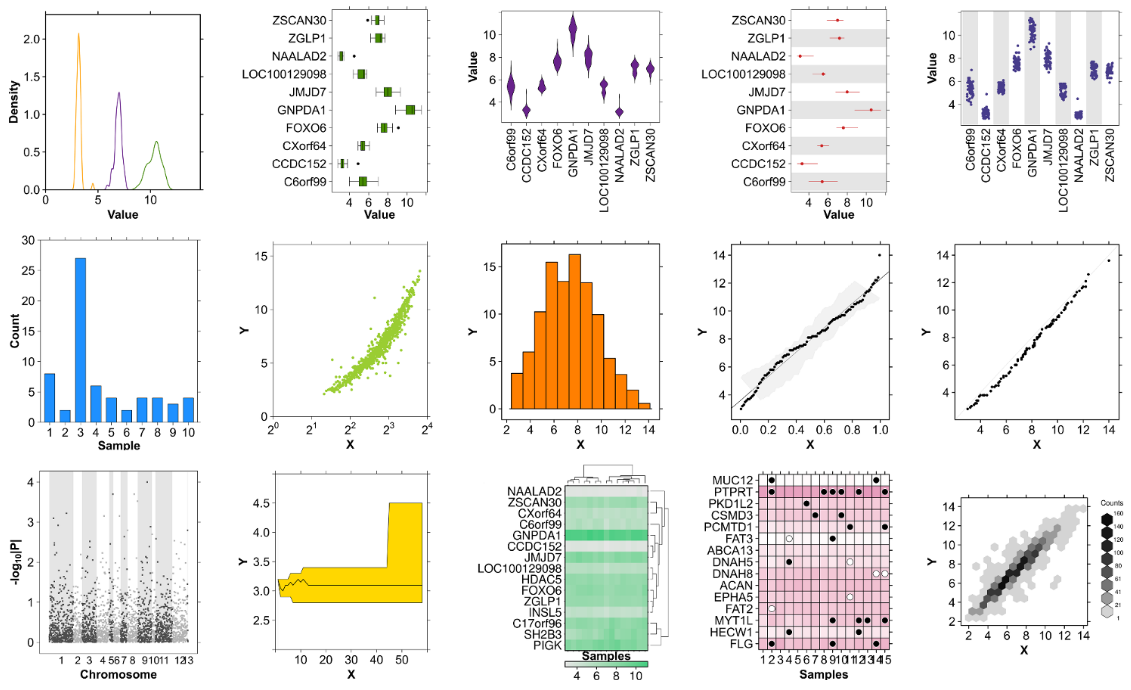


Figure 4: Available chart-types

| Chart-type | Data displayed | Encoding method |
|---|---|---|
| Barplot | Counts or proportions | Length |
| Boxplot | Distributions (summary statistics) | Length, Position |
| Dendrogram | Clustering arrangement | Length, Position |
| Density plot | Distributions | Area |
| Dotmap | Matrix of data points | Area, Colour saturation, Colour hue |
| Heatmap | Matrix of data points (ex. gene expression) | Colour saturation, Colour hue |
| Hexbinplot | Comparison of two variables | Colour saturation, Colour hue |
| Histogram | Distribution (binned into ranges) | Area, Length |
| Manhattan plot | Significance of SNPs at genome locations | Position |
| Multiplot | (variety) | (variety) |
| Polygon plot | Time-series or iterative data | Area |
| Quantile-quantile plot | Comparison of distributions | Position |
| Scatterplot | Comparison of two variables | Position |
| Segplot | Distributions (summary statistics) | Position |
| Strip plot | Distributions | Position |
| Violin plot | Distributions (optional summary statistics) | Area, Length, Position |

Each plotting function is designed to display different kinds of data, and to highlight different relationships in the data.

## 4.1 Scatterplot

`create.scatterplot`

The scatterplot compares two variables on a Cartesian coordinate grid by plotting individual data points. It can be used to investigate correlation between variables, and lines can join the points to emphasize trends. Error bars may be added to each point, and the create.scatterplot function can also determine optimal label positions for points. This versatile function is used to create other plot types, including receiver operating characteristic (ROC) curves.

An interesting note is that scatterplots appear more correlated when scales are increased. This means that when the points in a scatterplot use less of the total plotting space available, a stronger association is perceived compared to when the plotting space is filled[?] .

### 4.1.1 Simple scatterplot

The scatterplot function accepts data in the form of an R data frame. Here is an example of a data frame created using the `microarray` dataset provided with BoutrosLab.plotting.general.

A simple scatterplot has two requirements: the data frame, and a formula to indicate the x and y components of the data frame.

```
> require("knitr")
> opts_chunk$set(fig.path='Examples/', dev='png', warning=FALSE, dpi=120, fig.height=4, fig.width=4, fi
```

```
> library(BoutrosLab.plotting.general);
> scatter.data <- data.frame(
+     sample.one = microarray[1:800,1],
+     sample.two = microarray[1:800,2]
+     );
> create.scatterplot(
+     formula = sample.two ~ sample.one,
+     data = scatter.data
+     );
```

### 4.1.2 Scatterplot with formatted axes

The scatterplot can be customized very easily by specifying the appropriate parameters.

```
> create.scatterplot(
+     formula = sample.two ~ sample.one,
+     data = scatter.data,
+
+     # specify plot title
+     main = "Axes & labels",
+
+     # specify axes titles
+     xlab.label = "Sample 1",
+     ylab.label = "Sample 2",
+
+     # specify appearance of axes tick marks
+     xat = seq(0, 16, 2),
+     yat = seq(0, 16, 2),
+
+     # specify axes ranges
+     xlimits = c(0, 15),
+     ylimits = c(0, 15),
+
+     # specify font sizes of axes titles and tick mark labels
+     xaxis.cex = 1,
+     yaxis.cex = 1,
+     xlab.cex = 1.5,
+     ylab.cex = 1.5,
+     main.cex = 1.5
+     );
```

### 4.1.3 Scatterplot with log-scale axes

The plot can be changed to use a log-scale.

```
> create.scatterplot(
+     formula = sample.two ~ sample.one,
+     data = scatter.data,
+     main = "Log-scale Axis",
+     xlab.label = "Sample 1",
+     ylab.label = "Sample 2",
+     yat = seq(0, 16, 2),
+     ylimits = c(0, 15),
+     xaxis.cex = 1,
+     yaxis.cex = 1,
+     xlab.cex = 1.5,
+     ylab.cex = 1.5,
+     main.cex = 1.5,
+
+     # change axes accordingly
+     xat = 2 ** (0:4),
+
+     # change axes range accordingly
+     xlimits = c(2 ** 0,16),
+
+     # format labels
+     xaxis.lab = c(
+         expression('2'^'0'),
+         expression('2'^'1'),
+         expression('2'^'2'),
+         expression('2'^'3'),
+         expression('2'^'4')
+         ),
+
+     # transform x-axis into log-2 space
+     xaxis.log = 2
+     );
```

### 4.1.4 Scatterplot with gridlines

Additional parameter changes will continue to customize the plot.

```
> create.scatterplot(
+     formula = sample.two ~ sample.one,
+     data = scatter.data,
+     main = "Gridlines",
+     xlab.label = "Sample 1",
+     ylab.label = "Sample 2",
+     xat = seq(0, 16, 2),
+     yat = seq(0, 16, 2),
+     xlimits = c(0, 15),
+     ylimits = c(0, 15),
+     xaxis.cex = 1,
+     yaxis.cex = 1,
+     xlab.cex = 1.5,
+     ylab.cex = 1.5,
+     main.cex = 1.5,
+
+     # plotting character is changed using the "pch" parameter
+     # plotting characters with borders accepts both border colour and fill colour
+     # options are in help files for "pch"
+     pch = 21,
+     col = "black",
+
+     # create gridlines using the "type" parameter
+     # "p" indicates points, and "g" indicates grid
+     # other options for the "type" parameter found in "xyplot" documentation
+     type = c("p", "g")
+     );
```

### 4.1.5 Scatterplot with labelled points

The scatterplot function is able to calculate optimal label positions for labelling individual points.

```
> # determine which point(s) should get labels
> points.x <- c();
> points.y <- c();
> point.labels <- c();
> for (i in 1:800){
+         if(scatter.data$sample.one[i] > (scatter.data$sample.two[i] + 3.5) ||
+          scatter.data$sample.one[i] < (scatter.data$sample.two[i] - 3.5)){
+                 point.labels <- c(point.labels, rownames(microarray)[i])
+                 points.x <- c(points.x, scatter.data$sample.one[i])
+                 points.y <- c(points.y, scatter.data$sample.two[i])
+         }
+ }
> create.scatterplot(
+     formula = sample.two ~ sample.one,
+     data = scatter.data,
+     main = "Point labels",
+     xlab.label = "Sample 1",
+     ylab.label = "Sample 2",
+     xat = seq(0, 16, 2),
+     yat = seq(0, 16, 2),
+     xlimits = c(0, 15),
+     ylimits = c(0, 15),
+     xaxis.cex = 1,
+     yaxis.cex = 1,
+     xlab.cex = 1.5,
+     ylab.cex = 1.5,
+     main.cex = 1.5,
+     pch = 21,
+     col = "black",
+     type = c("p", "g"),
+
+     # allow for auto-placement of labels based on point locations
+     # (otherwise given coordinates will be used)
+     text.guess.labels = TRUE,
+     text.guess.skip.labels = FALSE,
+
+     # specify labels and corresponding points
+     add.text = TRUE,
+     text.x = points.x,
+     text.y = points.y,
+     text.cex = 0.8,
+     text.col = "red",
+     text.labels = point.labels,
+
+     # give special points custom pch and colour
+     # these are created by drawing extra points over top the existing points
+     # another option is to provide a vector to the "pch" and "col" parameters
+     add.points = TRUE,
+     points.x = points.x,
+     points.y = points.y,
+     points.col = "red",
+     points.pch = 17,
+     points.cex = 1.5
+     );
```

### 4.1.6 Scatterplot with correlation key

A correlation key legend can be added.

```
> create.scatterplot(
+     formula = sample.two ~ sample.one,
+     data = scatter.data,
+     main = "Correlation key",
+     xlab.label = "Sample 1",
+     ylab.label = "Sample 2",
+     xat = seq(0, 16, 2),
+     yat = seq(0, 16, 2),
+     xlimits = c(0, 15),
+     ylimits = c(0, 15),
+     xaxis.cex = 1,
+     yaxis.cex = 1,
+     xlab.cex = 1.5,
+     ylab.cex = 1.5,
+     main.cex = 1.5,
+     pch = 21,
+     col = "black",
+     type = c("p", "g"),
+     text.guess.labels = TRUE,
+     text.guess.skip.labels = FALSE,
+     add.text = TRUE,
+     text.x = points.x,
+     text.y = points.y,
+     text.cex = 0.8,
+     text.col = "red",
+     text.labels = point.labels,
+     add.points = TRUE,
+     points.x = points.x,
+     points.y = points.y,
+     points.col = "red",
+     points.pch = 17,
+     points.cex = 1.5,
+
+     # add correlation key using get.corr.key function
+     legend = list(
+         inside = list(
+             fun = draw.key,
+             args = list(
+                 key = get.corr.key(
+
+                     # two vectors of the same length
+                     x = scatter.data$sample.one,
+                     y = scatter.data$sample.two,
+
+                     # specify what to include in the key
+                     label.items = c('spearman','kendall','beta1'),
+
+                     # format key
+                     alpha.background = 0,
+                     key.cex = 1
+                     )
+                 ),
+
+             # place key
+             x = 0.03,
+             y = 0.95,
+             corner = c(0,1)
+             )
+         )
+     );
```

### 4.1.7 Scatterplot with lines

Adding lines can be done in multiple ways.

```
> create.scatterplot(
+     formula = sample.two ~ sample.one,
+     data = scatter.data,
+     main = "Added lines",
+     xlab.label = "Sample 1",
+     ylab.label = "Sample 2",
+     xat = seq(0, 16, 2),
+     yat = seq(0, 16, 2),
+     xlimits = c(0, 15),
+     ylimits = c(0, 15),
+     xaxis.cex = 1,
+     yaxis.cex = 1,
+     xlab.cex = 1.5,
+     ylab.cex = 1.5,
+     main.cex = 1.5,
+     pch = 21,
+     col = "black",
+
+     # "r" indicates the addition of a linear regression line
+     type = c("p", "g", "r"),
+     text.guess.labels = TRUE,
+     text.guess.skip.labels = FALSE,
+     add.text = TRUE,
+     text.x = points.x,
+     text.y = points.y,
+     text.cex = 0.8,
+     text.col = "red",
+     text.labels = point.labels,
+     add.points = TRUE,
+     points.x = points.x,
+     points.y = points.y,
+     points.col = "red",
+     points.pch = 17,
+     points.cex = 1.5,
+     legend = list(
+         inside = list(
+             fun = draw.key,
+             args = list(
+                 key = get.corr.key(
+                     x = scatter.data$sample.one,
+                     y = scatter.data$sample.two,
+                     label.items = c('spearman','kendall','beta1'),
+                     alpha.background = 0,
+                     key.cex = 1
+                     )
+                 ),
+             x = 0.03,
+             y = 0.95,
+             corner = c(0,1)
+               )
+           ),
+
+           # draw y = x line
+           add.xyline = TRUE,
+           xyline.col = "red"
+       );
```

## 4.2 Boxplot

`create.boxplot`

The boxplot is used to display distributions of data. It is also known as the box-and-whisker plot. The 'box' ranges from the first to the third quartile, showing the interquartile range. The line within the box indicates the median (not the mean), and whiskers have variable meaning, such as the maximum and minimum of the data (but they are not error bars; they are not the s.d. or s.e.m). The meaning of whiskers should be indicated, and outliers may be plotted as points? . BoutrosLab.plotting.general defaults to setting the whiskers to the most extreme data point within 1.5 × IQR of the edge of the box.

### 4.2.1 Simple boxplot

The boxplot also requires a data frame and a corresponding formula.

```
> boxplot.data <- data.frame(
+     x <- as.vector(t(microarray[1:10,1:58])),
+     y <- as.factor(rep(rownames(microarray[1:10,1:58]), each = 58))
+     );
> create.boxplot(
+     formula = y ~ x,
+     data = boxplot.data,
+     xaxis.cex = 1.2,
+     yaxis.cex = 1.2
+     );
```

### 4.2.2 Boxplot with colour

Customization of boxplot parameters is done by specifying the appropriate parameters.

```
> create.boxplot(
+     formula = y ~ x,
+     data = boxplot.data,
+     xaxis.cex = 1.2,
+     yaxis.cex = 1.2,
+
+     # specify box colour
+     col = default.colours(1)
+     );
```

### 4.2.3 Boxplot with background shading

Coloured rectangles can be drawn in plots.

```
> create.boxplot(
+     formula = y ~ x,
+     data = boxplot.data,
+     col = default.colours(1),
+     xaxis.cex = 1.2,
+     yaxis.cex = 1.2,
+
+     # draw rectangles
+     add.rectangle = TRUE,
+
+     # coordinates of rectangles given by four parameters
+     xleft.rectangle = 0,
+     xright.rectangle = 13,
+     ybottom.rectangle = seq(0.5, 8.5, 2),
+     ytop.rectangle = seq(1.5, 9.5, 2),
+
+     # set rectangle colour
+     col.rectangle = "grey",
+
+     # set rectangle alpha (transparency)
+     alpha.rectangle = 0.25
+     );
```

### 4.2.4 Sorted boxplot

To make comparisons clearer, the boxplots can be sorted by median value.

```
> create.boxplot(
+     formula = y ~ x,
+     data = boxplot.data,
+     col = default.colours(1),
+     xaxis.cex = 1.2,
+     yaxis.cex = 1.2,
+     add.rectangle = TRUE,
+     xleft.rectangle = 0,
+     xright.rectangle = 13,
+     ybottom.rectangle = seq(0.5, 8.5, 2),
+     ytop.rectangle = seq(1.5, 9.5, 2),
+     col.rectangle = "grey",
+     alpha.rectangle = 0.25,
+
+     # sort median values in either increasing or decreasing order
+     sample.order = "increasing"
+     );
```

### 4.2.5 Boxplot with stripplot

The individual points can be plotted on the boxplot.

```
> create.boxplot(
+     formula = y ~ x,
+     data = boxplot.data,
+     col = default.colours(1),
+     xaxis.cex = 1.2,
+     yaxis.cex = 1.2,
+     add.rectangle = TRUE,
+     xleft.rectangle = 0,
+     xright.rectangle = 13,
+     ybottom.rectangle = seq(0.5, 8.5, 2),
+     ytop.rectangle = seq(1.5, 9.5, 2),
+     col.rectangle = "grey",
+     alpha.rectangle = 0.25,
+     sample.order = "increasing",
+
+     # add points
+     add.stripplot = TRUE,
+     points.pch = 1
+     );
```

## 4.3 Heatmap

`create.heatmap`

The heatmap is a matrix with values represented by colour value and hue. It is often used to display gene expression levels compared against samples. One strength of the heatmap is that it is able to display many data points with more dimensions: it displays a matrix of values where each cell encodes a value through colour, and is associated with both an x-axis and y-axis value.

However, colour is a subjective method of conveying data. It is useful for displaying general trends and comparing values, but is not accurate for conveying exact values. Neighbouring colours change the perception of a colour such that a particular colour surrounded by lighter colours will appear darker, and vice versa. This means that two cells in a heatmap which are the same colour can appear to be different colours because of the cells surrounding them[?] . One option for minimizing this bias is by clustering the heatmap so that similar colours will neighbour each other. Another option is to add grid lines to help distinguish the neighbouring colours.

### 4.3.1 Simple heatmap

The minimal input for a heatmap is either a data frame or matrix.

```
> create.heatmap(
+     x = microarray[1:20, 1:20]
+     );
```

### 4.3.2 Adding formatted axes

Axes formatting can help the appearance of the plot. The heatmap function estimates an appropriate font size based on the size of the data set.

```
> create.heatmap(
+     x = microarray[1:20, 1:20],
+
+     # format the colour key
+     colourkey.cex = 1,
+     colourkey.labels.at = seq(2, 12, 1),
+
+     # set labels to NA -- results in default labels
+     xaxis.lab = NA,
+     yaxis.lab = NA,
+
+     xaxis.cex = 0.8,
+     yaxis.cex = 0.6,
+
+     # set font style (default is bold, 1 is roman)
+     xaxis.fontface = 1,
+     yaxis.fontface = 1
+     );
```

### 4.3.3 Changing axes labels

The axes labels can be customized.

```
> create.heatmap(
+     x = microarray[1:20, 1:20],
+     colourkey.cex = 1,
+     colourkey.labels.at = seq(2, 12, 1),
+     xaxis.lab = NA,
+     xaxis.cex = 0.8,
+     yaxis.cex = 0.6,
+     xaxis.fontface = 1,
+     yaxis.fontface = 1,
+
+     # set custom labels
+     yaxis.lab = letters[1:20]
+     );
```

### 4.3.4 Changing the clustering method

The heatmap defaults to clustering using the“diana" method (divisive analysis clustering). Other clustering options include all agglomerative clustering methods available in hclust.

```
> create.heatmap(
+     x = microarray[1:20, 1:20],
+     colourkey.cex = 1,
+     colourkey.labels.at = seq(2, 12, 1),
+     xaxis.lab = NA,
+     yaxis.lab = NA,
+     xaxis.fontface = 1,
+     yaxis.fontface = 1,
+     xaxis.cex = 0.8,
+     yaxis.cex = 0.6,
+
+     # specify clustering method
+     # if no clustering is desired, set this to "none"
+     clustering.method = "complete",
+
+     # select distance measure
+     rows.distance.method = "euclidean",
+     cols.distance.method = "manhattan"
+     );
```

### 4.3.5 Heatmap with stratified clustering

Heatmap clustering can be stratified so that segments of the data can be clustered separately.

```
> create.heatmap(
+     x = microarray[1:20, 1:20],
+     colourkey.cex = 1,
+     colourkey.labels.at = seq(2, 12, 1),
+     xaxis.lab = NA,
+     yaxis.lab = NA,
+     xaxis.fontface = 1,
+     yaxis.fontface = 1,
+      xaxis.cex = 0.8,
+     yaxis.cex = 0.6,
+     clustering.method = "complete",
+     rows.distance.method = "euclidean",
+     cols.distance.method = "manhattan",
+
+     # specify groups. In this example clustering is done by columns
+     stratified.clusters.cols = list(c(1:10), c(11:20)),
+
+     # add line to show division between two strata
+     grid.col = TRUE,
+     col.lines = 10.5,
+     col.lwd = 5,
+     col.colour = "red"
+     );
```

### 4.3.6 Heatmap with covariates

Covariate bars are often added to heatmaps to add additional information. In this example, the sex of each sample is indicated using a covariate bar.

```
> # create covariate bar to indicate sex
> sample.covariate <- list(
+     rect = list(
+         col = "black",
+         fill = force.colour.scheme(patient$sex[1:20], scheme = "sex"),
+         lwd = 1.5
+         )
+     );
> # create legend to match covariate bar
> sample.cov.legend <- list(
+     legend = list(
+         colours =  force.colour.scheme(c("male",  "female"), scheme = "sex"),
+         labels = c("male", "female"),
+         title = "Sex"
+         )
+     );
> create.heatmap(
+     x = microarray[1:20, 1:20],
+     colourkey.cex = 1,
+     colourkey.labels.at = seq(2, 12, 1),
+     xaxis.lab = NA,
+     xaxis.fontface = 1,
+     xaxis.cex = 0.6,
+     clustering.method = "complete",
+     rows.distance.method = "euclidean",
+     cols.distance.method = "manhattan",
+     stratified.clusters.cols = list(c(1:10), c(11:20)),
+     grid.col = TRUE,
+     col.lines = 10.5,
+     col.lwd = 5,
+     col.colour = "red",
+
+     # add covariates and corresponding legend
+     covariates = sample.covariate,
+     covariate.legend = sample.cov.legend,
+     legend.side = "right",
+
+     # customize covariate bar
+     covariates.grid.border = list(col = "red", lwd = 3)
+     );
```

### 4.3.7 Heatmap with grid lines

Using grid lines can help to more accurately perceive the differences between colours.

```
> create.heatmap(
+     x = microarray[1:20, 1:20],
+     colourkey.cex = 1,
+     colourkey.labels.at = seq(2, 12, 1),
+     xaxis.lab = NA,
+     xaxis.fontface = 1,
+     xaxis.cex = 0.8,
+     clustering.method = "complete",
+     rows.distance.method = "euclidean",
+     cols.distance.method = "manhattan",
+     stratified.clusters.cols = list(c(1:10), c(11:20)),
+
+     # turn grid lines on
+     grid.row = TRUE,
+     grid.col = TRUE,
+
+     # set grid line colour
+     row.colour = "white",
+     col.colour = c(rep("white", 10), "red", rep("white",10)),
+
+     # set grid line width
+     row.lwd = 1,
+     col.lwd = c(rep(1, 10), 5, rep(1,10))
+     );
```

### 4.3.8 Heatmap with discrete colours

Sometimes heatmaps are used to display data with can be divided into discrete bins. In these cases, it is appropriate to use discrete colour schemes. The example below illustrates how a discrete colour scheme is created.

```
> create.heatmap(
+     x = microarray[1:20, 1:20],
+     colourkey.cex = 1,
+     colourkey.labels.at = seq(2, 12, 1),
+     xaxis.lab = NA,
+     xaxis.fontface = 1,
+     xaxis.cex = 0.8,
+     clustering.method = "complete",
+     rows.distance.method = "euclidean",
+     cols.distance.method = "manhattan",
+     stratified.clusters.cols = list(c(1:10), c(11:20)),
+     grid.row = TRUE,
+     grid.col = TRUE,
+     row.colour = "white",
+     col.colour = c(rep("white", 10), "red", rep("white",10)),
+     row.lwd = 0.75,
+     col.lwd = c(rep(0.75, 10), 3, rep(0.75,10)),
+
+    # set colour scheme
+    # set more "total.colours" than colours in "colour.scheme" to create discretized gradient
+    colour.scheme = default.colours(5, palette.type = "spiral.sunrise"),
+
+    # specify how many total colours (add one for a null colour)
+    # in this example, 5 colours will be displayed
+    total.colours = 6
+    );
```

### 4.3.9 Heatmap with symbols

Symbols can be added over the cells.

```
> create.heatmap(
+     x = microarray[1:20, 1:20],
+     colourkey.cex = 1,
+     colourkey.labels.at = seq(2, 12, 1),
+     xaxis.lab = NA,
+     xaxis.fontface = 1,
+     xaxis.cex = 0.8,
+     clustering.method = "complete",
+     rows.distance.method = "euclidean",
+     cols.distance.method = "manhattan",
+     stratified.clusters.cols = list(c(1:10), c(11:20)),
+     grid.row = TRUE,
+     grid.col = TRUE,
+     row.colour = "white",
+     col.colour = c(rep("white", 10), "red", rep("white",10)),
+     row.lwd = 0.75,
+     col.lwd = c(rep(0.75, 10), 3, rep(0.75,10)),
+    colour.scheme = default.colours(5, palette.type = "spiral.sunrise"),
+    total.colours = 6,
+
+    # add symbols using one method
+    row.pos = which(microarray[1:20, 1:20] > 11, arr.ind = TRUE)[,2],
+    col.pos = which(microarray[1:20, 1:20] > 11, arr.ind = TRUE)[,1],
+    cell.text = rep("x", times = sum(microarray[1:20, 1:20] > 11)),
+    text.col = "red",
+    text.cex = 0.9
+    );
```

## 4.4 Barplot

`create.barplot`

The barplot is designed to display counts of categorical data. This means data which is divided into discrete bins.

When displaying data with uncertainty, therefore suggesting the addition of error bars, barplots are not an optimal choice (although, it *is* possible to add error bars to barplots using BoutrosLab.plotting.general). Barplots are not designed for displaying distributions, and error bars might cover ranges never observed in the data: it is advised that using a boxplot or a strip plot might be more appropriate? .

### 4.4.1 Simple barplot

A minimal barplot requires a data frame and corresponding formula.

```
> # set up data
> total.counts <- apply(SNV[1:15], 2, function(x){ mutation.count <- (30 - sum(is.na(x)))});
> count.nonsyn <- function(x){ mutation.count <- length(which(x == 1)); }
> nonsynonymous.SNV <- apply(SNV[1:15], 2, count.nonsyn);
> other.mutations <- total.counts - nonsynonymous.SNV;
> # create data frame
> barplot.data <- data.frame(
+     samples = rep(1:15, 2),
+     mutation = c(rep("nonsynonymous", 15), rep("other",15)),
+     values = c(nonsynonymous.SNV, other.mutations)
+     );
> # create simple plot
> create.barplot(
+     formula = values ~ samples,
+     data = barplot.data[barplot.data$mutation == "nonsynonymous",],
+     xaxis.lab = LETTERS[1:15]
+     );
```

### 4.4.2 Barplot with colours and line

Select bar colours can be changed.

```
> # select bars
> tall.bars <- which(barplot.data[barplot.data$mutation == "nonsynonymous",]$values > 5);
> colour.scheme <- rep("grey", 15);
> colour.scheme[tall.bars] <- "red";
> create.barplot(
+     formula = values ~ samples,
+     data = barplot.data[barplot.data$mutation == "nonsynonymous",],
+     xaxis.lab = LETTERS[1:15],
+
+     # add line
+     abline.h = 5,
+     abline.col = "red",
+     abline.lty = "dotted",
+     abline.lwd = 2,
+
+     # change colours
+     col = colour.scheme
+     );
```

### 4.4.3 Stacked barplot with legend

Stacked barplots compare overall quantities across items, and also indicate the contribution of sub-categories to the total count? .

```
> create.barplot(
+     formula = values ~ samples,
+     data = barplot.data,
+     xaxis.lab = LETTERS[1:15],
+
+     # declare that the plot will be stacked, not grouped
+     stack = TRUE,
+
+     # specify data division
+     groups = mutation,
+
+     # set different colours for the groups
+     col = default.colours(2, palette.type = "pastel"),
+
+     # create a legend
+     legend = list(
+         inside = list(
+             fun = draw.key,
+             args = list(
+                 key = list(
+
+                     # draw legend points
+                     points = list(
+                         col = "black",
+                         pch = 22,
+                         cex = 2,
+                         # reverse order to match stacked bar order
+                         fill = rev(default.colours(2, palette.type = "pastel"))
+                         ),
+
+                     # draw legend text
+                     text = list(
+                         # reverse order to match stacked bar order
+                         lab = rev(c("Nonsynonymous SNV", "Other SNV"))
+                         ),
+                     padding.text = 3,
+                     cex = 1
+                     )
+                 ),
+             # place legend
+             x = 0.25,
+             y = 0.95
+             )
+         )
+     );
```

### 4.4.4 Grouped barplot with legend

Grouped barplots focus on comparing values across sub-categories, and allow for comparison across items? .

```
> create.barplot(
+     formula = values ~ samples,
+     data = barplot.data,
+     xaxis.lab = LETTERS[1:15],
+
+     # specify data division
+     # defaults to creating grouped bar plots when this is set
+     groups = mutation,
+
+     # set group colours
+     col = default.colours(2, palette.type = "pastel"),
+
+     # create legend
+     legend = list(
+         inside = list(
+             fun = draw.key,
+             args = list(
+                 key = list(
+
+                     # Draw legend points
+                     points = list(
+                         col = "black",
+                         pch = 22,
+                         cex = 2,
+                         fill = default.colours(2, palette.type = "pastel")
+                         ),
+
+                     # draw legend text
+                     text = list(
+                         lab = c("Nonsynonymous SNV", "Other SNV")
+                         ),
+
+                     # space items
+                     padding.text = 3,
+                     cex = 1
+                     )
+                 ),
+             # place legend
+             x = 0.25,
+             y = 0.95
+             )
+         )
+     );
```

## 4.5 Density plot

create.densityplot

The density plot also shows data distributions. It is an empirical estimate of the probability density function, and the area under the curve is the cumulative distribution function. It can be thought of as a smoothed version of the histogram.

In comparison to the boxplot, the density plot can compare fewer distributions because after a certain point the lines will interfere with one another. Additionally, summary statistics are not directly provided by the density plot. However, the density plot reveals more subtleties in the “shape" of the data, which may be hidden in the boxplot.

### 4.5.1 Simple density plot

The minimal input required by a density plot is a list of vectors.

```
> create.densityplot(
+     x = as.data.frame(t(microarray[1:3,1:58]))
+     );
```

### 4.5.2 Density plot with line type

Different colours and line types can be used to better distinguish between the lines

```
> create.densityplot(
+     x = as.data.frame(t(microarray[1:3,1:58])),
+
+     # line type
+     lty = c("solid", "dashed", "dotted"),
+
+     # colours
+     col = default.colours(3)
+     );
```

## 4.6 Dendrogram

`create.dendrogram`

The dendrogram is a tree diagram used to display the clustering arrangement determined by hierarchical clustering. It is often used in conjunction with heatmaps to indicate the clustering of genes or samples. In BoutrosLab.plotting.general, this function is called by the `create.heatmap` function when cells are clustered.

The distance from a leaf node to a node indicates the correlation to other leaf nodes, where further distances indicate less correlation. Closely clustered leaf nodes (located at the bottom of the dendrogram) are highly correlated.

### 4.6.1 Simple dendrogram

This function is not structured the same way as other plotting functions. It does not create Trellis objects, and is not usually plotted alone.

```
> dendrogram <- create.dendrogram(x = microarray[1:20, 1:20]);
> plot(x = dendrogram);
```

## 4.7 Dotmap

`create.dotmap`

The dotmap is a novel chart-type found in BoutrosLab.plotting.general. It is a matrix of dots, where the dot sizes and dot colours are used to encode information. The matrix can also be filled with a background colour, similar to heatmaps. This plot is very useful for displaying high-dimensional data because it is able to display two dimensions more than a heatmap through dot size and dot colour.

One example of how to use this plot is to use the dot size to indicate magnitude of fold change, dot colour to show the direction of change, and the background colour to indicate p-values.

### 4.7.1 Simple dotmap

A basic dotmap takes a data frame as input.

```
> # Generate data
> set.seed(12345);
> dotmap.data <- data.frame(
+     "A" = runif(n = 10, min = -1, max = 1),
+     "B" = runif(n = 10, min = -1, max = 1),
+     "C" = runif(n = 10, min = -1, max = 1)
+     );
> # functions to decide the spot size and colour can be included
> # if not, default functions will be used
> spot.size.function <- function(x) { 0.1 + (2 * abs(x)); }
> spot.colour.function <- function(x) {
+     colours <- rep("white", length(x));
+     colours[sign(x) == -1] <- default.colours(2, palette.type = "dotmap")[1];
+     colours[sign(x) ==  1] <- default.colours(2, palette.type = "dotmap")[2];
+     return(colours);
+     }
> create.dotmap(
+     x = dotmap.data,
+     yaxis.cex = 1.5,
+     xaxis.cex = 1.5
+     );
```

### 4.7.2 Dotmap with legend

A legend can be added to indicate the meaning of the dot colours and sizes.

```
> create.dotmap(
+     x = dotmap.data,
+     yaxis.cex = 1.5,
+     xaxis.cex = 1.5,
+
+     # use specified spot size and colour functions
+     spot.size.function = spot.size.function,
+     spot.colour.function = spot.colour.function,
+
+     # create a legend matching the dot sizes
+     key = list(
+
+         # indicate which side of the plot the legend will appear
+         space = "right",
+
+         # create points using the spot size and colour functions
+         # this ensures that they match the spots found in the plot
+         points = list(
+             cex = spot.size.function(seq(-1, 1, 0.2)),
+             col = spot.colour.function(seq(-1, 1, 0.2)),
+             pch = 19
+             ),
+
+         # dot labels
+         text = list(
+             lab = c("-1.0", "-0.8", "-0.6", "-0.4", "-0.2", " 0.0", "+0.2",
+                  "+0.4", "+0.6", "+0.8", "+1.0"),
+             cex = 1.5,
+             adj = 1.0,
+             fontface = "bold"
+             )
+         )
+     );
```

### 4.7.3 Dotmap with background

The background of each cell can also have a colour. Colours do not appear the same with and without borders? . Therefore it is advisable to keep the surrounding colours of points consistent between the legend and the plotting space.

```
> # generate background data
> bg.data <- data.frame(
+     "A" = runif(n = 10, min = -1, max = 1),
+     "B" = runif(n = 10, min = -1, max = 1),
+     "C" = runif(n = 10, min = -1, max = 1)
+     );
> create.dotmap(
+     x = dotmap.data,
+     yaxis.cex = 1.5,
+     xaxis.cex = 1.5,
+     spot.size.function = spot.size.function,
+     spot.colour.function = spot.colour.function,
+     key = list(
+         space = "right",
+         points = list(
+             cex = spot.size.function(seq(-1, 1, 0.2)),
+             col = spot.colour.function(seq(-1, 1, 0.2)),
+             pch = 19
+             ),
+         text = list(
+             lab = c("-1.0", "-0.8", "-0.6", "-0.4", "-0.2", " 0.0", "+0.2",
+                 "+0.4", "+0.6", "+0.8", "+1.0"),
+             cex = 1.5,
+             adj = 1.0,
+             fontface = "bold"
+             )
+         ),
+
+     # control spacing at top of key
+     key.top = 1,
+
+     # add borders to points
+     pch = 21,
+     pch.border.col = "white",
+
+     # add the background
+     bg.data = bg.data,
+
+     # add a colourkey
+     colourkey = TRUE,
+
+     # set colour scheme for background data
+     colour.scheme = c("white", "black"),
+
+     # make bg colour scheme a discrete colour scheme, with breaks at these places
+     at = seq(-1, 1, 0.5)
+     );
```

## 4.8 Hexbinplot

`create.hexbinplot`

The hexbinplot is a hexagonally binned plot which uses colour (usually colour value) to plot points. It uses a coordinate grid similar to scatterplots, but while scatterplots display individual data points, the hexbinplot displays the density of data points in a given space on the plot. It might be thought of as similar to a two-dimensional histogram. One use-case for this plot is in circumstances where data points are too densely packed to be distinguishable on a scatterplot.

### 4.8.1 Simple hexbinplot

The minimum input for a hexbinplot is data frame and formula.

```
> hexbin.data <- data.frame(
+     x = microarray[,1],
+     y = microarray[,2]
+     );
> create.hexbinplot(
+     data = hexbin.data,
+     formula = x ~ y,
+     xaxis.cex = 1,
+     yaxis.cex = 1
+     );
```

### 4.8.2 Hexbinplot with custom bins

The bin sizes can be customized to use more standard increments.

```
> create.hexbinplot(
+     data = hexbin.data,
+     formula = x ~ y,
+     xaxis.cex = 1,
+     yaxis.cex = 1,
+
+     # set the number of bins
+     xbins = 15,
+     colourcut = seq(0, 1, length = 9),
+     maxcnt = 160
+     );
```

## 4.9 Histogram

`create.histogram`

The histogram represents the distribution of data. It is an estimation of the probability distribution of a continuous variable. If a histogram had an infinite number of bars, it would have the same shape as a density plot.

Although its appearance is similar to a barplot, the main difference is that a barplot uses discrete data, while a histogram uses continuous data. This means that the 'bins' of a histogram are data-ranges. This points to why the bars in a barplot are often separated by a space, while the bars in a histogram are usually directly next to one another.

### 4.9.1 Simple histogram

The minimal input for the histogram is either a numeric vector, or a formula and data frame.

```
> create.histogram(
+     x = microarray[,1]
+     );
```

## 4.10 Manhattan plot

`create.manhattanplot`

The Manhattan plot is a type of scatterplot for displaying large datasets. It is often used to visualize or locate particularly significant or interesting data values. For example, it may be used to show chromosomal locations along the x-axis, and the negative logarithm of p-values of mutations along the y-axis. This means that the points which are higher on the plot are highly significant, while the points near the 0 axis are less significant.

### 4.10.1 Simple Manhattan plot

This plot accepts a data frame together with a formula for the x and y components. Manhattan plots usually display a large number of data points.

```
> # generate data
> manhattan.data <- data.frame(
+     x = runif(20000, 0, 1),
+     y = 1:20000
+     );
> create.manhattanplot(
+     formula = -log10(x) ~ y,
+     data = manhattan.data,
+     xaxis.cex = 1.5,
+     yaxis.cex = 1.5        ,
+     xaxis.tck = 1
+     );
```

### 4.10.2 Manhattan plot with line

Adding a line can be used to easily view data points above a particular p-value.

```
> create.manhattanplot(
+     formula = -log10(x) ~ y,
+     data = manhattan.data,
+     xaxis.cex = 1.5,
+     yaxis.cex = 1.5        ,
+     xaxis.tck = 1,
+
+     # add horizontal line
+     abline.h = 2,
+
+     # change line type and style
+     abline.col = "red",
+     abline.lwd = 3,
+     abline.lty = "solid"
+     );
```

## 4.11 Polygon plot

create.polygonplot

The polygon plot draws a shape which can represent a data-range. There are many options for how to use this function. It takes as input the maximum and minimum values and optional median values, and fills in the difference to be the polygon ‘shape’. This chart is often used for repeated, iterative data, such as time-series data.

### 4.11.1 Simple polygon plot

The minimum input for a polygon plot is data frame and formula to extract the components. Additional input is needed to determine the maximum and minimum values of the polygon, as well as the optional median values.

```
> # generate test data
> set.seed(12345);
> temp  <- matrix(runif(1010), ncol = 10) + sort(runif(101));
> polygon.data <- data.frame(
+     x = 0:100,
+     max = apply(temp, 1, max),
+     min = apply(temp, 1, min),
+     median = apply(temp, 1, median)
+     );
> # create the simple plot
> create.polygonplot(
+     formula = NA ~ x,
+     data = polygon.data,
+     max = polygon.data$max,
+     min = polygon.data$min,
+
+     # adjust axes limits
+     xlimits = c(0,100),
+     ylimits = c (0,2),
+
+     # add fill colour
+     col = default.colours(1, palette.type = "pastel"),
+
+     # add middle line
+     add.median = TRUE,
+     median = polygon.data$median
+     );
```

## 4.12 Quantile-quantile plot

A quantile-quantile plot compares distributions by plotting them in a scatterplot fashion, where one distribution provides the x-axis values, and the other distribution provides the y-axis values. The closer in appearance the plot is to an y = x line indicates how similar the two distributions are.

`create.qqplot.comparison`

This quantile-quantile plot compares two distributions.

`create.qqplot.fit`

This quantile-quantile plot compares a distribution against a theoretical distribution.

### 4.12.1 Simple QQ plot comparison

The minimum input for the QQ plot comparison is either a list of numeric vectors or a data source and formula.

```
> create.qqplot.comparison(
+     x = list(rnorm(100), rnorm(100))
+     );
```

### 4.12.2 Simple QQ plot fit

The minimum input for the QQ plot fit is either a list of numeric vectors or a data source and formula.

```
> create.qqplot.fit(
+     x = rnorm(300),
+
+     # compare against a uniform distribution
+     distribution = qunif
+     );
```

## 4.13 Segplot

`create.segplot`

The segplot is also known as a forest plot, and is often used to compare the effectiveness of treatments suggested by different studies, but can be applied to a variety of other datasets. The middle value can be used to indicate a point, while the bar may show the surrounding range of values. This kind of plot can be used to compare distributions, where summary statistics are displayed.

### 4.13.1 Simple segplot

The minimum input to a segplot is a data frame and corresponding formula.

```
> # generate data
> set.seed(12345);
> segplot.data <- data.frame(
+     min = runif(10,5,10),
+     max = runif(10,20,25),
+     median = runif(10, 10, 20),
+     labels = as.factor(LETTERS[1:10])
+     );
> create.segplot(
+     formula = labels ~ min + max,
+     data = segplot.data,
+
+     # add middle dots
+     centers = segplot.data$median
+     );
```

### 4.13.2 Segplot with bands

Segplots can be displayed using thick bands.

```
> create.segplot(
+     formula = labels ~ min + max,
+     data = segplot.data,
+
+     # use bands instead of lines
+     draw.bands = TRUE
+     );
```

## 4.14 Strip plot

`create.stripplot`

The strip plot is a type of scatterplot where one axis is discrete, meaning that the points are divided into discrete bins. If points are densely packed, they can be more easily distinguished by applying a jitter. This kind of plot can be used to display distributions, where each data point is shown.

### 4.14.1 Simple strip plot

The minimum input for a strip plot is a data frame and formula indicating x and y components.

```
> stripplot.data <- data.frame(
+     values = c(t(microarray[1:10, 1:58])),
+     genes = rep(rownames(microarray)[1:10], each = 58)
+     );
> create.stripplot(
+     formula = genes ~ values,
+     data = stripplot.data,
+     xaxis.cex = 1,
+     yaxis.cex = 1
+     );
```

### 4.14.2 Strip plot with jitter and shading

Adding jitter to the points can make it easier to distinguish the values in a strip plot.

```
> create.stripplot(
+     formula = genes ~ values,
+     data = stripplot.data,
+     xaxis.cex = 1,
+     yaxis.cex = 1,
+
+     # add jitter
+     # amount of jitter is controlled using jitter.factor and jitter.amount
+     jitter.data = TRUE,
+
+     # draw rectangle
+     add.rectangle = TRUE,
+
+     # coordinates of the rectangles given by four parameters
+     xleft.rectangle = 0,
+     xright.rectangle = 13,
+     ybottom.rectangle = seq(0.5, 8.5, 2),
+     ytop.rectangle = seq(1.5, 9.5, 2),
+
+     # colour rectangles
+     col.rectangle = "grey",
+
+     # set alpha of the rectangle
+     alpha.rectangle = 0.25
+     );
```

## 4.15 Violin plot

`create.violinplot`

The violin plot is a combination of a boxplot and a kernel density plot. The shape of the violin plot shows the density, where wider sections of the plot indicate regions of higher density. Additional optional points within the violins can be used to encode the median and interquartile range. This is a method of comparing distributions where both summary statistics and a visualization of the data is provided. One difference with this method compared to the density plot is that the violin plot does not indicate what the actual density values are: the height or width of bulges in the violin plots is relative, not absolute.

### 4.15.1 Simple violin plot

The minimal input for a violin plot is a data frame and formula.

```
> violin.data <- data.frame(
+     values = c(t(microarray[1:10, 1:58])),
+     genes = rep(rownames(microarray)[1:10], each = 58)
+     );
> create.violinplot(
+     formula = values ~ genes,
+     data = violin.data,
+         xaxis.cex = 1,
+         yaxis.cex = 1,
+
+     # rotate axes labels
+     xaxis.rot = 90
+     );
```

# 5.0 Multiplot

`create.multiplot`

The muliplot is not a chart-type in itself: rather, it is a function to combine multiple plots into a single plot. It enables the creation of complex figures in a programmable way and encourages good design by standardizing elements such as line widths and font size.

## 5.1 Example 1: Simple multiplot

This example creates a figure out of a single bar plot and three heatmaps using the HairEyeColor data set supplied by the R datasets package.

### 5.1.1 Setting up the data

The first step is to format the data.

```
> # put array into a data frame
> haireye <- as.data.frame(HairEyeColor);
> # sort data
> haireye <- haireye[order(-haireye[,4]),];
> # put columns of data frame into matrices for the heatmap
> Hair <- as.matrix(as.numeric(haireye[,1]));
> Eye <- as.matrix(as.numeric(haireye[,2]));
> Sex <- as.matrix(as.numeric(haireye[,3]));
```

### 5.1.2 Creating the bar plot

The code below creates a bar plot showing the number of males and females with a particular hair and eye colour. Proper labels and titles will be added at the end using the create.multiplot function. The plot is not printed to file – instead, it is saved in the `hair.eye.colour.barplot` variable for later use by the multiplot function.

```
> # create barplot for frequency of each type of person
> # formatting at this stage is unimportant because the create.multiplot function call will override an
> hair.eye.colour.barplot <- create.barplot(
+         formula = Freq ~ c(1:32),
+         data = haireye,
+                 xaxis.cex = 0.5,
+
+         # set plot limits from 0 and 80
+         ylimits = c(0.00, 80)
+         );
> # show barplot
> hair.eye.colour.barplot
```

### 5.1.3 Creating the covariate bars

The covariate bars in this example are created using the create.heatmap function. Discrete colour schemes are used in these covariate bars, and each colour used in specified in the colour.scheme parameter. The total number of colours used is specified in the total.col parameter, plus one to account for the white which is displayed in the case of missing values.

```
> hair.heatmap <- create.heatmap(
+         x = Hair,
+         clustering.method = "none",
+         scale.data = FALSE,
+         colour.scheme = c("black", "chocolate4", "orange", "yellow"),
+
+         # show each colour once (i.e, four values, four colours)
+         total.col = 5,
+         grid.col = TRUE,
+         print.colour.key = FALSE,
+
+         # remove y-axis ticks
+         yaxis.tck = 0,
+         height = 1
+         );
> hair.heatmap
```

The same process is followed for the remaining covariate bars.

```
> eye.heatmap <- create.heatmap(
+         x = Eye,
+         clustering.method = "none",
+         scale.data = FALSE,
+         colour.scheme = c("lightsalmon4", "lightskyblue1", "lightsalmon2", "green4"),
+         total.col = 5,
+         grid.col = TRUE,
+         print.colour.key = FALSE,
+         yaxis.tck = 0,
+         height = 1
+         );
> eye.heatmap
```

```
> sex.heatmap <- create.heatmap(
+         x = Sex,
+         clustering.method = "none",
+         scale.data = FALSE,
+         colour.scheme = force.colour.scheme(c("Male","Female"),scheme = "sex"),
+         total.col = 3,
+         grid.col = TRUE,
+         print.colour.key = FALSE,
+         yaxis.tck = 0,
+         height = 1
+         );
> sex.heatmap
```

### 5.1.4 Creating the legend

A legend is created to differentiate the meaning of the heatmaps. This legend has three sections: one for hair colour, eye colour and sex.

```
> # create legend for covariates
> legends <- legend.grob( list(
+
+        # create legend for hair colour
+        legend = list(
+                colours = c("black", "chocolate4", "orange", "yellow"),
+                title = expression(underline("Hair Colour")),
+                labels = c("Black", "Brown", "Red", "Blonde"),
+                size = 3,
+                title.cex = 2,
+                label.cex = 2
+                ),
+
+        # create legend for eye colour
+        legend = list(
+                colours = c("lightsalmon4", "lightskyblue1", "lightsalmon2", "green4"),
+                title = expression(underline("Eye Colour")),
+                labels = c("Brown", "Blue", "Hazel", "Green"),
+                size = 3,
+                title.cex = 2,
+                label.cex = 2
+                ),
+
+        # create legend for sex
+        legend = list(
+                colours = force.colour.scheme(c("Male", "Female"),scheme = "sex"),
+                title = expression(underline("Sex")),
+                labels = c("Male", "Female"),
+                size = 3,
+                title.cex = 2,
+                label.cex = 2
+                )
+        ),
+        title.just = "left",
+        title.fontface = "plain")
```

### 5.1.5 Creating the final figure

After all the plots and legends are created, they are combined using the create.multiplot function. This figure is arranged so the bar plot is the largest and at the top, and the three heatmaps appear underneath to explain what each of the covariate bars represent. The order is achieved through listing the plots in the order in which they should appear (from bottom to top) in the plot.objects parameter. Furthermore, axis labels are only on the barplot, leaving others without labels and tick marks.

```
>  create.multiplot(
+
+          # four plots are listed from bottom to top
+          plot.objects = list(sex.heatmap, eye.heatmap, hair.heatmap, hair.eye.colour.barplot),
+
+          # labels are placed on y-axis side of the multiplot
+          # (tabs/spaces are used to properly place labels -- the default placement is centred on the
+          ylab.label = c("\t", "Number of people", "\t", "\t"),
+          main.key.padding = 2,
+          ylab.cex = 1.25,
+          main.cex = 1.25,
+          yaxis.cex = 1,
+          panel.heights = c(1, 0.1, 0.1, 0.1),
+
+          # set spacing between the plots
+          y.spacing = c(-1, -1, -1),
+          xaxis.lab = NULL,
+
+          # remove axes tick marks
+          xaxis.alternating = 0,
+          yaxis.alternating = 0,
+
+          # set the yaxis labels (only needed on the bar plot)
+          yaxis.lab = list(NULL, NULL, NULL, seq(0, 100, 20)),
+
+          # put the legend on the right side of the multiplot
+          legend = list(right = list(fun = legends)),
+          print.new.legend = TRUE
+          );
```

## 5.2 Example 2: Complex layout

In this example we make a plot to convey gene expression using a more complex layout. This plot consists of two bar plots and two heatmaps. The heatmap takes up most of the figure and represents gene expression level changes in different samples. The top bar plot will represent the amount of sample used, and the side bar plot represents the importance of the gene. (Note that this example is not based upon real data.)

### 5.2.1 Setting up the data

Data is generated for this example:

```
> set.seed(12345);
> # main heatmap data
> heatmap.data <- data.frame(
+     a = rnorm(n = 25, mean = 0, sd = 0.75),
+     b = rnorm(n = 25, mean = 0, sd = 0.75),
+     c = rnorm(n = 25, mean = 0, sd = 0.75),
+     d = rnorm(n = 25, mean = 0, sd = 0.75),
+     e = rnorm(n = 25, mean = 0, sd = 0.75),
+     f = rnorm(n = 25, mean = 0, sd = 0.75),
+     g = rnorm(n = 25, mean = 0, sd = 0.75),
+     h = rnorm(n = 25, mean = 0, sd = 0.75)
+     );
> # colourkey data
> colorkey.data <- data.frame(
+     x <- seq(-50,50,1)
+     );
> # top barplot data
> top.barplot.data <- data.frame(
+     x = rnorm(n = 25, mean = 2, sd = 0.75),
+     y = seq(1,25,1)
+     );
>  # side barplot data
> side.barplot.data <- data.frame(
+     x = rnorm(n = 8, mean = 0, sd = 0.75),
+     y = seq(1,8,1)
+     );
```

### 5.2.2 Creating the main heatmap

Here the heatmap used to display the expression level for each sample and gene is created.

```
> gene.expression.heatmap <- create.heatmap(
+      x = heatmap.data,
+      xaxis.tck = 0,
+      yaxis.tck = 0,
+      colourkey.cex = 1,
+
+      # don't want to cluster this heatmap
+      clustering.method = "none",
+      axes.lwd = 1,
+       ylab.label = "y",
+       xlab.label = "x",
+       yaxis.fontface = 1,
+       xaxis.fontface = 1,
+       xlab.cex = 1,
+       ylab.cex = 1,
+       main.cex = 1,
+       colour.scheme = c("red", "white", "turquoise")
+       );
> gene.expression.heatmap
```

### 5.2.3 Creating the colourkey

Here the colourkey for the main heatmap is created. It displays the same colours as the heatmap, as well as indicate what the colours represent.

```
> key <- create.heatmap(
+     x = colorkey.data,
+     clustering.method = "none",
+     scale.data = FALSE,
+     # set same colours as are in the heatmap
+     colour.scheme = c("turquoise", "white", "red"),
+     print.colour.key = FALSE,
+     yaxis.tck = 0,
+     xat = c(10, 90),
+     xaxis.lab = c("low", "high"),
+     xaxis.rot = 0,
+     xaxis.cex = 2,
+     height = 1
+     );
> key
```

### 5.2.4 Creating the top bar plot

The bar plot at the top of the plot representing the amount of sample is created. The bar plot will look very simple, but details will be added later when the create.multiplot function is called.

```
> sample.barplot <- create.barplot(
+       formula = x~y,
+       data = top.barplot.data,
+       col = "grey",
+
+       # remove lines
+       axes.lwd = 0,
+       border.lwd = 0
+       );
> sample.barplot
```

### 5.2.5 Creating the side bar plot

Here the side bar plot representing the importance of the gene is created. The bar plot will look very simple, but details will be added later when the create.multiplot function is called.

```
> importance.barplot <- create.barplot(
+       formula = x~y,
+       data = side.barplot.data,
+       axes.lwd = 0,
+       border.lwd = 0,
+       col = "grey",
+
+       # The data here is sorted.
+       # if real data was used, would have to ensure that the corresponding
+       # heatmap rows were similarly sorted to match
+       sample.order = "decreasing",
+       plot.horizontal = TRUE
+       );
> importance.barplot
```

### 5.2.6 Creating the final figure

The create.multiplot function is used to combine the plots. First consider the layout: this plot uses 3 rows of 2 columns and skips the bottom right plotting space. The bottom row contains just the colourkey, the middle row contains the main heatmap and the side bar plot, and the top row contains the top barplot. Below is a figure describing the layout; blue represents an area that a plot will appear there, red represents an area that is skipped, and yellow represents an unused area.

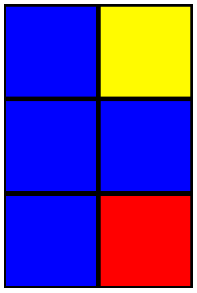


Figure 5: Plot layout

```
> create.multiplot(
+     # use four plots we created earlier as the objects for the multiplot
+     plot.objects = list(key, gene.expression.heatmap, importance.barplot, sample.barplot),
+     panel.heights = c(0.25, 1, 0.05),
+     panel.widths = c(1, 0.25),
+
+     # specify the plot layout
+     # plot.layout specifies the number of rows and columns (3 rows, 2 columns)
+     # layout.skip specifies which of the plots in the specification will be skipped
+     # the skip specification starts at the bottom left, moves to the right and then up
+     plot.layout = c(2, 3),
+     layout.skip = c(FALSE, TRUE, FALSE, FALSE, FALSE, FALSE),
+     xaxis.alternating = 0,
+     yaxis.alternating = 0,
+     xaxis.cex = 1,
+     yaxis.cex = 1,
+     xlab.cex = 1,
+     ylab.cex = 1,
+
+     # format labels (tabs are needed for proper spacing between labels)
+     xlab.label = c("\t", "Samples", "\t", "    Importance"),
+     ylab.label = c( "Amount (g)", "\t", "\t", "Genes", "\t", "\t"),
+     ylab.padding = 6,
+     xlab.to.xaxis.padding = 0,
+     xaxis.lab = list(
+         c("","low","", "","high", ""),
+         LETTERS[1:25],
+         seq(0,5,1),
+         NULL
+         ),
+     yaxis.lab = list(
+         NULL,
+         replicate(8, paste(sample(LETTERS, 4, replace = TRUE), collapse = "")),
+         NULL,
+         seq(0,4,0.05)
+         ),
+     x.spacing = -0.5,
+     y.spacing = c(0, -1),
+     xaxis.fontface = 1,
+     yaxis.fontface = 1
+     );
```

## 5.3 Example 3: Legends

This example focuses on the addition and customization of legends.

### 5.3.1 Create the three scatterplots

```
> for(i in 1:3){
+    # generate data
+         scatter.data <- data.frame(
+                 x = seq(1, 20, 1),
+                 y = rnorm(20, 0.5, 0.1)
+                 );
+
+         tmp.plot <- create.scatterplot(
+                 formula = y ~ x,
+                 data = scatter.data,
+                 type = c("p", "h"),
+
+                 # Set axes limits
+                 xlimits = c(0.5, 20.5),
+                 ylimits = c(0,1),
+
+                 # add background rectangles
+                 add.rectangle = TRUE,
+                 xleft.rectangle = seq(0.5, 18.5, 2),
+                 ybottom.rectangle = 0,
+                 xright.rectangle = seq(1.5, 19.5, 2),
+                 ytop.rectangle = 10,
+                 col.rectangle = c(rep("grey", 6), "indianred1", rep("grey", 2), "skyblue")
+                 );
+
+         if (1 == i) { scatter.one <- tmp.plot; }
+         else if (2 == i) { scatter.two <- tmp.plot; }
+         else if (3 == i) { scatter.three <- tmp.plot; }
+ }
> # display one of the three plots
> scatter.one
```

### 5.3.2 Create covariate bar

```
> # make covariate bars
> covariate.data <- data.frame(
+         license = sample( c("GPL", "AFL", "EULA"), 20, replace = TRUE),
+         runningtime = sample( c("Seconds", "Minutes", "Days", "Weeks", "Months"), 20, replace = TRUE)
+         underdevelopment = sample(c("Yes", "No"), 20, replace = TRUE),
+         language = sample(c("R", "C", "Perl", "Other"), 20, replace = TRUE)
+         );
> # change covariate data to numeric
> covariate.numeric <- data.frame(lapply(covariate.data, as.character), stringsAsFactors = FALSE);
> covariate.numeric$license[covariate.numeric$license == "GPL"] <- 1;
> covariate.numeric$license[covariate.numeric$license == "AFL"] <- 2;
> covariate.numeric$license[covariate.numeric$license == "EULA"] <- 3;
> covariate.numeric$runningtime[covariate.numeric$runningtime == "Seconds"] <- 4;
> covariate.numeric$runningtime[covariate.numeric$runningtime == "Minutes"] <- 5;
> covariate.numeric$runningtime[covariate.numeric$runningtime == "Days"] <- 6;
> covariate.numeric$runningtime[covariate.numeric$runningtime == "Weeks"] <- 7;
> covariate.numeric$runningtime[covariate.numeric$runningtime == "Months"] <- 8;
> covariate.numeric$underdevelopment[covariate.numeric$underdevelopment == "Yes"] <- 9;
> covariate.numeric$underdevelopment[covariate.numeric$underdevelopment == "No"] <- 10;
> covariate.numeric$language[covariate.numeric$language == "R"] <- 11;
> covariate.numeric$language[covariate.numeric$language == "C"] <- 12;
> covariate.numeric$language[covariate.numeric$language == "Perl"] <- 13;
> covariate.numeric$language[covariate.numeric$language == "Other"] <- 14;
> # set colour scheme
> license.colour <- default.colours(4, "spiral.sunrise")[-3];
> runningtime.colour <- default.colours(5, "spiral.dawn");
> underdevelopment.colour <- c("white", "darkgrey");
> language.colour <- default.colours(4, "spiral.afternoon");
> covariate.bar <- create.heatmap(
+         x = data.matrix(covariate.numeric),
+         clustering.method = "none",
+         print.colour.key = FALSE,
+
+         # set colour scheme
+         total.colours = 15,
+         colour.scheme = c(license.colour, runningtime.colour, underdevelopment.colour, language.colou
+
+         # add row lines
+         grid.row = TRUE,
+         grid.col = TRUE,
+         row.colour = "black",
+         col.colour = "black"
+         );
> covariate.bar
```

### 5.3.3 Create legends

```
> # create covariate legend
> covariate.legends <- list(
+     legend = list(
+         colours = license.colour,
+         labels  = c('GPL', 'AFL', 'EULA'),
+         title   = expression(bold('License'))
+         ),
+     legend = list(
+         colours = runningtime.colour,
+         labels  = c("Seconds", "Minutes", "Days", "Weeks", "Months"),
+         title   = expression(bold('Running Time'))
+         ),
+     legend = list(
+         colours = underdevelopment.colour,
+         labels  = c("Yes", "No"),
+         title   = expression(bold('Dev'))
+         ),
+     legend = list(
+         colours = language.colour,
+         labels  = c("R", "C", "Perl", "Other"),
+         title   = expression(bold('Language'))
+         )
+     );
> # create legend grob
> legend1 <- legend.grob(
+     covariate.legends,
+     size = 1.25,
+     label.cex = 0.75,
+     title.cex = 0.75,
+     layout = c(length(covariate.legends), 1),
+
+     # add black box around legend
+     border = list(col = "black", lwd = 3, lty = 1),
+     border.padding = 1.5,
+     between.col = c(1.6, 1.6, 1, 1.6)
+     );
> # create side legend
> side.legend <- list(
+     legend = list(
+         colours = c("indianred1", "skyblue"),
+         labels  = c("Proposed", "Standard")
+         )
+     )
> legend2 <- legend.grob(
+     side.legend,
+     size = 1.25,
+     label.cex = 0.75
+     );
```

### 5.3.4 Create final multiplot

```
> create.multiplot(
+     plot.objects = list(covariate.bar, scatter.three, scatter.two, scatter.one),
+     plot.layout = c(1,4),
+     xaxis.cex = 1.3,
+
+     xaxis.alternating = 0,
+     yaxis.alternating = 0,
+     ylab.label = c('Version 3', 'Version 2', 'Version 1'),
+     yaxis.cex = 1.3,
+     ylab.cex = 1.6,
+     ylab.padding = 5,
+     x.relation = 'free',
+     y.relation = 'free',
+     yat = list(1:4, seq(0, 0.6, 0.2), seq(0, 0.6, 0.2), seq(0, 0.6, 0.2)),
+     yaxis.labels = list(NULL, seq(0, 0.6, 0.2), seq(0, 0.6, 0.2), seq(0, 0.6, 0.2)),
+     xat = seq(1, 20, 1),
+     xaxis.labels = NULL,
+     panel.heights = c(1, 1, 1, 0.2),
+     y.spacing = c(0.5, 0.5, 0.5),
+
+     # provide space for the legend
+     bottom.padding = 15,
+
+     # add legends
+     print.new.legend = TRUE,
+     legend = list(
+
+             # place legend inside plotting space
+         inside = list(
+             fun = legend1,
+             x = 0.5,
+             y = -0.1,
+             corner = c(0.5, 0.5)
+             ),
+
+         # place legend to the right-hand side
+         right = list(
+               fun = legend2
+               )
+         )
+     );
```

## 5.4 Troubleshooting

This section briefly outlines some issues that beginners may have with the multiplot function.

1. The way that the plots are displayed according to the layout you specify may be confusing. The first plot you specify in `plot.objects` is going to be drawn at the bottom left most area for a plot and the ordering will then fill the plots to the right. After reaching the rightmost plot, the plots in the row above will begin to be drawn.

2. Further, while `plot.object` takes in a list of lattice objects and draws them from bottom to up, `panel.heights` takes a vector of elements and assigns it from top most plot to the bottom plot.

3. A common mistake is that the amount of panel widths/heights must be a multiple of the number of plots. The warning will look like:

```
1: In widths.x[pos.widths[[nm]]] <- widths.settings[[nm]] * widths.defaults[[nm]]$x :
number of items to replace is not a multiple of replacement length
2: In widths.x[pos.widths[["panel"]]] <- widths.settings[["panel"]] *  :
number of items to replace is not a multiple of replacement length
```

This may cause the plots to not look as you intend, but is easily fixed by ensuring that your `panel.widths` and `panel.heights` parameters are multiples of the number of plots that can be plotted. For example, if your layout has 2 rows and 3 columns, the `panel.heights` parameter can have 2 inputs, and the `panel.widths` parameter can have 3 inputs.

4. `xaxis.labels`, `xat`, `yaxis.labels`, and `yat` need to take in list of values equal to the number of plot objects. For example, in a 2x2 multiplot, the input is not a list of 2 values, but 4. sv

## 6.0 Other functions

The BoutrosLab.plotting.general package contains a number of non-plotting functions which are included to assist in the plot-generation process.

### 6.1 Colours

There are number of functions to help choose colours for plotting:

- `default.colours` provides a number of pre-made colour palettes to use
- `force.colour.scheme` similarly provides colour palettes, but for specific use-cases
- `colour.gradient` generates sequential colour schemes
- `display.colours` gives a preview of a colour scheme along with grey scale values and colour names

The available colour schemes can be previewed simply using the `display.colours` and `show.available.palettes` functions:

```
# displays the pastel colour scheme
display.colours(default.colours(12, palette.type = "pastel"));

# displays the spiral.dusk colour scheme
display.colours(default.colours(5, palette.type = "spiral.dusk"));

# displays the case-specific biomolecule colour scheme
biomolecule_names <- c("DNA", "RNA", "Protein", "Carbohydrate", "Lipid");
display.colours(force.colour.scheme(biomolecule_names, "biomolecule"), biomolecule_names);
```

```
> # display general schemes
> show.available.palettes(
+     type = 'general'
+     );
```

```
> # display specific schemes
> show.available.palettes(
+     type = 'specific'
+     );
```

The full range of colour schemes available is described in the documentation of the `default.colours` and `force.colour.scheme` functions.

The spiral.sunrise, spiral.dusk, etc colour schemes were created by "spiralling" through the colour wheel, varying the hue, saturation, and value of the colours to achieve a greater perceptual difference[? ?].

### 6.2 Legends

- `legend.grob` and `covariates.grob` create grob objects for legends and covariates respectively
- `create.colourkey` is useful for adding colourkeys to multiplots
- `get.corr.key` creates correlation key legends to be added to plots

## 6.3 Text

- `scientific.notation` function formats numbers into scientific notation
- `display.statistical.result` function uses this to display statistical results in plots
- `get.line.breaks` is used in the specific case of placing indices in heatmaps

## 6.4 Axis Scaling

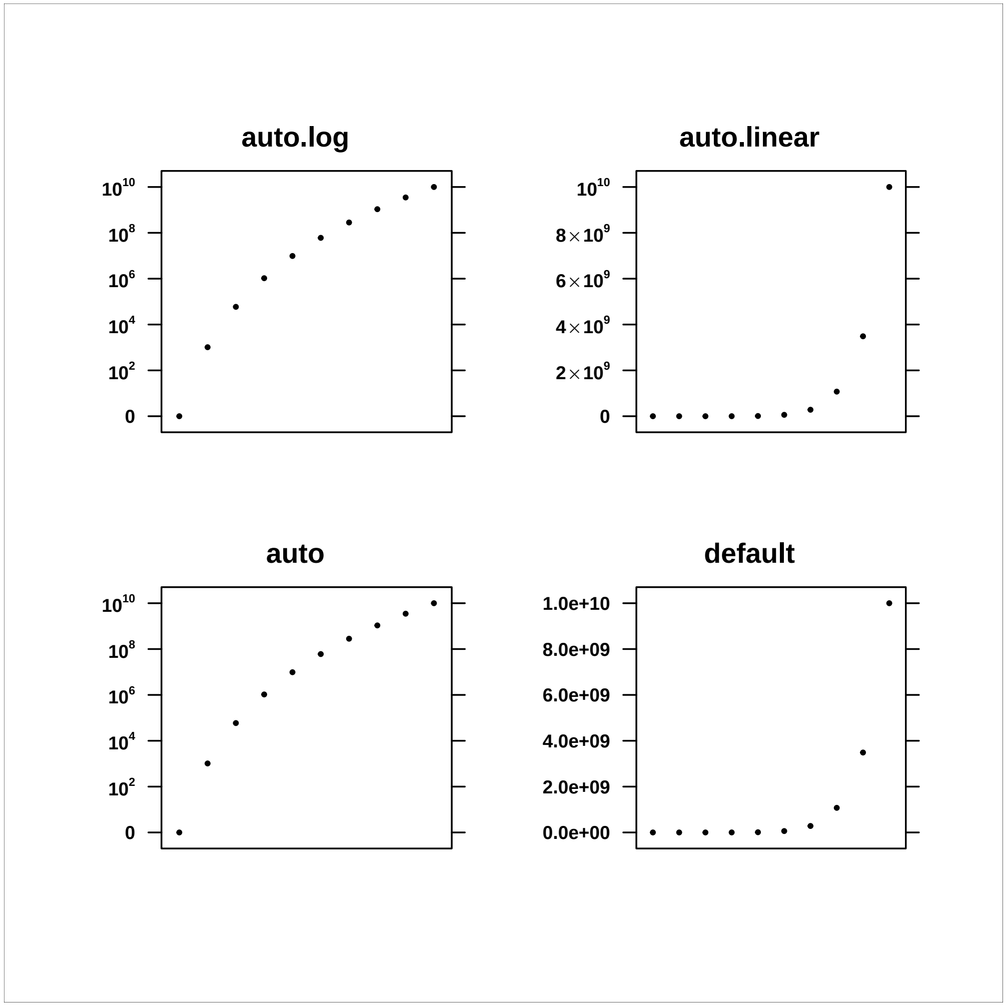
The scale at which data is viewed is extremely impactful to the insight that can be drawn from a visualization. While linear scaling is perfectly fine, some datasets are more easily interpreted after log transforming the data, for example.

### 6.4.1 Automatic Axis Scaling

BPG supports automatic axis scaling for several plot types with the `auto.axis` function. This functionality can be accessed through the `xat` and `yat` parameters when calling a plotting function. There are 3 settings to choose from:

- `auto.log`: $log_{10}$ scaled, with automatically spaced and formatted tick locations.
- `auto.linear`: Linear scaled, with automatic tick placement based on analysis of the data.
- `auto`: Automatically chooses the most appropriate setting (`auto.log` or `auto.linear`).

If nothing is passed, the plot will use lattice's default axis creation method, which is often less "pretty" than BPG's automatic axes.



### 6.4.2 Manual Axis Scaling

It's possible to manually scale the axes for use cases not supported by `auto.axis`. This is a three step process:

- Transform the data.
- Transform the axis tick locations in `xat` and `yat` to match the data transformation.
- Override the axis tick labels in `xaxis.lab` and `yaxis.lab` to correspond to the pre-transformation values from the data.

## 6.5 Private

Other functions are called by plotting functions but are not intended for use by end-users:

`generate.at.final, get.defaults, write.metadata, write.plot`

# 7.0 Session Info

```
> sessionInfo();

R version 4.3.1 (2023-06-16)
Platform: aarch64-apple-darwin20 (64-bit)
```

```
Running under: macOS Ventura 13.4.1

Matrix products: default
BLAS:   /Library/Frameworks/R.framework/Versions/4.3-arm64/Resources/lib/libRblas.0.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/4.3-arm64/Resources/lib/libRlapack.dylib;  LAPACK vers

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

time zone: America/Los_Angeles
tzcode source: internal

attached base packages:
[1] grid      stats     graphics  grDevices utils     datasets  methods
[8] base

other attached packages:
[1] BoutrosLab.plotting.general_7.0.8 hexbin_1.28.3
[3] cluster_2.1.4                     latticeExtra_0.6-30
[5] lattice_0.21-8                    knitr_1.43

loaded via a namespace (and not attached):
 [1] RColorBrewer_1.1-3 xfun_0.39          e1071_1.7-13       gtable_0.3.4
 [5] glue_1.6.2         gridExtra_2.3      jpeg_0.1-10        deldir_1.0-9
 [9] png_0.1-8          lifecycle_1.0.3    cli_3.6.1          proxy_0.4-27
[13] interp_1.1-4       class_7.3-22       compiler_4.3.1     rstudioapi_0.14
[17] tools_4.3.1        Rcpp_1.0.11        rlang_1.1.1        MASS_7.3-60
```

## 8.0 Resources

Further information on the BoutrosLab.plotting.general package can be found at http://labs.oicr.on.ca/boutros-lab/software/BoutrosLab.plotting.general.

# Bibliography

[1] F.J. Anscombe, Graphics in Statistical Analysis. The American Statistician 27, 17-21 (1973).

[2] Noam Shoresh & Bang Wong, Data exploration. Nature Methods 9, 5 (2012).

[3] Bang Wong, Design of data figures. Nature Methods 7, 665 (2010).

[4] William S. Cleveland & Robert McGill, Graphical Perception and Graphical Methods for Analyzing Scientific Data. Science 229, 828-833 (1985).

[5] Edward R. Tufte, The Visual Display of Quantitative Information. Graphics Press, Cheshire, Connecticut (2001).

[6] Sarkar, Deepayan (2008) Lattice: Multivariate Data Visualization with R, Springer. ISBN: 978-0-387-75968-5 URL: http://lmdvr.r-forge.r-project.org/

[7] Becker, R. A. and Cleveland, W. S., S-PLUS Trellis Graphics User's Manual, Seattle: MathSoft, Inc., Murray Hill: Bell Labs, 1996.

[8] William S. Cleveland, Persi Diaconis & Robert McGill, Variables on Scatterplots Look More Highly Correlated When the Scales Are Increased. Science 216, 1138-1141 (1982).

[9] Marc Streit & Nils Gehlenborg, Bar charts and box plots. Nature Methods 11, 117 (2014).

[10] Mark Harrower & Cynthia A. Brewer, ColorBrewer.org: An Online Tool for Selecting Colour Schemes for Maps. The Cartographic Journal 40, 27-37 (2003).

[11] Penny Rheingans, Task-based Color Scale Design. In: Proceedings of the SPIE–28th AIPR Workshop: 3D Visualization for Data Exploration and Decision Making. 3905, 35-43 (2000).

[12] David Borland & Russell M. Taylor II, Rainbow Color Map (Still) Considered Harmful. IEEE Computer Graphics and Applications 27, 14-17 (2007).

[13] Martin Krzywinski & Naomi Altman, Visualizing samples with box plots. Nature Methods 11, 119-120 (2014).

[14] Bang Wong, Color coding. Nature Methods 7, 573 (2010).

[15] Bang Wong, Typography. Nature Methods 8, 277 (2011).

[16] Robin Williams, The Non-Designer's Design Book, 3rd ed. Peachpit Press, Berkeley, California (2008).

[17] Bang Wong, Layout. Nature Methods 8, 783 (2011).

[18] W. H. Tedford Jr., S. L. Bergquist & W. E. Flynn, The Size-Color Illusion. The Journal of General Psychology 97, 145-149 (1977).

[19] William S. Cleveland & Robert McGill, A Color-Caused Optical Illusion on a Statistical Graph. The American Statistician. 37, 101-105 (1983).

[20] Samuel Silva, Beatriz Sousa Santos & Joaquim Madeira, Using color in visualization: A survey. Computers & Graphics 35, 320-333 (2011).

[21] Josef Albers, Interaction of Color. Yale University Press, New Haven & London (1963).

[22] Cynthia A. Brewer, Guidelines for Selecting Colors for Diverging Schemes on Maps. The Cartographic Journal 22, 79-86 (1996).

[23] Bang Wong, Color blindness. Nature Methods 8, 441 (2011).

[24] Bang Wong, Avoiding color. Nature Methods 8, 525 (2011).