Changes and Additions to ggplot2-0.9.0

ggplot2 Development Team

February 13, 2012

Contents

1 Overview

Version 0.9.0 of the **ggplot2** package contains a number of changes that provide a user with more flexibility and greater ease of use in the construction of a ggplot. The two most evident improvements from a user's perspective are: (i) the help pages have been expanded considerably, with many new examples; and (ii) the computing time has been reduced significantly. Several new geoms are introduced, as well as a few new stat_ functions. These will be described in the sections to follow.

This version marks the inception of a renovation project designed to off-load groups of related functions into separate packages and streamlining the code to produce a more consistent user interface across those sets of functions. The **scales** package is the first evidence of this process, but others will follow in due course. Another important decision made in this process

is to gradually rewrite the core code of the package in S3, which is anticipated to have two important effects: (i) faster execution of code; (ii) greater participation from **ggplot2** users in the development of the package. Moreover, steps have been taken to initiate rigorous unit testing to ensure that new and/or revised code is performing as expected, with the aim of reducing the probability of bugs in the future and faster response time in fixing bugs should they occur. This redesign is part of an effort to transition from a single-developer project to a user community project.

This document is an extension of the NEWS files in the **ggplot2** and **scales** packages, describing the changes and new features in some detail, in the hope that it will ease the transition for existing **ggplot2** users. The changes that will most directly affect users in this version have to do with guides–specifically, positional axes, text annotation and legends.

1.1 Visible changes

- The help files are now fully documented with many new examples in addition to those already in the on-line help pages. Notably, the help page of `opts()` lists all of the available theme options with an extensive set of examples and several help pages are devoted to the comparison of `qplot()` with base, **lattice** and gpl graphics.

- **ggplot2** now has a NAMESPACE, which is required in all R packages as of version 2.14.0. This means, for example, that **plyr** and **reshape** will no longer be autoloaded with **ggplot2**, so you will need to write an `Rprofile` function, either at the site level (in RHOME/etc) or in a project directory, if you want to continue loading **plyr**, **reshape**[2] or other packages at startup. Otherwise, packages must be loaded from the command line.

- Thanks to the efforts of Kohske Takahashi, two new functions named `guide_legend()` and `guide_colorbar()` greatly extend the capability of users to tailor legends in a ggplot. The help pages of each have an extensive set of examples, a few of which will be illustrated in section 2.

- `scale_` functions now have a more consistent set of arguments; for example, they all now have arguments `breaks`, `values`, `limits`, `labels` and `name`. Moreover, arguments `breaks`, `limits` and `labels` now accept a function as its value. The positional guide functions `scale_continuous` and `scale_discrete` have an additional argument `trans` that also accepts a function as its value while the attribute scale functions (e.g.,, `scale_fill_discrete()` or `scale_colour_manual()`) have a new argument `guide = `, which can assume one of the values `"none"` or `"legend"`; for continuous fill and color scales, another option is `"colorbar"`. Examples of these are present in the help files; others will be shown in section 7. This is a significant change in the package that may affect existing user code.

- Several new geoms are introduced in this package: `geom_map()`, `geom_raster()`, `geom_dotplot()` and `geom_violin()`. The latter pair of geoms were developed by Winston Chang. These will be discussed in section 3, but see their individual help files for more detailed description and examples.

- A new set of fortify methods has been written for objects of class `multcomp`. See the `fortify-multcomp` help page for an overview.
- Four new `stat_` functions make their debut:
  - `stat_summary2d()`, which applies a function over a 2D grid of bins;
  - `stat_summary_hex()`, which applies a function over a collection of hexagonal bins;
  - `stat_bindot()`, a `stat_` function that accompanies `geom_dotplot()`;
  - `stat_ydensity()`, a `stat_` function that accompanies `geom_violin()`.

  Each of these has its own help page with accompanying examples.
- Users no longer have to worry about the variable name/aesthetic name buglet that plagued earlier versions, which was a particular problem when a faceting variable was the name of an aesthetic (e.g., colour or size). You can now expect that code like the following will work:

```
ggplot(df, aes(x, y, group = group, colour = color))
```

- The problem with axis titles being too close to axis labels at the default font size has been fixed in 0.9.0. In addition, axis titles are now centered with respect to the panels rather than the overall graphics region.
- An expanded collection of possible linetypes is introduced in this version. See the help page of `aes_linetype_size_shape()` for some examples, and for a brief description, the help page of `scale_linetype()`.
- The `facet_grid()` function has a couple of new features. The `scales =` argument now supports values `free_x` and `free_y`, which allow a user to adjust the spatial scaling of facets in either the horizontal or vertical direction. Moreover, the `labeller =` argument now accepts a function as its value and the `margins =` argument now works again. A number of examples are given on the help page of `facet_grid`, but some examples incorporating these changes will be shown in section 5.
- `geom_boxplot()` is now capable of rendering notched boxplots. In addition, the new default is for the center line of a boxplot to be thicker than the box and whiskers, using the new argument `fatten =` , which defaults to 2. If you want the old behavior, set `fatten = 1` in a `geom_boxplot()` call.
- A new function `ggmissing()` visualizes missing data to investigate the plausibility of the 'missing at random' assumption. Two other new functions are `ggorder()`, which plots data in the order in which they were recorded, and `ggstructure()`, which is designed to highlight structural anomalies in numeric multivariate data should they exist.
- New help pages (`translate_*`) have been written by Jake Russ to show the connections between `qplot()` and base graphics, `qplot()` and `ggplot()`, `qplot()` and the Graphics Production Library (GPL) as well as differences between **ggplot2** and **lattice**.

1.2 Not so visible changes

As noted in the intro, the package is being reorganized and partially rewritten for a number of reasons. These changes don't necessarily affect users *per se*, but they do affect current and future co-developers. Some of the changes in progress include the following:

1. The **roxygen2** package is now being used exclusively for documentation of functions in **ggplot2**. The practical advantage of **roxygen2** is that it encourages self-documentation of a function as it is being developed.

2. A start has been made in applying unit testing to functions in **ggplot2** and its derivative packages using the **testthat** package. As more people contribute to the project, testing code in advance becomes increasingly important. It is also a way of making sure that past bugs are fixed and a means of preventing future bugs. This should make code more reliable even as the functionality of the package grows over time.

3. In a major design change, the core code of the package is gradually being rewritten in S3 as a replacement for the **proto** package, which, quite frankly, few R programmers know well. This decision is expected to encourage more developers to join the project and, incidentally, to make the code run a bit faster.

4. Another major design decision which will have more visible impact in the future is that of disaggregating the code in **ggplot2** into packages of related functions, so that eventually, **ggplot2** will be more like a control center of sorts that picks up necessary pieces from satellite packages and organizes them into a graphic in **ggplot2**. The goal, however, is more ambitious: certain satellite packages should also work with other graphics engines, including base graphics and the **lattice** package. The **scales** package, for example, falls in this category.

2 Legend and colorbar guides

Control over legends is greatly expanded in version 0.9.0, largely due to the efforts of Kohske Takahashi. Two functions are discussed in this section: `guide_legend()` and `guide_colorbar()`. We discuss these first because they will be used extensively in the sections to follow.

In the grammar of graphics, there are two types of guides: scale guides and annotation guides[1]. In the former group lie positional guides, discussed in section 7, and legends, discussed herein. Annotation guides include `geom_text()`, `annotate()` and the new annotation functions in section 4.

Legends are associated with aesthetics corresponding to plot attributes such as color, fill color, size, line type or plotting character shape. One can still use the `scale_*()` functions as before (e.g., `scale_fill_manual()` or `scale_colour_brewer()`), but the new functions provide a user with more control over aspects of the legend display. Both functions have numerous examples in their respective help pages that illustrate each argument at least once.

[1] These include main titles, embedded text in the graphics region using `annotate()` or `geom_text()` and other annotations such as the North arrow in a spatial plot.

Some that may occur commonly in practice are illustrated in the following two subsections. Bear in mind that legend guides are meant to be associated with discrete-valued aesthetics while colorbar guides are meant to correspond to continuous-valued aesthetics.

Each guide function has a number of arguments. The following table lists those that are common to both legend guides and colorbar guides:

Category	**Argument**	**Notes**
title	`title`	legend title
	`title.position`	`"left"`, `"right"`, `"top"`, `"bottom"`
	`title.theme`	uses `theme_text()`
	`title.hjust`	horizontal justification
	`title.vjust`	vertical justification
label	`label`	TRUE ⟶ labels drawn
	`label.position`	`"left"`, `"right"`, `"top"`, `"bottom"`
	`label.theme`	uses `theme_text()`
	`label.hjust`	horizontal justification
	`label.vjust`	vertical justification
key options	`default.unit`	measurement unit
	`direction`	`"horizontal"` or `"vertical"`

The titles and labels of guides can be manipulated in similar ways since both have the same set of arguments. The `title/label.theme` argument allows one to adjust the properties of titles and/or labels that are covered by the `theme_text()` function.

2.1 `guide_legend()`

This function provides a user with several options to control features of a legend guide, including titles, labels, the height and width of legend keys, the direction of the legend and several other options that are useful when a large number of categories exist. We will see that `guide_legend()` is not used as a primary function, but rather as the 'value' of an argument inside another function, much like the various functions from the **scales** package to be discussed in section 6.

The arguments that are unique to `guide_legend()` are:

`keywidth`	width of legend key item
`keylength`	length of legend key item
`override.aes`	(values of) aesthetics to be reset
`nrow`	number of rows in the legend
`ncol`	number of columns in the legend
`byrow`	TRUE ⟶ order key items by row
`bycol`	TRUE ⟶ order key items by column

In a legend guide, one can manipulate the size of individual legend keys and specify the way they are arranged in a legend through the set of arguments above.

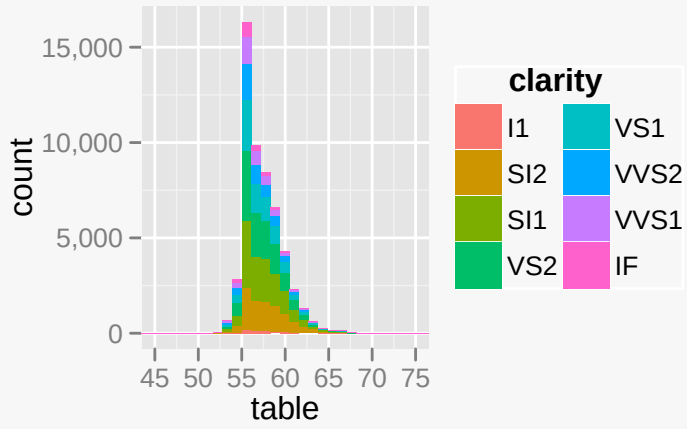
There are several ways to call `guide_legend()` One way to invoke it is in conjunction with the new `guides()` function. In the following example using the `diamonds` data, the

factor clarity is the fill aesthetic for which a legend is to be built. The call to `guide_legend()` is the value of the `fill =` argument of `guides()`. This syntax allows you to adjust one or multiple legends in a single call. Think of `guides()` as analogous to the function `labs()`; we will elaborate on this analogy a bit more in the next section.

The code below creates four rows of legend keys, increases the font size of the legend title and emboldens it:

```
# Base plot
q <- ggplot(diamonds, aes(x = table, fill = clarity)) +
       geom_histogram() +
       scale_y_continuous(labels = comma_format(digits = 5))
# Create two columns of legend labels and increase the
# size of the legend title font
```

```
q + guides(fill = guide_legend(nrow = 4, title.hjust = 0.4,
       title.theme = theme_text(size = 12, face = "bold"))) +
    xlim(45, 75)
```



The same graph can be made by replacing the `guides()` call with one to `scale_fill_discrete()`, where the new `guide =` argument is used to make the legend using a `guide_legend()` function call as its value. The `guide =` argument is specific to scale functions for aesthetics, just as `trans =` is specific to positional scale functions[2].

```
q + scale_fill_discrete(guide = guide_legend(nrow = 4,
       title.hjust = 0.4,
       title.theme = theme_text(size = 12, face = "bold"))) +
    xlim(45, 75)
```

[2]See section 7.

The following example concerns an application that comes up occasionally on the ggplot2 list: overriding the alpha transparency value in the legend so that the values of an aesthetic are visible in the legend key. The relevant argument is `override.aes`: in the left side plot, the very low alpha value makes it impossible to see the legend key items, but `override.aes` resets the alpha value in the legend to make them visible in the right hand plot.

```
(p3 <- qplot(carat, price, data = diamonds, colour = color,
             alpha = I(0.01)) +
         scale_y_continuous(labels = dollar))
p3 + guides(colour = guide_legend(override.aes = list(alpha = 1)))
```

2.2 `guide_colorbar()`

The new function `guide_colorbar()` is designed for continuous ranges of (fill) colors—as its name implies, it outputs a rectangle over which the color gradient varies, while providing more flexibility to the user in terms of the labeling and positioning of guide elements. At this time, it is applicable only to the `scale_fill_continuous()` and `scale_colour_continuous()` functions; in particular, it does not work with size.

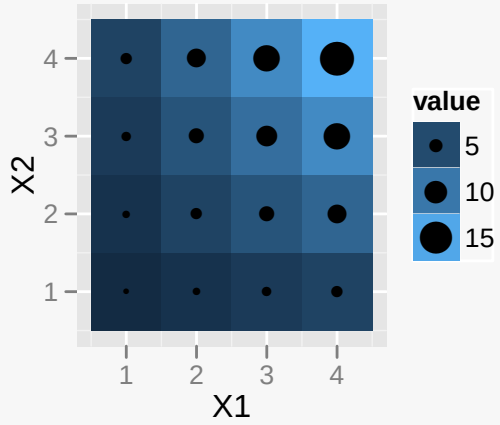
The following table lists the set of arguments unique to `guide_colorbar()`:

`barwidth`	width of the colorbar
`barheight`	height of the colorbar
`nbin`	number of bins to use in the colorbar
`raster`	draw colorbar as a raster image? (Default: TRUE)
`ticks`	should tick marks be drawn? (Default: TRUE)
`draw.llim`	should lower limit tick marks be visible? (Default: FALSE)
`draw.ulim`	should upper limit tick marks be visible? (Default: FALSE)

Whereas legend guides could tinker with the length and height of individual legend keys, a colorbar guide can control the width and height of a colorbar, as well as the number of ticks and whether or not the upper and lower limits of a colorbar should be drawn.

The examples on the `guide_colorbar()` help page show how the function works with `geom_tile()`, so an immediate application of this function is to heatmaps. Here is a simple example from the help page:

```
df2 <- melt(outer(1:4, 1:4), varnames = c("X1", "X2"))
(p1 <- ggplot(df2, aes(X1, X2)) + geom_tile(aes(fill = value)) +
           geom_point(aes(size = value)))
```

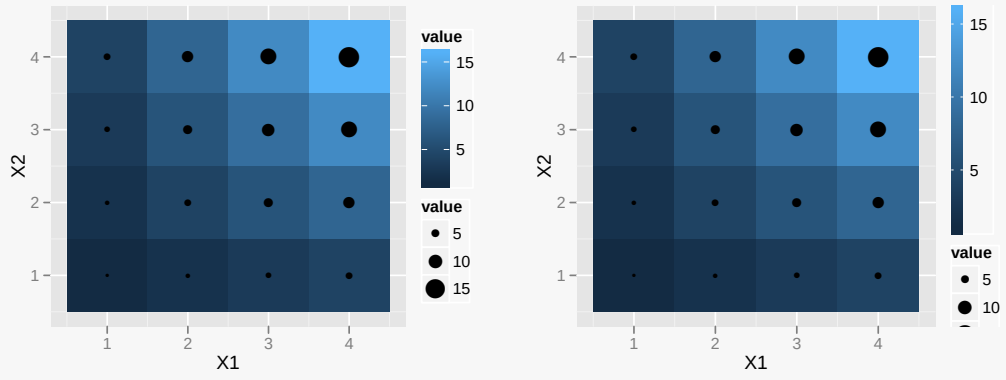


Three equivalent ways to invoke the colorbar are given below. The first simply specifies the guide name; this is sufficient if you want to use the default arguments. Either of the other two should be used when you want to change one or more of the default arguments.

```
p1 + scale_fill_continuous(guide = "colorbar")
p1 + scale_fill_continuous(guide = guide_colorbar())
p1 + guides(fill = guide_colorbar())
```

The following figure illustrates the first and third methods, respectively:

```
p1 + scale_fill_continuous(guide = "colorbar")
p1 + guides(fill = guide_colorbar(barwidth = 0.5, barheight = 10))
```



The examples above indicate that if the fill and size aesthetics are both associated with the same variable and both produce a legend, the legends can be merged into one, ae we saw with the plot of p1 at the top of this subsection. However, if one aesthetic uses a legend guide and another uses a colorbar guide, the two cannot be merged, even if both aesthetics correspond to the same variable. This is seen in the two plots above.

Equivalent ways of calling the left and right side plots above, respectively, are

```
p1 + guides(fill = "colorbar", size = "legend")
p1 + scale_fill_continuous(guide = guide_colorbar(barwidth = 0.5,
                           barheight = 10))
```

Let's return to the analogy between the `labs()` function and the `guides()` function. Recall that one can use `xlab()` and `ylab()` individually in a `ggplot()` call; moreover, one can redefine legend titles with the `name` argument in the appropriate `scale_aesthetic_*()` function. The `labs()` function allows one to define titles for any positional axis or legend in one call, where the aesthetic is the argument name and the title is its value; e.g.,

```
labs(x = "My x-axis", y = "My y-axis", colour = "Category")
```

The `guides()` function is analogous to `labs()` in the sense that one can define a legend or colorbar guide for each aesthetic that generates a guide. The call can be as simple as defining the type of guide for each aesthetic, as in

```
p1 + guides(fill = "colorbar", size = "legend")
```

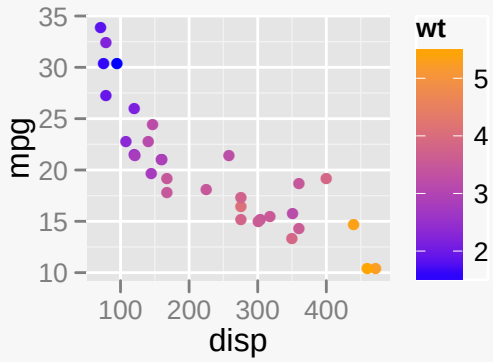
or it can be a more complete specification for each guide, e.g.,

```
p1 + guides(fill = guide_colorbar(nbin = 6, ticks = FALSE),
            colour = guide_legend(label.position = "left"))
```

As in `labs()`, the aesthetic is the argument and the `guide_*()` function is its value. Alternatively, one can use a scale function with a `guide =` argument just as one can use a scale function and specify a legend title with the `name =` argument. The difference is that the value of the `guide =` argument can be a text string that specifies the guide (assuming its default arguments) or it can be a call to one of `guide_legend()` or `guide_colorbar()` wherein one or more default arguments can be modified.

Colorbars can also be used with a continuous color aesthetic; here is an example from the built-in data frame `mpg`:

```
ggplot(mtcars, aes(x = disp, y = mpg, colour = wt)) +
    geom_point() +
    scale_colour_gradient(low = "blue", high = "orange",
                          guide = "colorbar")
```



In this case, we wanted to use the `colour_gradientn()` scale function to determine the color range. The `guide =` argument in any of the `scale_colour_*()` functions allows one to specify a colorbar in place of a legend. If no legend or colorbar is desired, you can substitute `guide = "none"` instead.

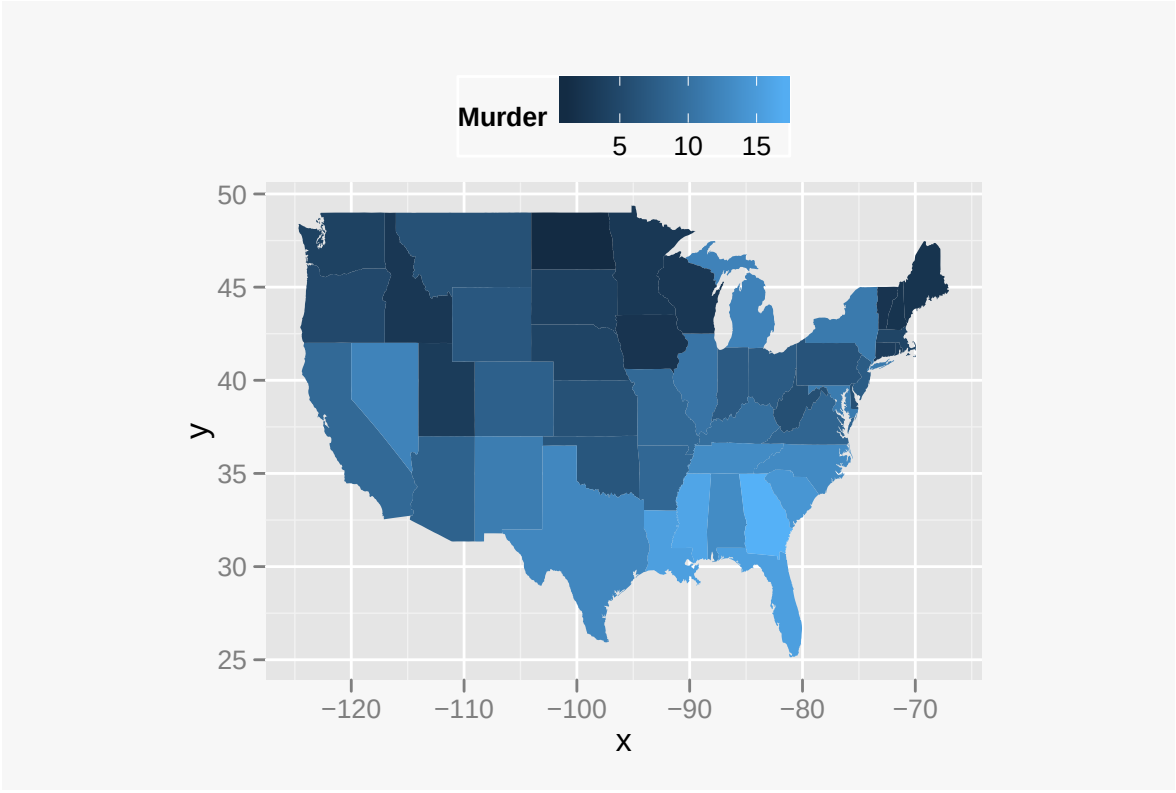
3 New geoms

This version of **ggplot2** introduces several new geoms: `geom_map()`, `geom_raster()`, `geom_dotplot()` and `geom_violin()`, the latter two of which were developed by Winston Chang.

3.1 `geom_map()`

`geom_map()` is a special case of `geom_polygon()`, specifically designed to simplify the construction of choropleth maps in **ggplot2**. The example below comes from the help page, which requires the **maps** package to produce the `states_map` data frame using the helper function `map_data()`.

```
crimes <- data.frame(state = tolower(rownames(USArrests)), USArrests)
states_map <- map_data("state")
ggplot(crimes, aes(map_id = state)) +
    geom_map(aes(fill = Murder), map = states_map) +
    expand_limits(x = states_map$long, y = states_map$lat) +
    guides(fill = guide_colorbar(colours = topo.colors(10))) +
    opts(legend.position = "top")
```



In this choropleth map, the initial `ggplot()` call invokes the data frame `crimes` with variable `state` as the ID variable to be associated with the map. In `geom_map()`, the `fill` aesthetic is the murder rate of a given state, where the map data is found in the object `states_map`.

The next example is one that imports a map file from one source and a data file from another. The following is an adaptation of an example posted by Tom Hopper on the ggplot2 list in 2010, whose solution was posted here with respect to version 0.8.9. The code takes some time to run, so I wouldn't suggest running it yourself unless you are really, really curious, but links to the data and code are given below in case you want to try it. Click here for the world map borders data and here for a sanitized file of electricity generation data. The zip file should be unpacked first in a directory and it's probably a good idea to keep the csv file in the same directory. The **rgdal** package needs to be loaded to run `readOGR()`; the first argument of the function is the directory that holds the unzipped world borders file.

```
# replace this with the directory that holds the world borders
# shapefiles and the electricity generation data
# setwd("/path/to/shapefiles")
world.map <- readOGR(layer="TM_WORLD_BORDERS-0.3")
## a good idea if you want to reuse the map data
# save(world.map, file = "worldmap.Rdata")
### Aside: to read it back in later,
## library("maptools")
## gpclibPermit()
```

```
## load("worldmap.Rdata")
gpclibPermit()
# Convert map data to data frame using fortify() for spatial objects
world.ggmap <- fortify(world.map, region = "NAME")
# Read in the global electricity generation data
# Assumes the working directory holds this data file - if not,
# modify the path
elect.gen.tot <- read.csv(
   "Total_Electricity_Net_Generation.csv", sep = ",", dec = ".")
names(elect.gen.tot) <- c("id", "y2004", "y2005", "y2006", "y2007",
"y2008")
# Plot the choropleth map
ggplot(elect.gen.tot, aes(map_id = id)) +
    geom_map(aes(fill = log(y2007)), map = world.ggmap) +
    expand_limits(x = world.ggmap$long, y = world.ggmap$lat) +
    scale_fill_gradient(low = "orange", high = "blue",
                        guide = "colorbar") +
    opts(panel.background = theme_rect(fill = "skyblue"))
```

There are some holes in the map due to missing data from a couple of countries in south central Africa and the Republic of Moldova in Eastern Europe.

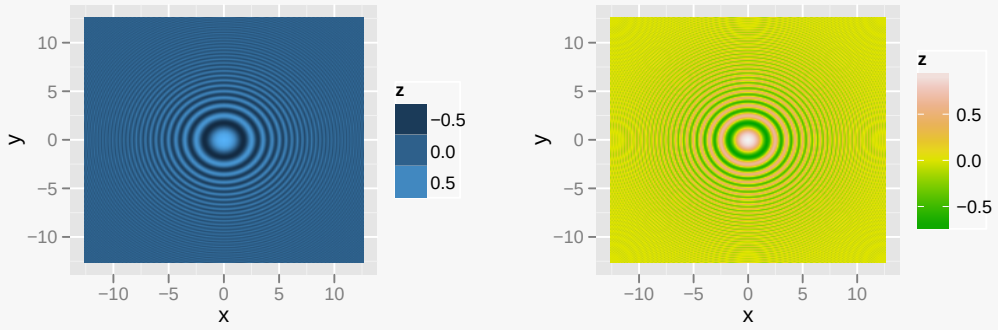
Another common source of map files is `gadb.org`, which contains administrative map files in `.Rdata` format. Not all of these maps are up to date, however, so you need to check the validity of any map downloaded. Here is some code for plotting regions of Japan from a recent question on the ggplot2 list, courtesy of Charlotte Wickham, adapted to 0.9.0 for use with `geom_map()`. Her code for 0.8.9 is shown in this message. Once again, the code is not run, but you can copy and paste it into your R session—just be willing to wait a few minutes for all of it to process, since this is a detailed administrative map.

```
load(url("http://gadm.org/data/rda/JPN_adm2.RData"))
gpclibPermit()
# convert to a structure ggplot2 can handle...takes a while
japan_map <- fortify(gadm, region = "ID_2")
# manufacture a fake data frame of values to use in geom_map()
vals <- data.frame(id = unique(japan_map$id), val = rpois(1811, 5))
ggplot(vals, aes(map_id = id)) + geom_map(aes(fill = val), map =
japan_map) +
   expand_limits(x = japan_map$long, y = japan_map$lat) +
   scale_fill_continuous(low = "navy", high = "yellow", guide =
"colorbar")
```

3.2 geom_raster()

This function is a more efficient version of geom_tile(), meant to be used when all of the tiles are the same size. Here is an example, slightly modified from its help page; the left plot uses a continuous color legend, while the one on the right uses a colorbar instead.

```
pp <- function (n,r=4) {
 x <- seq(-r*pi, r*pi, len=n)
 df <- expand.grid(x=x, y=x)
 df$r <- sqrt(df$x^2 + df$y^2)
 df$z <- cos(df$r^2)*exp(-df$r/6)
 df
}
base <- ggplot(pp(200), aes(x, y, fill = z))
base + geom_raster()
base + geom_raster() +
    scale_fill_gradientn(colours = terrain.colors(10), guide =
"colorbar")
```

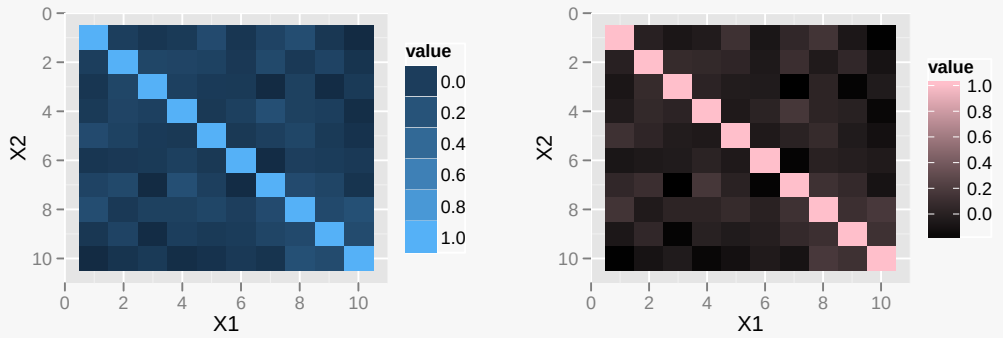


The following code uses geom_raster() to plot a correlation matrix:

```
# Create a random matrix of 10 columns (variables)
m <- matrix(rnorm(1000), nrow = 100,
            dimnames = list(NULL, paste("X", 1:10, sep = "")))
# Compute the correlation matrix
cmat <- cor(m)
# Melt it into a data frame
cmatm <- melt(cmat)
# Create two new numeric variables by stripping off the X's in the
# two factor variables Var1 and Var2:
cm1 <- mutate(cmatm, X1 = as.numeric(gsub("^X", "", cmatm$Var1)),
                     X2 = as.numeric(gsub("^X", "", cmatm$Var2)))
```

Two different ways of producing a correlation plot are shown below. As in the previous plot, the one on the left produces a legend while the one on the right uses a colorbar.

```
ggplot(cm1, aes(X1, X2, fill = value)) + geom_raster() +
scale_y_reverse()
last_plot() + scale_fill_continuous(low = "black", high = "pink", guide
= "colorbar")
```



3.3 geom_dotplot()

This function is an implementation of Wilkinson dot plots (Wilkinson, 1999), a generalization of the basic dot plot used to represent the distribution of a continuous variate when the sample size is small. The help page of this function has an extensive set of examples covering all of its basic features, a selected sample of which is included here for illustration.

Two basic algorithms exist for producing a dot plot: *dot density*, which uses a kernel density estimation algorithm to position dots (described in the Wilkinson paper), and *histodot*, a histogram whose bars are drawn as stacks of dots with the same bin width. A 'histodot' plot is distinguished by regular horizontal spacing between stacks (i.e., fixed positions and fixed widths), whereas in a 'dot-density' plot, the bin positions are determined by the data and the selected binwidth, which in this case represents the *maximum* binwidth. Most of the time a dot density plot is preferred, which is why it is the default method used by the geom.

A dot plot can be viewed as a 1D horizontal scatterplot in which (nearly) tied values are perturbed or displaced vertically. There are three basic ways in which points in a graphic can be displaced: (i) jitter; (ii) textured dot strips (Tukey and Tukey, 1990); and (iii) dot plots, where the points are displaced in increments of one dot width. The advantage of these approaches over the 'histodot' method is that outliers are positioned where they should be rather than assigned to a midpoint value defined by fixed binning (Wilkinson, 1999).

Dots can be manipulated in certain ways:

(i) The size of a dot can vary depending on the sample size and number of bins, or it can be set in a `geom_dotplot()` call with the `dotsize =` argument.

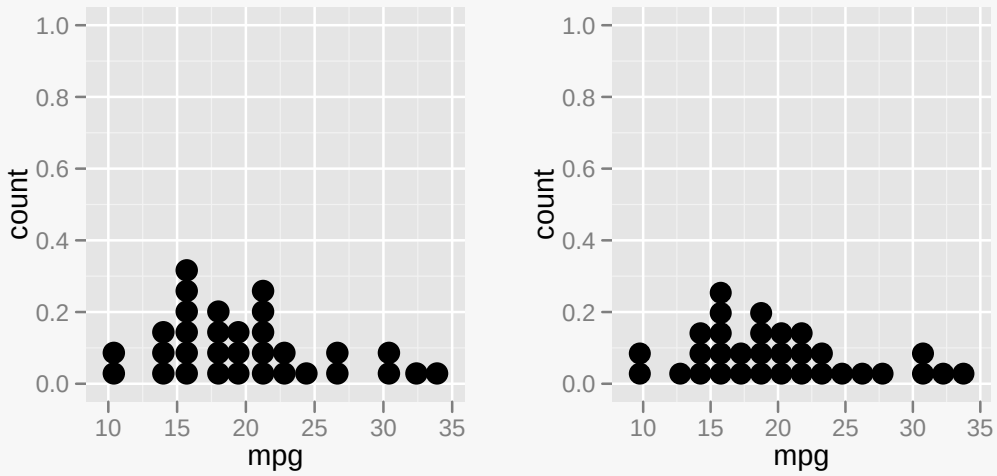
(ii) Dots can be stacked in various ways using the `stackdir =` argument.

(iii) The `stackratio` argument specifies how closely to stack the dots within a stack; values smaller than the default 1 move stacks closer together and create overlapping points, whereas values larger than 1 place dots further apart from one another.

(iv) The `binaxis =` argument specifies the axis along which to bin (`"x"` or `"y"`).

The following example shows the subtle difference between the 'dotdensity' and 'histodot' methods when the binwidth is specified:

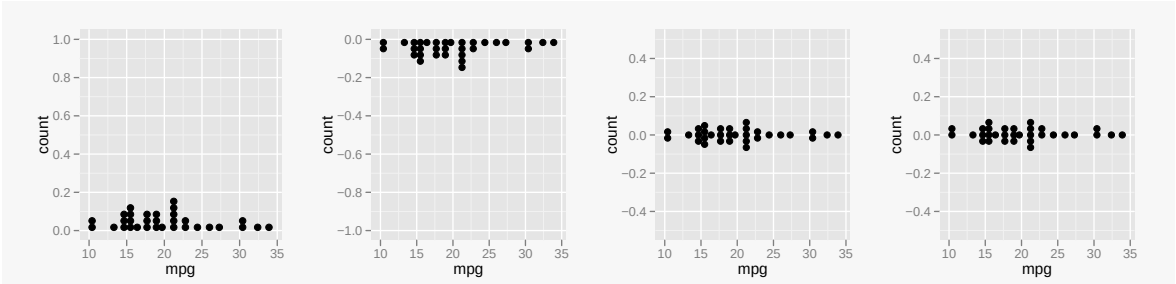
```
# default dotdensity method (left)
ggplot(mtcars, aes(x = mpg)) + geom_dotplot(binwidth = 1.5)
# histodot method (right)
ggplot(mtcars, aes(x = mpg)) +
  geom_dotplot(method="histodot", binwidth = 1.5)
```



The stacks are slightly different in the two plots, but the stacks are closer together in the histodot method because the dot size is set to the bin width. The dot size appears to be slightly larger in the left plot (which uses the 'dotdensity' algorithm).

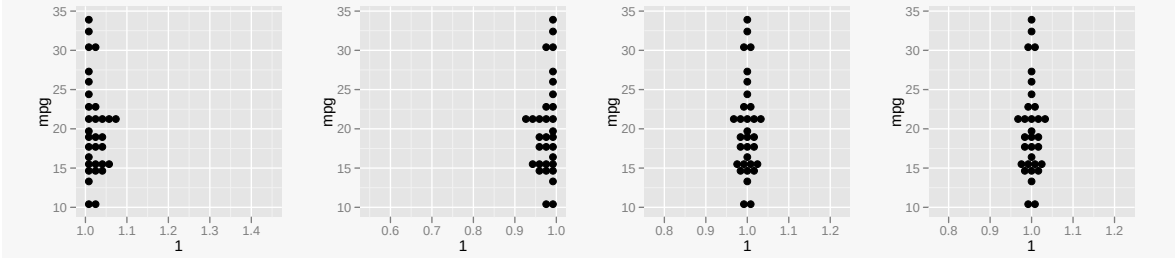
The next example shows several ways to stack dots in a 1D dot plot in the horizontal direction using the default binwidth (range/30):

```
# default stack method (up)
ggplot(mtcars, aes(x = mpg)) + geom_dotplot()
# alternate stack method (down)
ggplot(mtcars, aes(x = mpg)) +
  geom_dotplot(stackdir = "down")
# alternate stack method (center)
ggplot(mtcars, aes(x = mpg)) +
  geom_dotplot(stackdir = "center")
# alternate stack method (centerwhole)
ggplot(mtcars, aes(x = mpg)) +
  geom_dotplot(stackdir = "centerwhole")
```



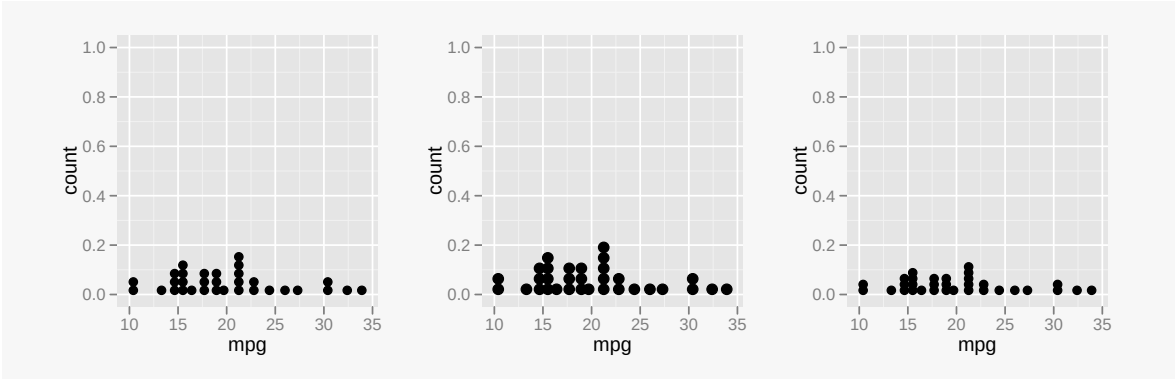
Similarly, one can stack dots in the vertical direction. To do this for a single dot plot, one needs to set `x = 1` and make the variable the `y` in the `aes()` argument. Observe that these are essentially reflections of the preceding set of plots rotated 90°.

```
# default stack method (up)
ggplot(mtcars, aes(x = 1, y = mpg)) + geom_dotplot(binaxis = "y")
# alternate stack method (down)
ggplot(mtcars, aes(x = 1, y = mpg)) +
  geom_dotplot(binaxis = "y", stackdir = "down")
# alternate stack method (center)
ggplot(mtcars, aes(x = 1, y = mpg)) +
  geom_dotplot(binaxis = "y", stackdir = "center")
# alternate stack method (centerwhole)
ggplot(mtcars, aes(x = 1, y = mpg)) +
  geom_dotplot(binaxis = "y", stackdir = "center")
```



Here are a few examples using the `dotsize` and `stackratio` arguments:

```
# base plot
ggplot(mtcars, aes(x = mpg)) + geom_dotplot()
# increase the dot size
ggplot(mtcars, aes(x = mpg)) + geom_dotplot(dotsize = 1.25)
# overlap dots within stacks:
ggplot(mtcars, aes(x = mpg)) + geom_dotplot(stackratio = 0.7)
```

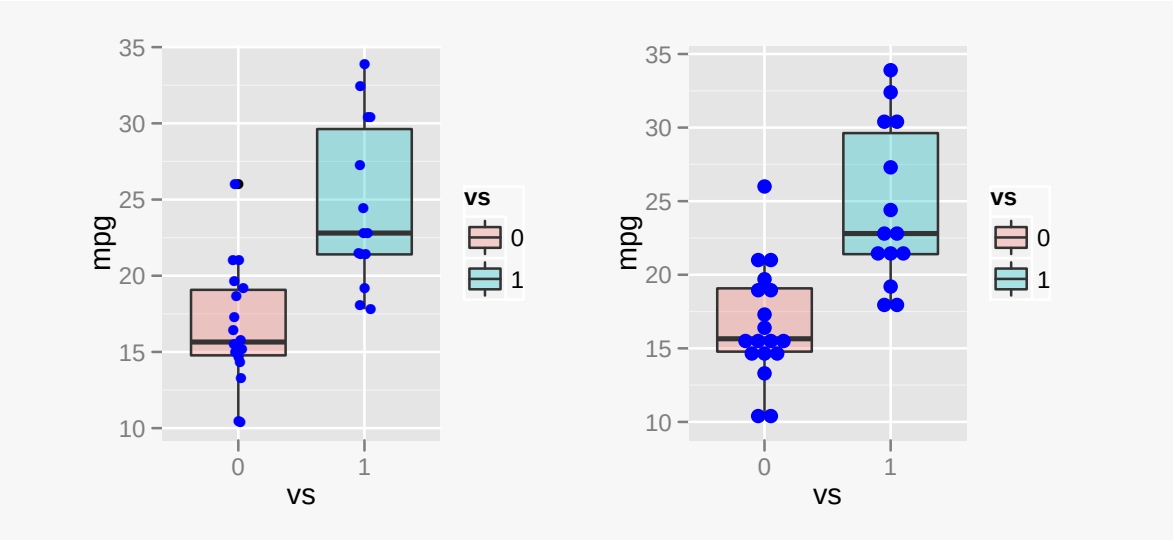


As you may have noticed, there is a problem when rendering a 1D dot plot: the dot axis scaling is essentially meaningless since it is difficult to scale the y-axis relative to dot size. It is also why 1D dot plots have a lot of empty space in most applications. One option is to remove the vertical axis ticks altogether by adding the following code snippet to a `ggplot()` call:

```
scale_y_continuous(name = "", breaks = NA)
## or if the dot plot is vertical,
scale_x_continuous(name = "", breaks = NA)
```

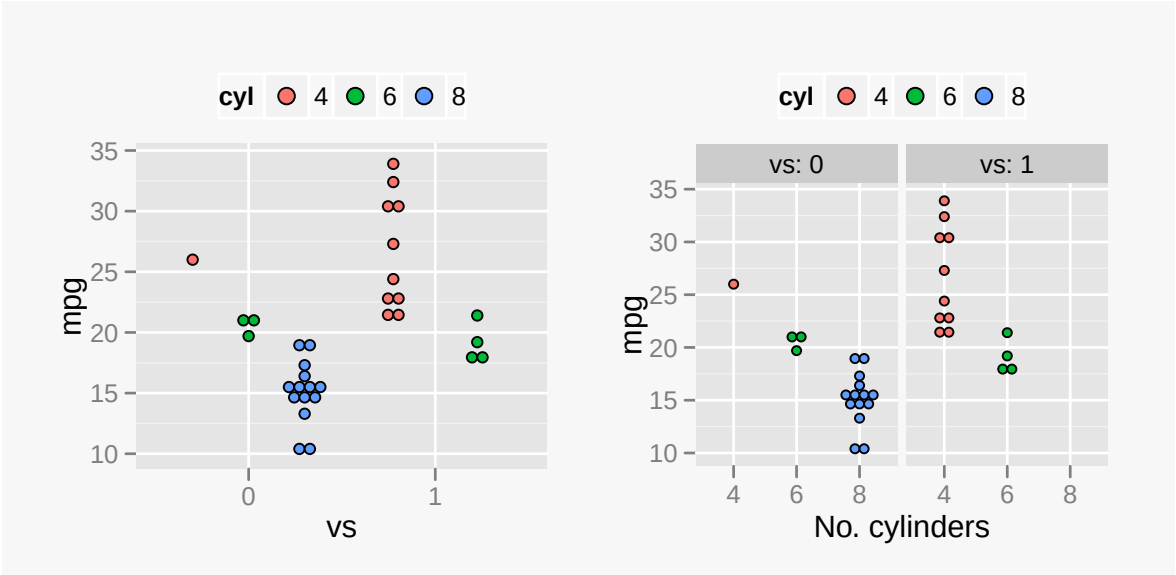
Much like a box plot, a dot plot is a more interesting graphical device when used to compare the distributions of several related groups. In this context, dot plots are a better organized method of handling overlapping points than jittering. The following pair of plots illustrate this idea.

```
ggplot(mtcars, aes(x = factor(vs), y = mpg)) +
   geom_boxplot(aes(fill = factor(vs)), alpha = 0.3, outlier.color =
NA) +
   geom_point(position = position_jitter(width = 0.05),
              colour = "blue", fill = "blue") +
   labs(x = "vs", y = "mpg", fill = "vs")
ggplot(mtcars, aes(x = factor(vs), y = mpg)) +
   geom_boxplot(aes(fill = factor(vs)), alpha = 0.3, outlier.color =
NA) +
   geom_dotplot(binaxis = "y", stackdir = "center",
                position = "dodge", colour = "blue", fill = "blue") +
   labs(x = "vs", y = "mpg", fill = "vs")
```



Whenever a factor is used as an aesthetic in a dot plot, the rule is that the dot plot for each level will be dodged; this is also the case when multiple factors are at play. For example, if *A* and *B* are two factors such that the levels of *A* map to the horizontal axis, then dotplots of each level of *B* are dodged within individual levels of *A*. The following plot shows two different ways of handling this situation. The left plot dodges the dot plots by number of cylinders within each level of `vs`, whereas the right plot uses `facet_grid()` as an alternative mode of display. In this particular example, the faceted plot makes it easier to see that there are no cars with eight cylinders when `vs = 1`. Note that in the faceted plot, the labeller function `label_both()` is used to indicate both the factor name and its level in each panel strip.

```
ggplot(mtcars, aes(x = factor(vs), fill = factor(cyl), y = mpg)) +
  geom_dotplot(binaxis = "y", stackdir = "center",
               position = "dodge") +
  labs(x = "vs", y = "mpg", fill = "cyl") +
  opts(legend.position = "top")
ggplot(mtcars, aes(x = factor(cyl), fill = factor(cyl), y = mpg)) +
  geom_dotplot(binaxis = "y", stackdir = "center") +
  labs(x = "No. cylinders", y = "mpg", fill = "cyl") +
  facet_grid(. ~ vs, labeller = label_both) +
  opts(legend.position = "top")
```

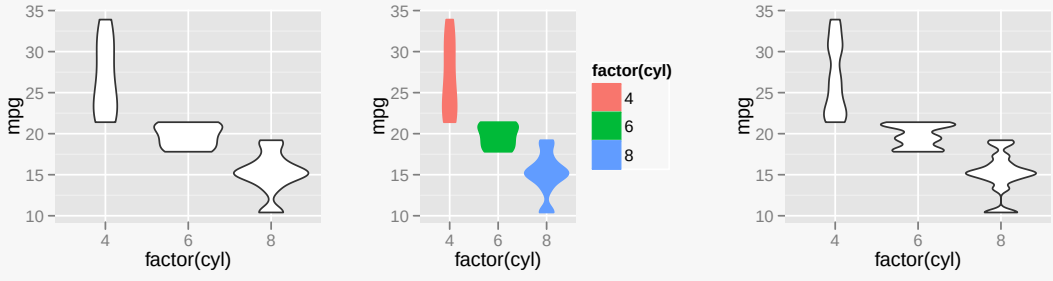


3.4 geom_violin()

This function generates violin plots in **ggplot2**, a way to plot one or more continuous density estimates that is particularly useful when comparing multiple groups. A violin plot is a combination of a box plot and a kernel density estimate, the latter of which is rotated to run alongside the box plot symmetrically on each side. The examples below come from the function's help page.

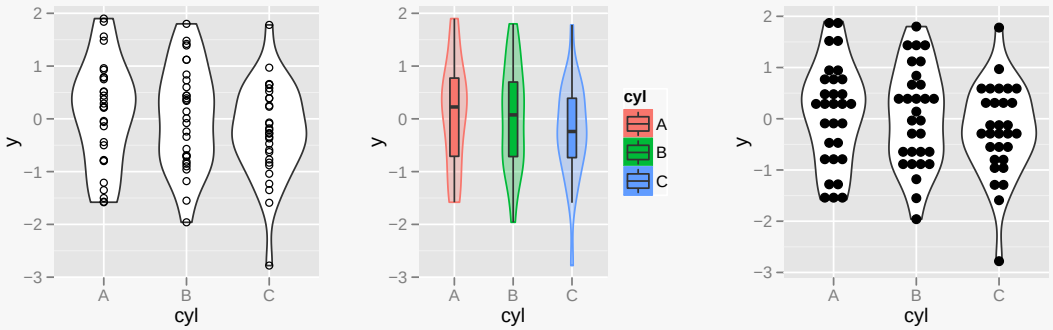
In `geom_violin()`, violins are automatically dodged when any aesthetic is a factor. By default, the maximum width is scaled to be proportional to the sample size. In the plot on the far right below, the bandwidth of the kernel density estimator is reduced from the default 1, which makes for a less smooth density estimate and hence a less smooth violin plot.

```
p <- ggplot(mtcars, aes(factor(cyl), mpg))
p + geom_violin()    # default scale is "count"
p + geom_violin(aes(fill = factor(cyl), colour = factor(cyl)))
p + geom_violin(adjust = 0.5)
```



The next set of plots simply play around with a few extra features. The plot on the left adds a strip plot to the violin for each group. The central plot adds fill color and alpha transparency to the violins and is augmented with boxplots. The plot on the far right adds a dot plot around the center of each violin, which is useful in that the dot plot resembles the shape of the violin (as it should since each estimates the underlying true density in its own way).

```
set.seed(110)
dat <- data.frame(x=LETTERS[1:3], y=round(rnorm(90), 2))
ggplot(dat, aes(x=x, y=y)) + geom_violin() + geom_point(shape=21) +
    xlab("cyl")
ggplot(dat, aes(x = x, y = y)) +
    geom_violin(aes(fill = x, colour = x), alpha = 0.3, width = 0.5) +
    geom_boxplot(aes(fill = x), width = 0.2, outlier.colour = NA) +
    labs(x = "cyl", fill = "cyl", colour = "cyl") +
    guides(fill = guide_legend(override.aes = list(alpha = 1)))
ggplot(dat, aes(x = x, y = y)) +
    geom_violin() +
    geom_dotplot(binaxis = "y", stackdir = "center") +
    xlab("cyl")
```



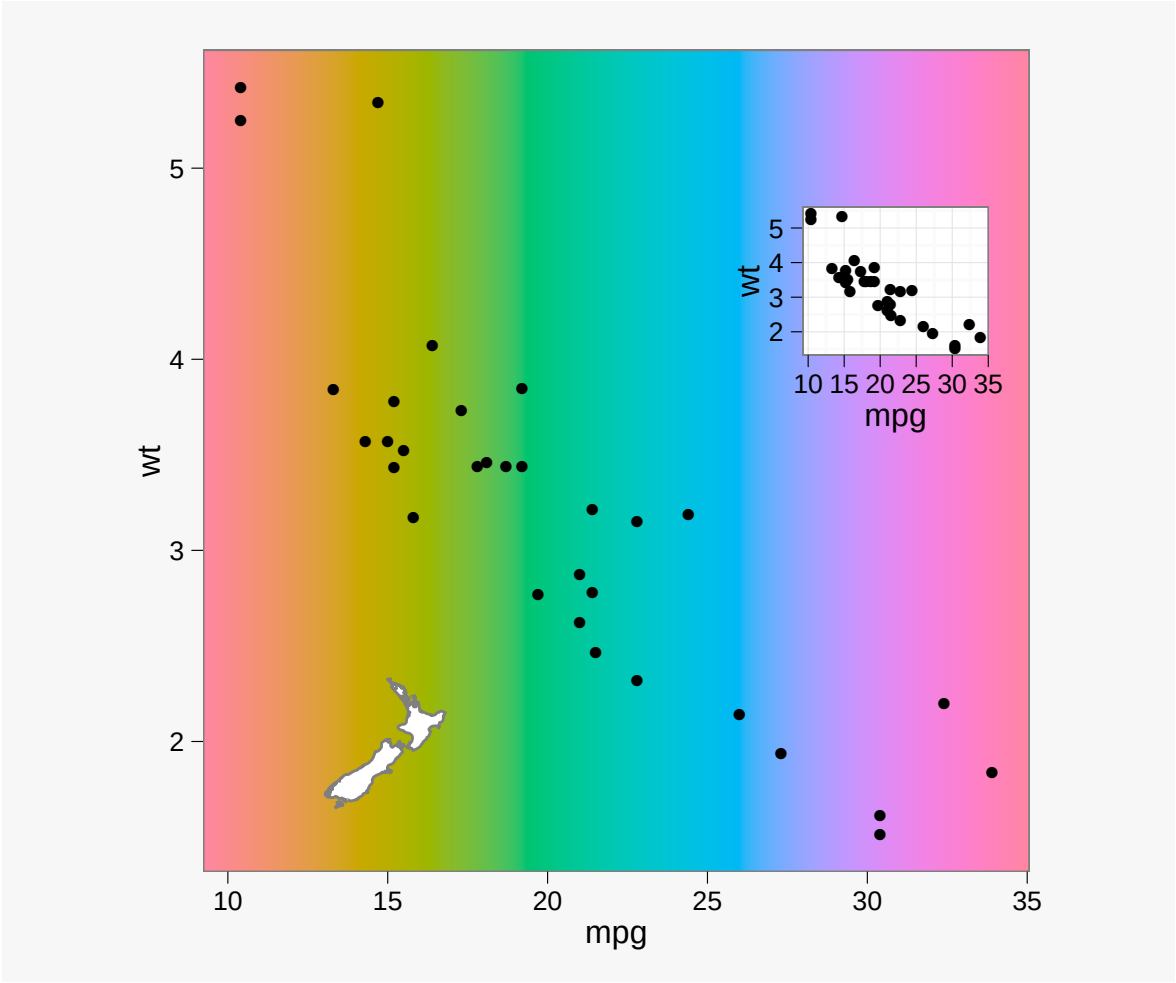
4 New annotation functions

Version 0.9.0 debuts three new functions for annotation of graphics: `annotation_raster()`, `annotation_map()` and `annotation_custom()`. These can be used to decorate a `ggplot`:

- `annotation_raster()` allows one to insert a (raster) image into a plot. It can serve as a plot background (see the example below) or be a graphical object positioned somewhere in the graphics area of a `ggplot`.
- `annotation_map()` allows one to insert a map inside of a graphics region, e.g., as an inset.
- `annotation_custom()` allows one to insert graphs, tables or other graphical objects of value in a graphic.

The first example is a modification of the example in `annotation_raster()` to combine all three new annotation functions in one graphic.

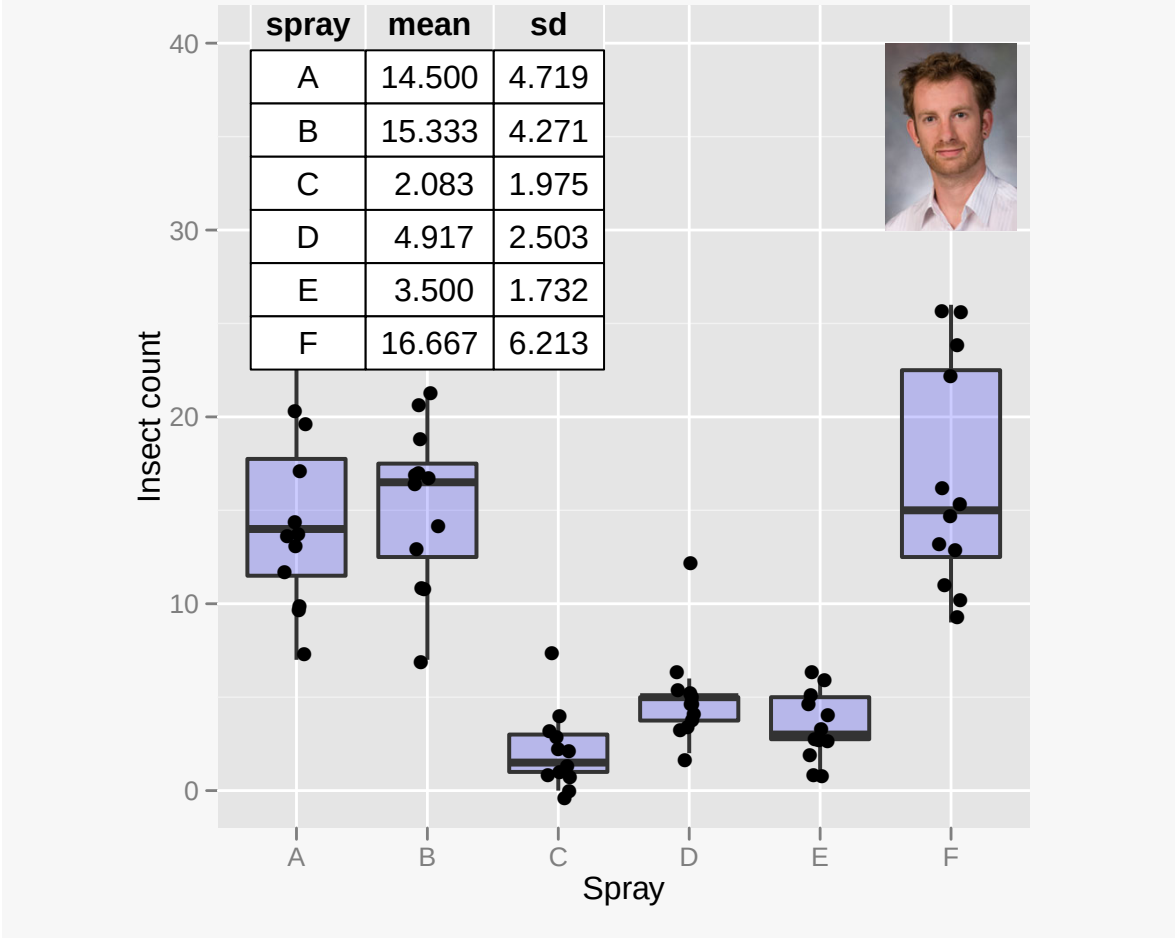
```
p <- qplot(mpg, wt, data = mtcars) + theme_bw()
## color gradient
rainbow <- t(hcl(seq(0, 360, length = 300 * 50), 80, 70))
## scaled down map of NZ
nz <- transform(map_data("nz"), long = scale(long) + 15,
                lat = scale(lat)/5 + 2)
# Inset plot
g <- ggplotGrob(p)
# Put it together:
p + annotation_raster(rainbow, -Inf, Inf, -Inf, Inf) +
    geom_point() +
    annotation_map(nz, fill = "NA", colour = "grey50") +
    annotation_custom(grob = g, xmin = 25, xmax = 35,
                      ymin = 3.5, ymax = 5)
```



The next example inserts a table and a picture in the graphic using the `InsectSprays` data. The table is produced from the `tableGrob()` function in the **gridExtra** package.

```
ISsumm <- ddply(InsectSprays, .(spray), summarise,
                mean = mean(count), sd = sd(count))
tb <- tableGrob(ISsumm, rows = ISsumm$spray, show.rownames = FALSE,
                gpar.corefill = gpar(fill = 'white'))
hadley <- read.jpeg('hadley.jpg')
ggplot(ISsumm, aes(x = spray, y = mean)) +
    geom_boxplot(data = InsectSprays, aes(y = count),
                 fill = 'blue', alpha = 0.2,
                 size = 0.7, outlier.colour = NA) +
    geom_point(data = InsectSprays, aes(y = count), size = 2.5,
               position = position_jitter(width = 0.1)) +
    ylim(0, 40) + labs(x = 'Spray', y = 'Insect count') +
    annotation_custom(grob = tb, xmin = 0.5, xmax = 3.5,
                                 ymin = 25, ymax = 40) +
```

```
    annotation_raster(hadley, xmin = 5.5, xmax = 6.5,
                              ymin = 30, ymax = 40)
```

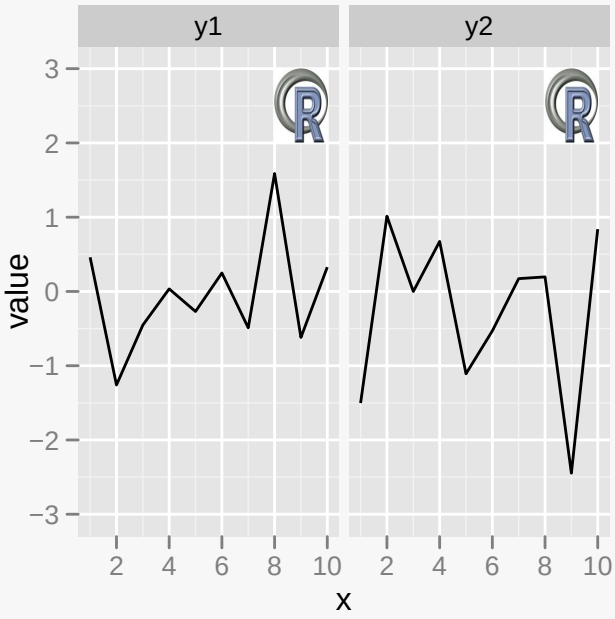


Note that the `read.jpeg()` function comes from the **ReadImages** package, designed to read JPEG and PNG files.

If you resize a `ggplot` graphic in a document with decorations such as those shown in the previous two examples, there is no guarantee that the decorations will scale in the same ratio as the rest of the graphic. Maps and rasters will scale proportionally with the plot, but custom grobs may not scale at all, such as the `tableGrob` produced in the graphic above. When adding annotations to a graphic that will end up in a document, it may be a good idea to create a graphics window in R of the same size as what you want in the document. By developing the graphic at that size, you should have more confidence that what you see in R is what you'll see in the document.

Another aspect of annotations in **ggplot2** is that they are displayed in each facet because they are insensitive to mapping. Here's a simple example:

```
DF <- data.frame(x = 1:10, y1 = rnorm(10), y2 = rnorm(10))
# Read in the R logo:
Rlogo <- read.jpeg('Rlogo.jpg')
# Melt and plot:
dfm <- melt(DF, id = 'x')
ggplot(dfm, aes(x = x, y = value)) + geom_path() +
    ylim(-3, 3) +  annotation_raster(Rlogo, xmin= 8,
         xmax = 10, ymin = 2, ymax = 3) +
    facet_wrap(~ variable)
```

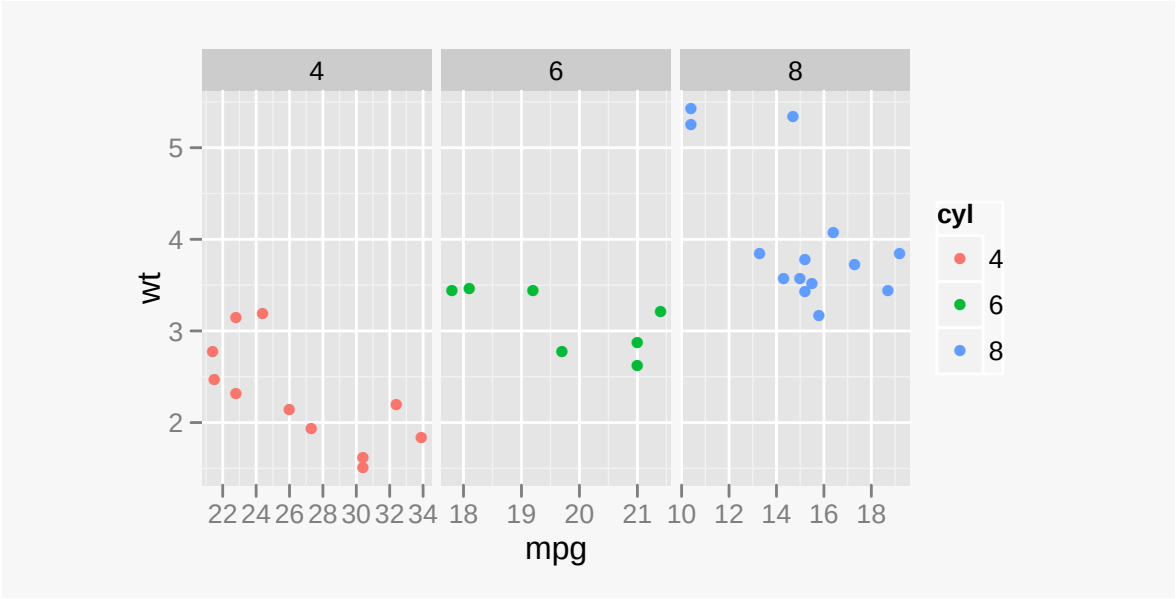


5 Improvements in `facet_grid()`

Several improvements have been made to `facet_grid()`, as noted in the NEWS file for this version of the package and in section 1.1. Below are some examples taken from the function's help page.

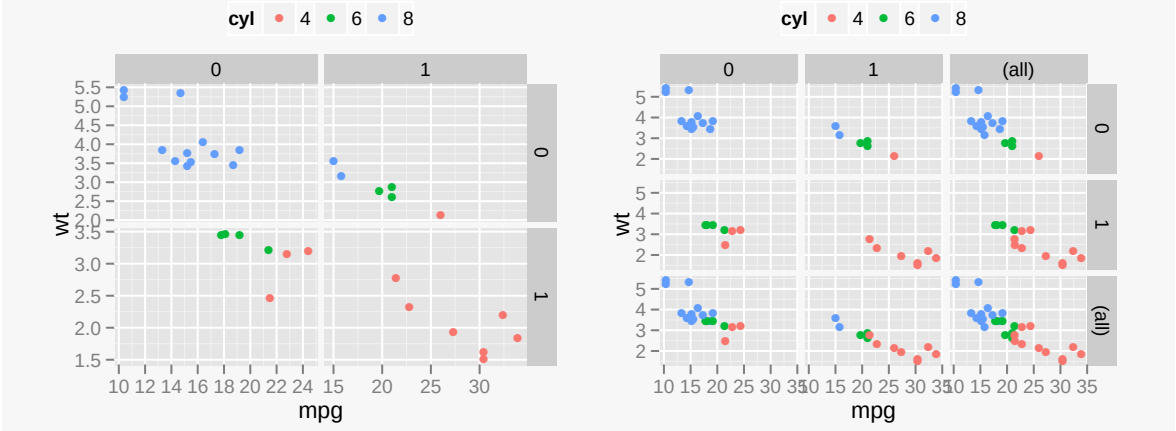
The first plot illustrates a one-dimensional grid with free scales—the x scales vary and the panel widths are the same:

```
m <- ggplot(mtcars, aes(mpg, wt, colour = factor(cyl))) + geom_point()
m + facet_grid(. ~ cyl, scales = "free") + labs(colour = "cyl")
```



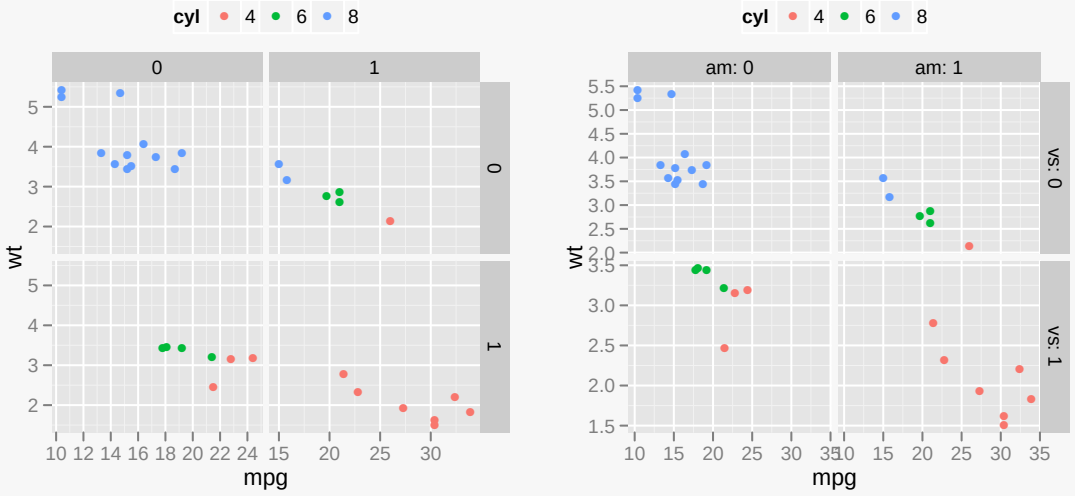
Next, we show a set of plots with two-dimensional grids. The first is a basic 2D grid with free scales in both dimensions—in this case, each panel has separate x and y scales. The plot on the right shows that `margins = TRUE` is once again a viable option in `facet_grid()`.

```
m + facet_grid(vs ~ am, scales = "free") + labs(colour = "cyl") +
    opts(legend.position = "top")
m + facet_grid(vs ~ am, margins = TRUE) + labs(colour = "cyl") +
    opts(legend.position = "top")
```



One of the new features, due to the efforts of Willem Hilgenberg, is to allow free scales in one direction only. The left plot below produces different x scales in the columns while the plot on the right produces different y scales:

```
m + facet_grid(vs ~ am, scales = "free_x") + labs(colour = "cyl") +
    opts(legend.position = "top")
# Add labeller to distinguish levels of vs
# from those of am:
m + facet_grid(vs ~ am, scales = "free_y", labeller = label_both) +
    labs(colour = "cyl") +
    opts(legend.position = "top")
```

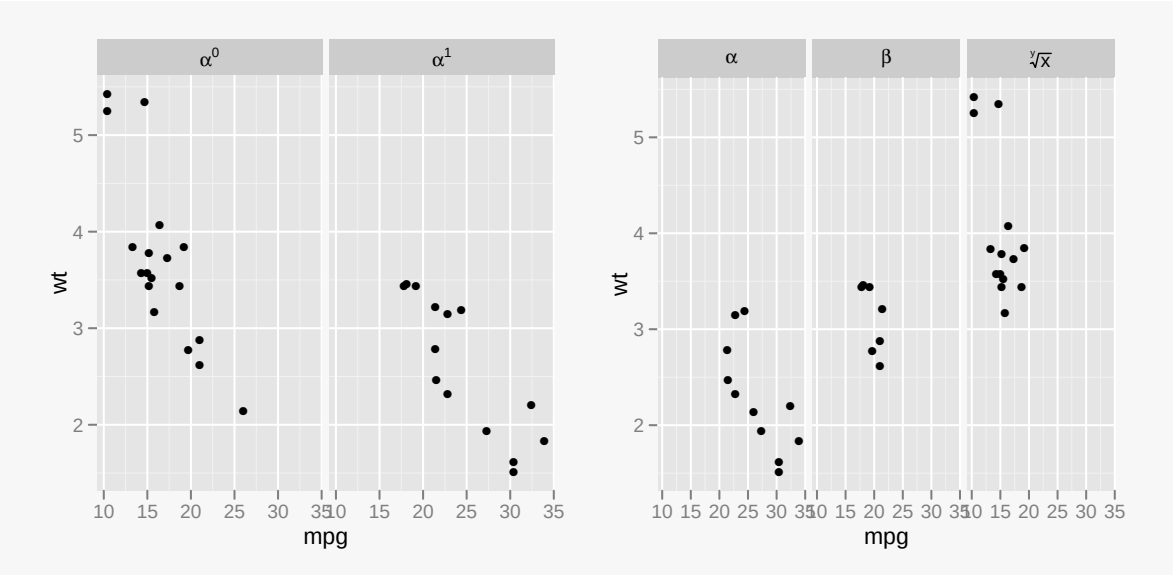


The following plots, which are not shown here, combine free scales with free space:

```
m + facet_grid(vs ~ am, scales = "free", space = "free")
m + facet_grid(vs ~ am, scales = "free", space = "free_x")
m + facet_grid(vs ~ am, scales = "free", space = "free_y")
```

The way strip labels are rendered in `facet_grid()` has changed in version 0.9.0. One now uses the `label_*()` functions from the **scales** package in conjunction with the `labeller =` argument of `facet_grid()`. A couple of examples using the `label_both()` were shown above; the default labeller is `label_value()`. The following example from the `facet_grid()` help page in 0.9.0 demonstrates the other built-in labeling functions:

```
mtcars$cyl2 <- factor(mtcars$cyl, labels = c("alpha", "beta", "sqrt(x,
y)"))
q <- qplot(mpg, wt, data = mtcars) + facet_grid(cyl ~ vs)
# Recall that the levels of vs are 0 and 1:
q + facet_grid(. ~ vs, labeller = label_bquote(alpha^.(x)))
q + facet_grid(. ~ cyl2, labeller = label_parsed)
```



The labeller function `label_bquote()` takes a 'bquoted' expression as its input argument. In previous versions of **ggplot2**, one would define a function and then pass it to the `labeller =` argument. With `label_parsed()`, one passes a vector of strings to `facet_grid()` that are parsed into expressions and evaluated.

6 **scales** package

The first phase of reorganizing the **ggplot2** package was to group functions that pertain to the construction of guides—i.e., axes (or positional guides) and legends (aesthetic guides)—into a single package. These functions are derived from **ggplot2-0.8.9** but have been rewritten so that they can be used in base graphics or **lattice**[3].

The functions in the **scales** package fall broadly into the following categories:

***_breaks():** functions to manipulate axis breaks;

***_format():** functions that are used to modify axis labels, replacing the `formatter =` argument of a positional scale function in previous versions of **ggplot2**;

label_*(): functions to modify panel strips in faceted ggplots;

***_trans():** a collection of predefined scale transformation functions and their inverses, to be used with the new `trans =` argument to positional scale functions;

***_pal():** a set of functions that correspond to attributes associated with legends.

The `scale_*()` functions have been rewritten so that both the `breaks` and `labels` arguments accept a function as its 'value'. A new argument `trans` allows one to specify a

[3]Tangible evidence of this feature is shown in the examples of the `cscale()` and `dscale()` functions in **scales**.

scale transformation in `scale_continuous()` or `scale_discrete()`; this new argument streamlines the number of continuous axis scaling functions in 0.9.0—the only ones that remain are `scale_continuous()`, `scale_discrete()`, `scale_reverse()`, `scale_sqrt()` and `scale_log10()`.

An important characteristic of the functions in the **scales** package is that they are **second-order R functions**. This means a given function, e.g., `comma_format()`, returns a *function*, so it needs two sets of arguments to be evaluated. To illustrate, `comma_format()` has an argument `digits` which can be used to avoid potential ugliness in axis breaks:

```
# Call with default digits argument
comma_format()(seq(0, 10000, by = 2000))

[1] "0"     "2,000" "4,000" "6,000" "8,000" "1e+04"

# Set digits = number of digits in the maximum value among the axis
breaks
comma_format(digits = 5)(seq(0, 10000, by = 2000))

[1] "0"      "2,000"  "4,000"  "6,000"  "8,000"  "10,000"

# Don't forget there are two sets of arguments:
comma_format(seq(0, 10000, by = 1000))

function (x)
comma(x, ...)
<environment: 0x000000000c383488>
```

The first set of arguments in a second-order R function correspond to those you want to change in the original function; the second set of arguments call the function returned by the original function.

The purpose of converting to functions and repackaging them is to provide a user with more tools to manage axis ticks, axis labels and certain elements of legends. Except for the `*_pal()` functions, these will normally be used inside a `scale_*()` function call; several examples are given in section 7.

6.1 Functions to manage breaks, axis labels and strip labels

The following table summarizes the set of functions associated with axis breaks, axis labels and strip labels (in faceted ggplots):

Breaks	**Formats**	**Labels**
`date_breaks()`	`comma_format()`	`label_both()`
`log_breaks()`	`dollar_format()`	`label_bquote()`
`pretty_breaks()`	`percent_format()`	`label_parsed()`
`trans_breaks()`	`scientific_format()`	`label_value()`
	`date_format()`	
	`parse_format()`	
	`math_format()`	
	`format_format()`	
	`trans_format()`	

Just as one could write a formatter function in previous versions of **ggplot2**, one can define a function inside `math_format()` to pass as the `labels =` argument of the appropriate scale function.

6.2 Transformation functions

The table below summarizes the available transformation functions in the **scales** package to be used in conjunction with the `coord_trans()` function or the `trans =` argument in a positional scale function.

`asn_trans()`	`identity_trans()`	`probit_trans()`
`atanh_trans()`	`log1p_trans()`	`reciprocal_trans()`
`boxcox_trans()`	`log_trans()`	`reverse_trans()`
`date_trans()`	`logit_trans()`	`sqrt_trans()`
`exp_trans()`	`probability_trans()`	`time_trans()`

Nearly all of these functions derive from **ggplot2**, rewritten to have a consistent format so that it will be easier for users to define their own transformation functions. The simplest pattern for a transformation function is

```
my_trans <- function()
   trans_new("my", function(x) ..., function(x) ...)
```

where the first argument is the name of the transformation, the second argument is the function corresponding to the transformation and the third argument is its inverse function. The ellipses are placeholders for the function definitions. Two examples are illustrated below: the arcsine transformation and the slightly more complicated logarithmic transformation:

```
asn_trans

function ()
{
    trans_new("asn", function(x) 2 * asin(sqrt(x)), function(x) sin(x/2)^2)
}
<environment: namespace:scales>

log_trans

function (base = exp(1))
{
    trans <- function(x) log(x, base)
    inv <- function(x) base^x
    trans_new(str_c("log-", format(base)), trans, inv, log_breaks(base = base),
        domain = c(1e-100, Inf))
}
<environment: namespace:scales>
```

6.3 Palette functions

This set of functions in the **scales** package is primarily for use with scale functions corresponding to plot attributes such as color, hue, linetype or character shape. The palette functions are summarized in the table below:

Color and fill	**Other aesthetics**
`brewer_pal()`	`area_pal()`
`dichromat_pal()`	`identity_pal()`
`div_gradient_pal()`	`linetype_pal()`
`grey_pal()`	`rescale_pal()`
`hue_pal()`	`shape_pal()`
`seq_gradient_pal()`	

Most of the time there is no need to use these functions with **ggplot2** graphics since the package already has built-in functions that incorporate these palettes. More often, one would use them in conjunction with the `cscale()` and `dscale()` functions in the **scales** package using base or Lattice graphics.

7 Scaling x- and y-axes

Axes are positional guides that contain a *scale*, a *label*, a *rule* and usually, a *title*. The fundamental difference between axes and legends is that axes are transformable whereas legends are not (Wilkinson, 2005). As a result, the `scale_continuous()` and `scale_discrete()`

functions in **ggplot2** now take a `trans =` argument, but scale functions for legends (e.g.,, `scale_colour_*()` or `scale_linetype()`) do not. Conversely, scale functions for legends have a `guides =` argument (see section 2) that positional axis functions do not.

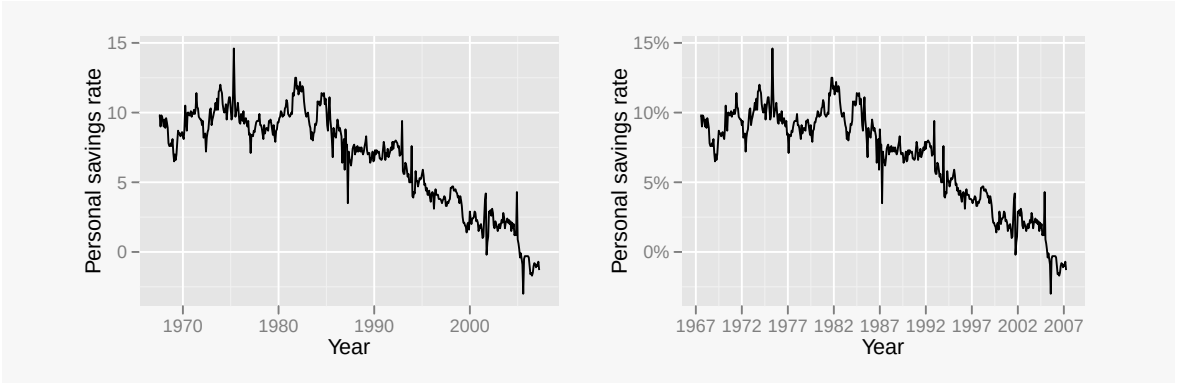
The `trans =` argument in a discrete or continuous scale function corresponds to the scale transformation to be applied to the variable of interest prior to any statistical transformation. The examples in this section coordinate the new features of the `scale_*()` functions with the functions in the **scales** package described in section 6.

The `*_breaks()` set of functions controls the values that define the rule, which may be formatted by one of the `*_format()` functions introduced in section 6.1 to produce the labels. In fact, you can think of the `labels` argument in 0.9.0 `scale_*()` functions as a merger of the `labels` and `formatter` arguments from previous versions. For dates and/or times, one can use the `scale_date()` or `scale_datetime()` functions, which also allow the `breaks =` and `labels =` arguments to take functions as 'values'. The most common applications for `_breaks` functions are (i) specification of date intervals and (ii) breaks for a log scale of arbitrary base. This section is divided into two parts: (a) date and datetime scales and (b) continuous scales.

7.1 Dates and datetimes

Some **ggplot2** users have found date formatting to be rather vexing in the past, so now there are two functions in the **scales** package that should make it easier to specify date formats. The `date_breaks()` function defines the time interval between adjacent breaks while the `date_format()` function specifies how the dates should be displayed on the time axis. The syntax for individual date elements such as years, days and months is described in `?strptime`. The following example comes from the `economics` data frame in **ggplot2** where we use `date_breaks()` and `date_format()` to tailor the date axis. The plot on the left is the default plot whereas the one on the right modifies the date breaks and labels as well as expressing the response as a percentage:

```
(qp <- qplot(date, psavert, data = economics, geom = "line") +
    labs(x = "Year", y = "Personal savings rate"))
# Divide savings rate by 100 to get the correct percentages
qplot(date, psavert/100, data = economics, geom = "line") +
    labs(x = "Year", y = "Personal savings rate") +
    scale_x_date(breaks = date_breaks("5 years"),
                 labels = date_format("%Y")) +
    scale_y_continuous(labels = percent)
```



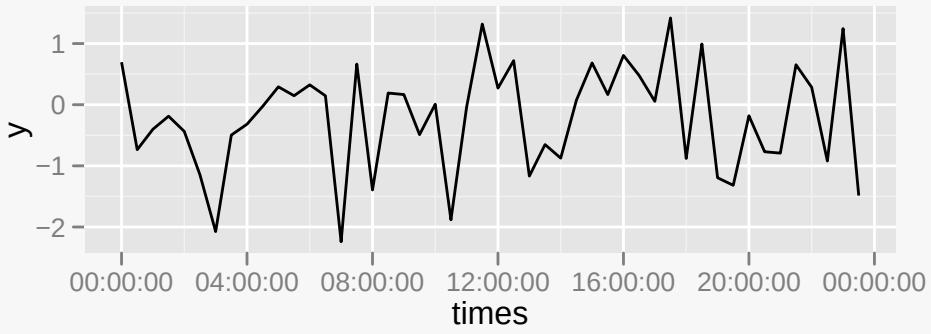
Notice that since `percent_format()` has no arguments, we can simply use the kernel of the function name as the argument of `labels =` in `scale_y_continuous()`.

As a side note, the equivalent code in version 0.8.9 for the plot on the right would be

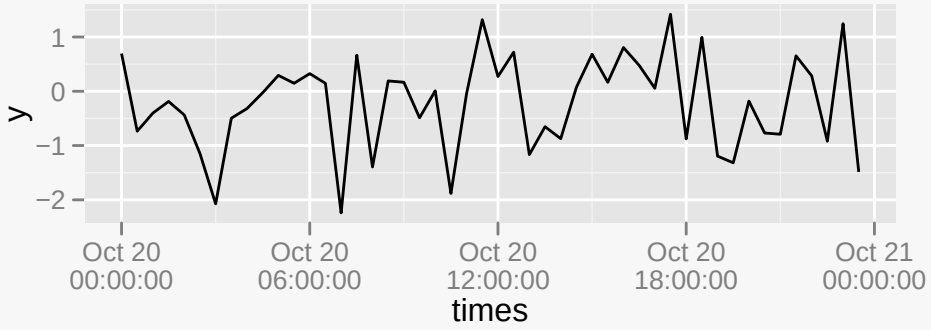
```
qplot(date, psavert/100, data = economics, geom = "line") +
     labs(x = "Year", y = "Personal savings rate") +
     scale_x_date(major = "5 years") +
     scale_y_continuous(formatter = "percent")
```

Next, consider an example with datetime data. A small fake data frame is constructed to produce a couple of plots using datetime axes.

```
df2 <- data.frame(times = seq(as.POSIXct("2011-10-20 00:00:00"),
                              as.POSIXct("2011-10-20 23:59:59"),
                               by = "30 min"), y = rnorm(48))
# 0.8.9 code:
# ggplot(df2, aes(x = times, y = y)) + geom_path() +
#     scale_x_datetime(major = "4 hours", format = "%H:%M:%S")
# 0.9.0 code:
# Labels with times only:
ggplot(df2, aes(x = times, y = y)) + geom_path() +
      scale_x_datetime(breaks = date_breaks("4 hours"),
                       labels = date_format("%H:%M:%S"))
```




```
# Labels with both day and time:
last_plot() + scale_x_datetime(breaks = date_breaks("6 hours"),
                               labels = date_format("%b %d\n%H:%M:%S"))
```



From these examples, one can see that the `date_breaks()` function replaces the deprecated argument `major =` in `scale_date_x()` and that `date_format()` replaces the old argument `format =` .

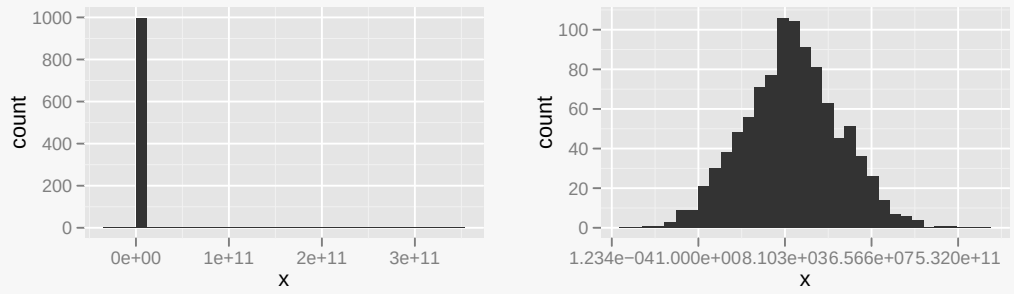
7.2 Continuous variables

Several important changes have been made to the arguments of `scale_continuous()` which will be illustrated in this subsection. Consider the problem of finding a histogram for a lognormal pseudo-random sample, where the histogram is the 'statistical transformation' of interest.

The left graph below is a histogram in the original scale of measurement while its counterpart on the right is computed from the natural logarithm of the original measurements.

```
DF <- data.frame(x = rlnorm(1000, m = 10, sd = 5))
# on original scale, produces a spiked histogram
(p <- ggplot(DF, aes(x = x)) + geom_histogram())
```

```
# Apply a scale transformation to x to get the histogram of ln(x):
p + scale_x_continuous(trans = "log")
```

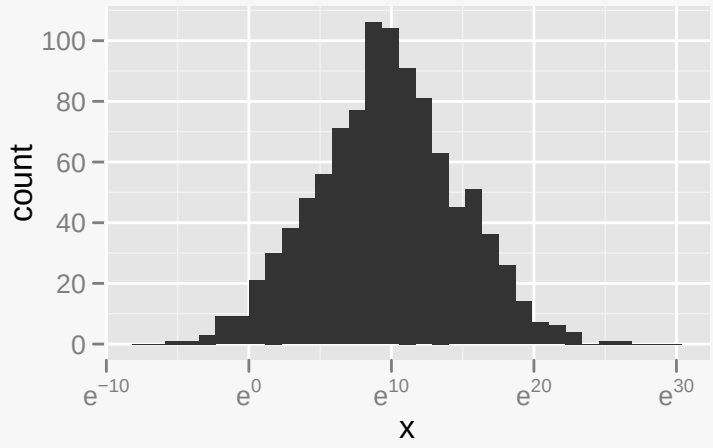


The right side graph would have been called with the following code in 0.8.9, with very different axis labels:

```
p + scale_x_log()
```

Let's suppose we want to label the horizontal axis so that the labels are of the form e^x, where x is an integer, similar to what one gets from the 0.8.9 version. The simplest approach is to transform x in the `ggplot()` call and use `math_format()` to get the desired labels. The token `.x` is a placeholder for the value within `math_format()`, to which the breaks defined in the `qplot()` call are (implicitly) passed.

```
q <- ggplot(DF, aes(x = log(x))) + geom_histogram()
q + scale_x_continuous(labels = math_format(e^.x)) + xlab("x")
```



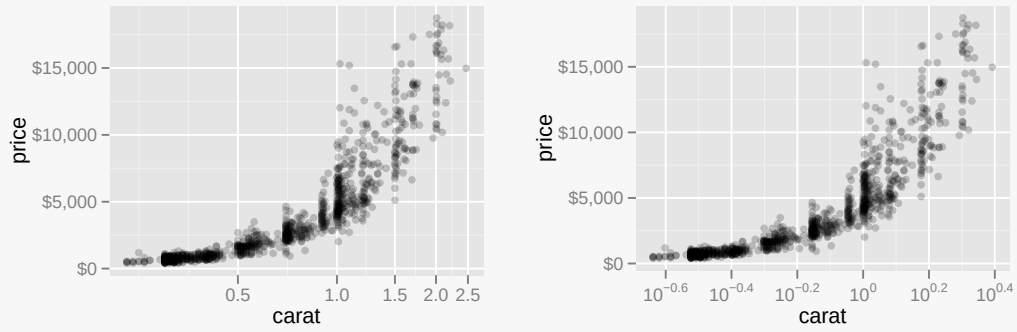
The more formal way to do this, starting from the original histogram `p`, involves use of the `trans =` argument with corresponding breaks and labels functions:

```
p + scale_x_continuous(trans = "log",
                       breaks = trans_breaks("log", "exp"),
                       labels = trans_format("log", math_format(e^.x)))
```

In this call, a (natural) log transformation is applied to the data using the `trans =` argument of `scale_x_continuous()`, the `trans_breaks()` function generates 'pretty' breaks in the log scale and `trans_format()` uses those breaks to produce the labels using the format given in `math_format()`. If you try the above call without `math_format()`, it will provide pretty linear breaks in the log scale by default, which is a sensible scaling *in natural log units*.

Here's a similar example that was discussed on the ggplot2 list recently, using the diamonds data with the `scale_x_log10()` function:

```
dsmall <- diamonds[sample(nrow(diamonds), 1000), ]
## default x-axis log scaling
qplot(carat, price, data = dsmall, alpha = I(0.2)) +
   scale_x_log10() +
   scale_y_continuous(labels = dollar)
## log-linear x-scaling with labels from math_format()
qplot(carat, price, data = dsmall, alpha = I(0.2)) +
   scale_x_log10(breaks = trans_breaks("log10", function(x) 10^x),
                 labels = trans_format("log10", math_format(10^.x))) +
   scale_y_continuous(labels = dollar)
```

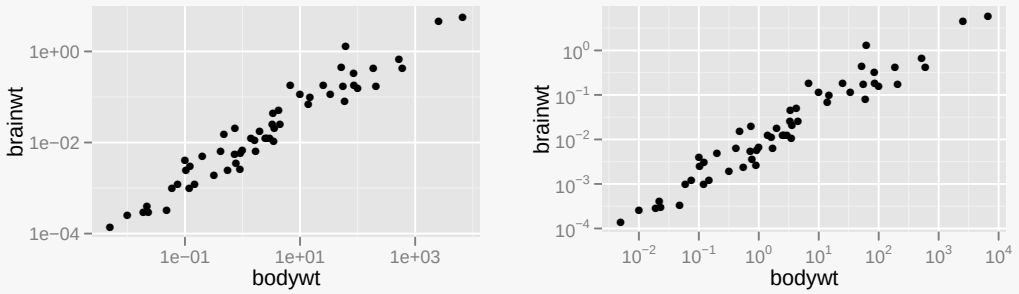


Observe that the inverse function of $\log_{10}$ is defined as a function in `trans_breaks()`.

There is a compelling argument to be made that the breaks in the left plot are more indicative of the original carat values when the plot is rendered in the $\log_{10}$ scale. The default choice of breaks clearly indicates that the variable on the horizontal axis has been transformed.

To define your own scale functions, you will likely need to use `trans_breaks()` and `trans_format()` in some way or another. Here is an example using the `msleep` data from **MASS**, where both brain weight and body weight require a $\log_{10}$ transformation. Suppose you want the 0.8.9 behavior in both positional axis; here's one way to do it.

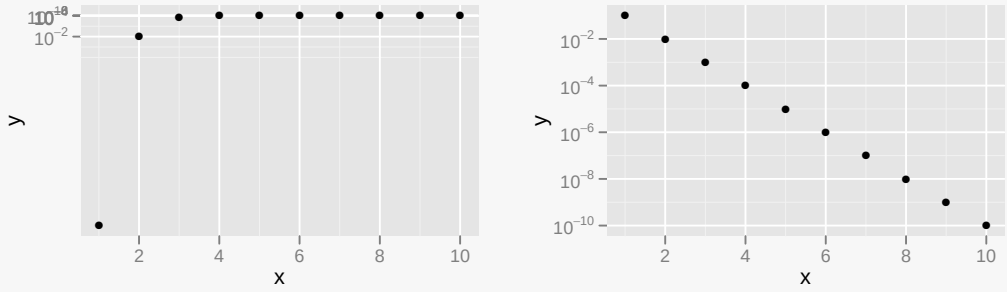
```
# Define 'old style' behavior for log10 axes
scale_x_log10old <- function(...) {
     scale_x_log10(breaks = trans_breaks('log10', function(x) 10^x),
            labels = trans_format('log10', math_format(10^.x)), ...)
    }
scale_y_log10old <- function(...) {
     scale_y_log10(breaks = trans_breaks('log10', function(x) 10^x),
            labels = trans_format('log10', math_format(10^.x)), ...)
    }
# standard 0.9.0 plot on a log10-log10 scale
ggplot(msleep, aes(x = bodywt, y = brainwt)) + geom_point() +
    scale_x_log10() + scale_y_log10()

Warning message: Removed 27 rows containing missing values (geom_point).

# old style axis labeling
ggplot(msleep, aes(x = bodywt, y = brainwt)) + geom_point() +
    scale_x_log10old() + scale_y_log10old()

Warning message: Removed 27 rows containing missing values (geom_point).
```



Another example comes from the ggplot2 list, where someone wanted a 'negative log 10' y-scale. Here are a couple of alternatives to choose from; there are certainly others.

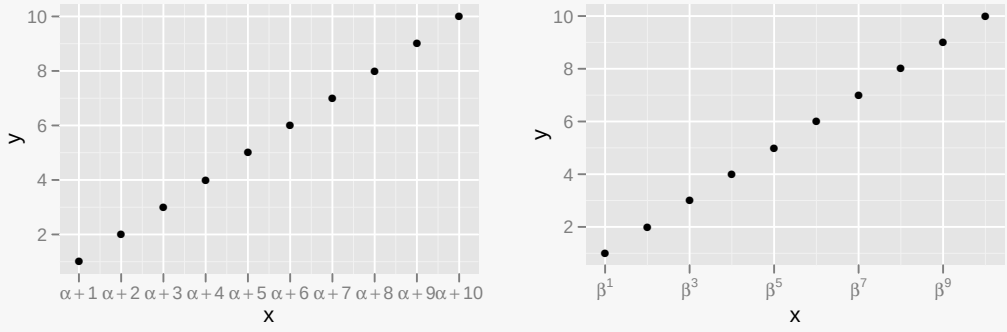
```
# Define the negative log 10 function
neglog10 <- function(x) -log10(x)
# Reverse scaling (zero on top)
scale_y_revneglog10 <- function(...) {
   scale_y_reverse(
     breaks = trans_breaks('neglog10', function(x) 10^(-x)),
     labels = trans_format('neglog10', math_format(10^-.x)), ...)
    }
# Standard negative log 10 scale
scale_y_neglog10 <- function(...) {
```

```
   scale_y_log10(
     breaks = trans_breaks('neglog10', function(x) 10^(-x)),
     labels = trans_format('neglog10', math_format(10^-.x)), ...)
    }
# Example:
DF <- data.frame(x = 1:10, y = 10^-(1:10))
ggplot(DF, aes(x, y)) + geom_point() + scale_y_revneglog10()
ggplot(DF, aes(x, y)) + geom_point() + scale_y_neglog10()
```



The final example in this subsection illustrates a couple more uses of the `math_format()` function to place mathematical expressions in axis labels.

```
d <- data.frame(x = 1:10, y = 1:10)
## Math notation in labels
ggplot(d, aes(x, y)) + geom_point() +
  scale_x_continuous(breaks = 1:10, labels = math_format(alpha + .x))
## Define a vector of breaks manually:
lb <- seq(1, 9, by = 2)
last_plot() +
  scale_x_continuous(breaks = lb, labels = math_format(beta^.x))
```



In both cases, the second set of arguments of `math_format()` are implicit from the defined set of breaks. An expression is passed as the argument to `math_format()`, which returns a

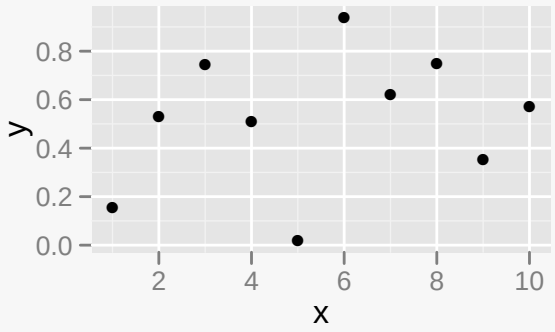
function with argument `.x`. The function call takes the defined breaks as input to create the labels. We could also have called `math_format()` with two explicit sets of arguments as follows:

```
ggplot(d, aes(x, y)) + geom_point() +
  scale_x_continuous(breaks = 1:10,
                     labels = math_format(alpha + .x)(1:10))
last_plot() + scale_x_continuous(breaks = lb,
                                 labels = math_format(beta^.x)(lb))
```

7.3 Scale and coordinate transformations

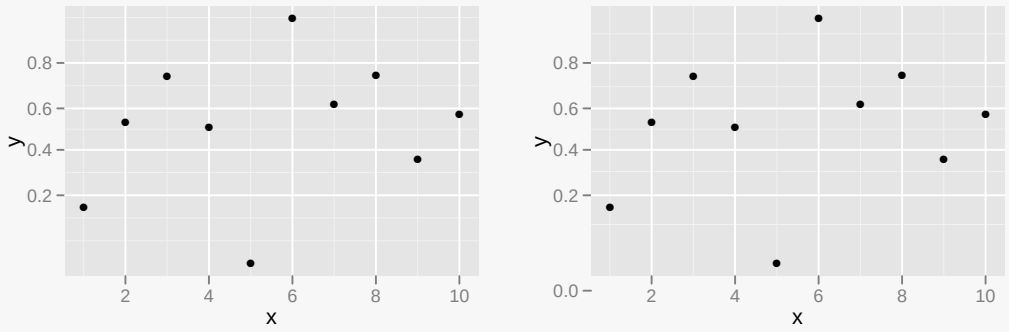
The distinction between scale and coordinate transformations is that scale transformations take place *before* any statistics are computed, whereas coordinate transformations take place *afterward*. In the example below, there is no difference in appearance between apposing pairs of plot calls because no statistical transformation takes place. However, when summary geoms are plotted (e.g., `geom_histogram()` or `geom_density()`), there will often be a difference in appearance depending on when the transformation is applied.

```
# y is a vector of proportions
df1 <- data.frame(x = 1:10, y = round(runif(10), 3))
qplot(x, y, data = df1)
```



Two ways of plotting the $\sin^{-1}$ transformation of y:

```
qplot(x, y, data = df1) + scale_y_continuous(trans = "asn")
qplot(x, y, data = df1) + coord_trans(ytrans = "asn")
```



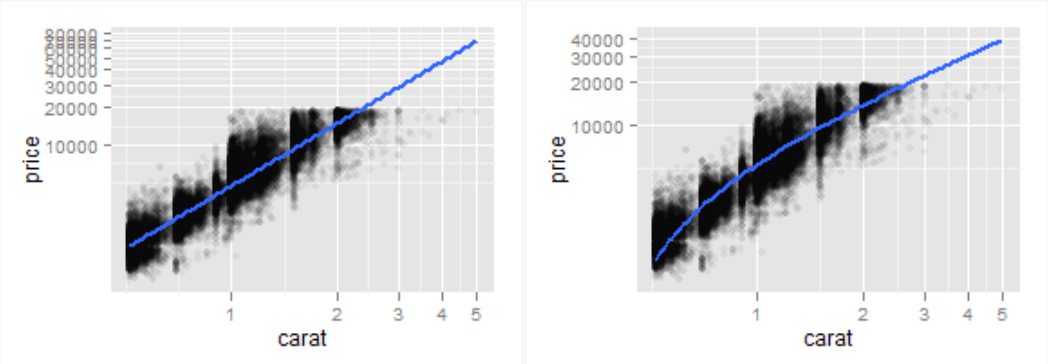
The lognormal example from the previous section shows this game doesn't always work. Run this code in an R session—the second plot call throws an error:

```
p + scale_x_continuous(trans = "log")
p + coord_trans(xtrans = "log")
```

In other words, one needs to exercise some care when using scale transformations as opposed to coordinate transformations when statistical transforms of the data are involved.

Here is an example from the `coord_trans()` help page to illustrate the difference between scale transformations prior to model fitting and coordinate transformations after a model fit. This is one type of graphic that could not be produced easily prior to version 0.9.0.

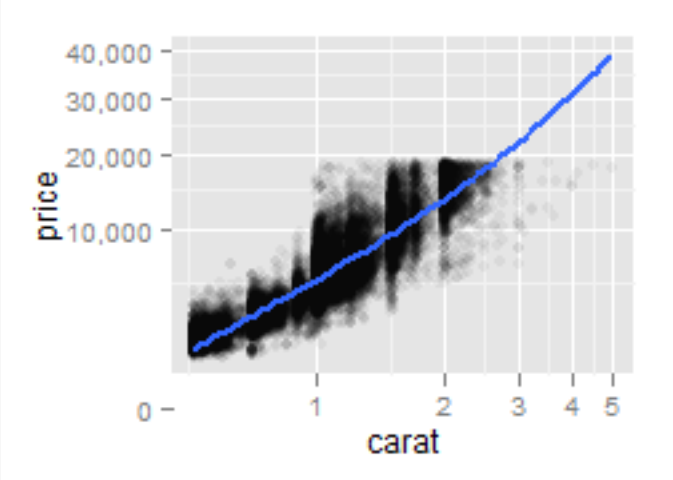
```
d <- subset(diamonds, carat > 0.5)
## Scale transformation prior to model fitting
qplot(carat, price, data = d, log="xy", alpha = I(0.05)) +
   geom_smooth(method="lm", size = 1)
## Coordinate transformation after model fitting
qplot(carat, price, data = d, alpha = I(0.05)) +
   geom_smooth(method="lm", size = 1) +
   coord_trans(x = "log10", y = "log10")
```



In the first graph above, the x and y axes were both log-transformed and then a linear model was fit in the transformed scales. In the second graph, the linear model was fit to the original data first and then the x and y axes were log-transformed. Mathematically, the fitted linear model in the first graph has the form $\widehat{\log y} = b_0 + b_1 \log x$, whereas the fitted model in the second graph is $\log \hat{y} = \log(b_0 + b_1 x)$, which happens to be a transform-both-sides model (Carroll and Ruppert, 1988) since the same coordinate transformation was applied to both sides of the fitted model.

Separate transformations for x and y are also possible:

```
qplot(carat, price, data = d, alpha = I(0.05)) +
   geom_smooth(method="lm", size = 1) +
   coord_trans(x = "log10", y = "sqrt") +
   scale_y_continuous(labels = comma_format(digits = 5))
```



8 Summary

Several new features in **ggplot2-0.9.0** have been introduced and illustrated by example. We have seen that a useR now has more tools at one's disposal with respect to producing legends and positional axis scales, some nice new features and improvements have taken place in `facet_grid()` and several new geoms that extend the scope of graphics applications in the package have been introduced with accompanying examples of their use. A number of incremental improvements have been made both in design and execution; many of these were described in section 1, but a more complete list is found in the NEWS file of the package.

Thanks to members of the `ggplot2-dev` list for comments on earlier drafts of this document. Notably, Baptiste Auguie suggested the section on annotations and provided valuable guidance and examples, and Winston Chang provided numerous insights and suggestions regarding `geom_dotplot()` and `geom_violin()`.

The document was processed using the **knitr** package, developed by Yihui Xie `http://yihui.name/knitr/`. Special thanks to Yihui for patiently answering many elementary questions about the package.

The source code for this version is located at
`https://github.com/djmurphy420/ggplot2-transition-guide`
in the file `ggplot2-0.9.0.Rnw`. The README file has instructions for processing the `.Rnw` file for color or black-and-white printing.

We hope that this guide is helpful as you move from 0.8.9 to 0.9.0 and that the changes and new facilities of the package aid you in making even better ggplots.

9 References

Carroll, R. J. and D. Huppert (1988). *Transformation and Weighting in Regression*. New York: Chapman and Hall.

Tukey, J. and P. Tukey (1990). *Strips Displaying Empirical Distributions: I. Textured Dot Strips*. Technical Memorandum, Bellcore.

Wickham, H. (2009). *ggplot2: Elegant graphics for data analysis.* New York: Springer.

Wilkinson, L. (1999). Dot Plots. *American Statistician*, **53**(3), 276–281.

Wilkinson, L. (2005). *The Grammar of Graphics*, 2nd ed. New York: Springer.

Xie, Y. (2012). *knitr: A General-Purpose Tool for Dynamic Report Generation in R*. `http://yihui.github.com/knitr/demo/manual/`.