Getting started with qplot

Introduction

In this vignette, you will learn to make a wide variety of plots with your first ggplot2 function, `qplot`, short for **q**uick plot. `qplot` makes it easy to produce complex plots, often requiring several lines of code using other plotting systems, in one line. `qplot()` can do this because it's based on the grammar of graphics, which allows you to create a simple, yet expressive, description of the plot.

`qplot()` has been designed to be very similar to `plot()`, which should make it easy if you're already familiar with plotting in R. Remember, during an R session you can get a summary of all the arguments to `qplot()` with R help, `?qplot`.

In this vignette you'll learn:

- The basic use of `qplot`. If you're already familiar with `plot()`, this will be particularly easy.
- How to map variables to aesthetic attributes, like colour, size and shape.
- How to create many different types of plots by specifying different geoms, and how to combine multiple types in a single plot.
- The use of faceting, also known as trellising or conditioning, to break apart subsets of your data.
- How to tune the appearance of the plot by specifying some basic options.
- A few important differences between `plot()` and `qplot()`

Datasets

In this chapter we'll just use one data source, so you can get familiar with the plotting details rather than having to familiarise yourself with different datasets. The `diamonds` dataset consists of prices and quality information about 54,000 diamonds, and is included in the ggplot2 package. The data contains the four C's of diamond quality, carat, cut, colour and clarity; and five physical measurements, depth, table, x, y and z. The first few rows of the data are shown below:

```
head(diamonds)
```

```
#>   carat       cut color clarity depth table price    x    y    z
#> 1  0.23     Ideal     E     SI2  61.5    55   326 3.95 3.98 2.43
#> 2  0.21   Premium     E     SI1  59.8    61   326 3.89 3.84 2.31
#> 3  0.23      Good     E     VS1  56.9    65   327 4.05 4.07 2.31
#> 4  0.29   Premium     I     VS2  62.4    58   334 4.20 4.23 2.63
#> 5  0.31      Good     J     SI2  63.3    58   335 4.34 4.35 2.75
#> 6  0.24 Very Good     J    VVS2  62.8    57   336 3.94 3.96 2.48
```

The dataset has not been well cleaned, so as well as demonstrating interesting relationships about diamonds, it also demonstrates some data quality problems. We'll also use another dataset, `dsmall`, which is a random sample of 100 diamonds. We'll use this data for plots that are more appropriate for smaller datasets.

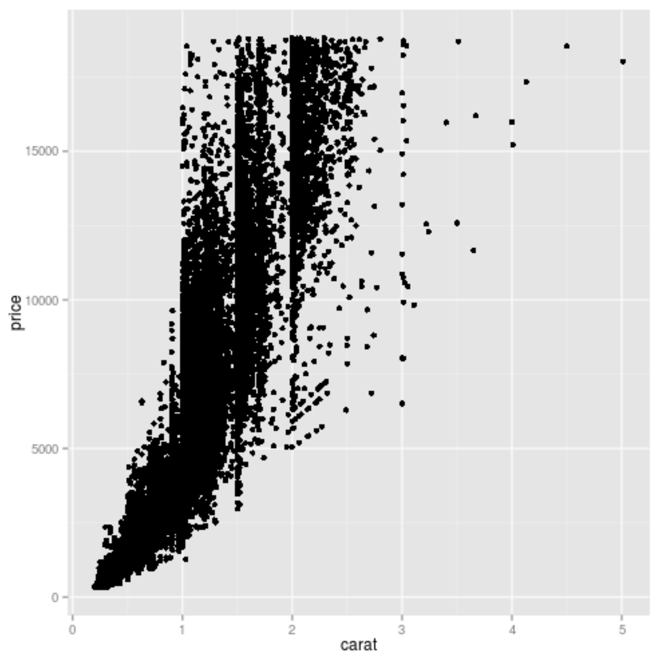
```
set.seed(1410) # Make the sample reproducible
dsmall <- diamonds[sample(nrow(diamonds), 100), ]
```

Basic use

As with `plot`, the first two arguments to `qplot()` are `x` and `y`, giving the x- and y-coordinates for the objects on the plot. There is also an optional `data` argument. If this is specified, `qplot()` will look inside that data frame before looking for objects in your workspace. Using the `data` argument is recommended: it's a good idea to keep related data in a single data frame. If you don't specify one, `qplot()` will try to build one up for you and may look in the wrong place.

Here is a simple example of the use of `qplot()`. It produces a scatterplot showing the relationship between the price and carats (weight) of a diamond.

```
qplot(carat, price, data = diamonds)
```



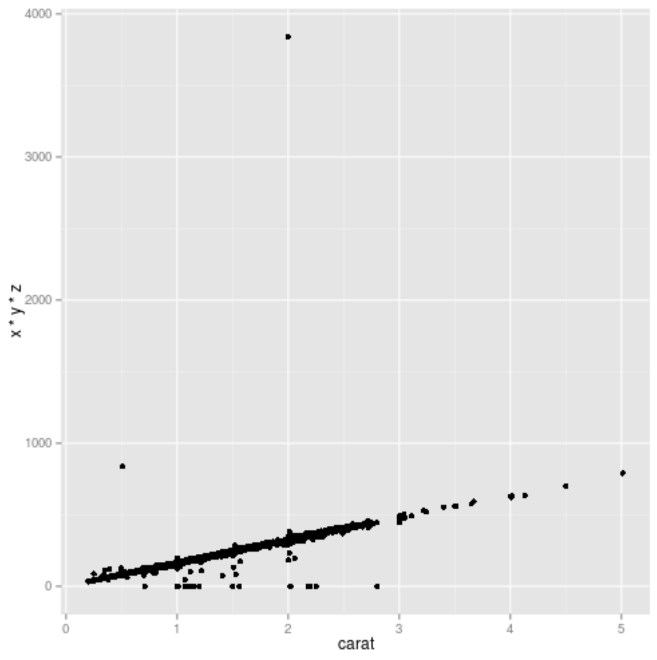
The plot shows a strong correlation with notable outliers and some interesting vertical striation. The relationship looks exponential, though, so the first thing we'd like to do is to transform the variables. Because `qplot()` accepts functions of variables as arguments, we plot `log(price)` vs. `log(carat)`:

```
qplot(log(carat), log(price), data = diamonds)
```

The relationship now looks linear. With this much overplotting, though, we need to be cautious about drawing firm conclusions.

Arguments can also be combinations of existing variables, so, if we are curious about the relationship between the volume of the diamond (approximated by `x * y * z`) and its weight, we could do the following:

```
qplot(carat, x * y * z, data = diamonds)
```



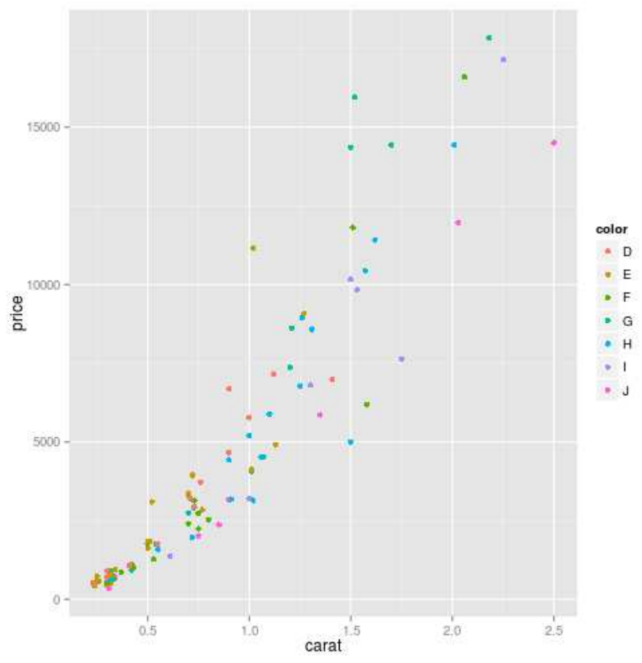
We would expect the density (weight/volume) of diamonds to be constant, and so see a linear relationship between volume and weight. The majority of diamonds do seem to fall along a line, but there are some large outliers.

Colour, size, shape and other aesthetic attributes

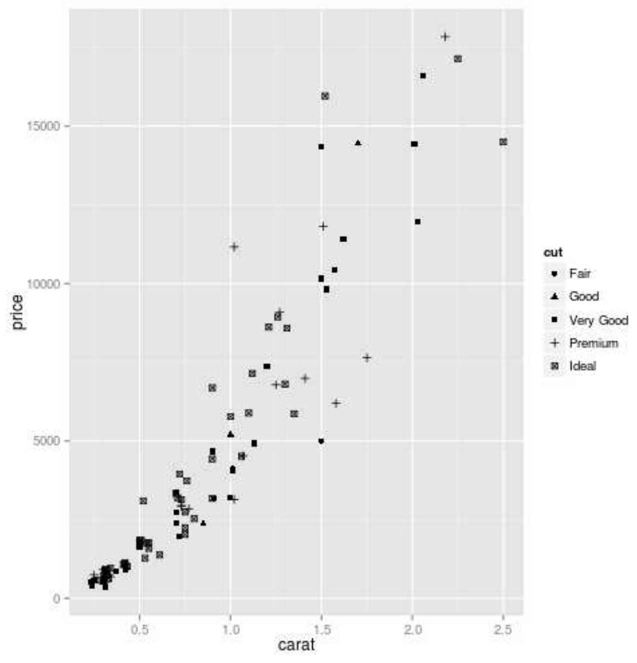
The first big difference when using `qplot` instead of `plot` comes when you want to assign colours—or sizes or shapes—to the points on your plot. With `plot`, it's your responsibility to convert a categorical variable in your data (e.g., `apples'',`bananas", `pears'') into something that `plot` knows how to use (e.g.,`red", `yellow'',`green"). `qplot` can do this for you automatically, and it will automatically provide a legend that maps the displayed attributes to the data values. This makes it easy to include additional data on the plot.

In the next example, we augment the plot of carat and price with information about diamond colour and cut:

```
qplot(carat, price, data = dsmall, colour = color)
```



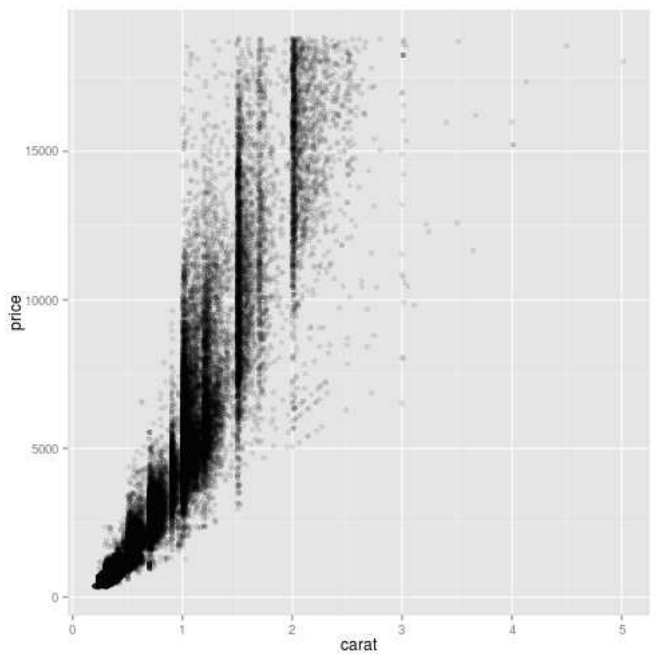
```
qplot(carat, price, data = dsmall, shape = cut)
```



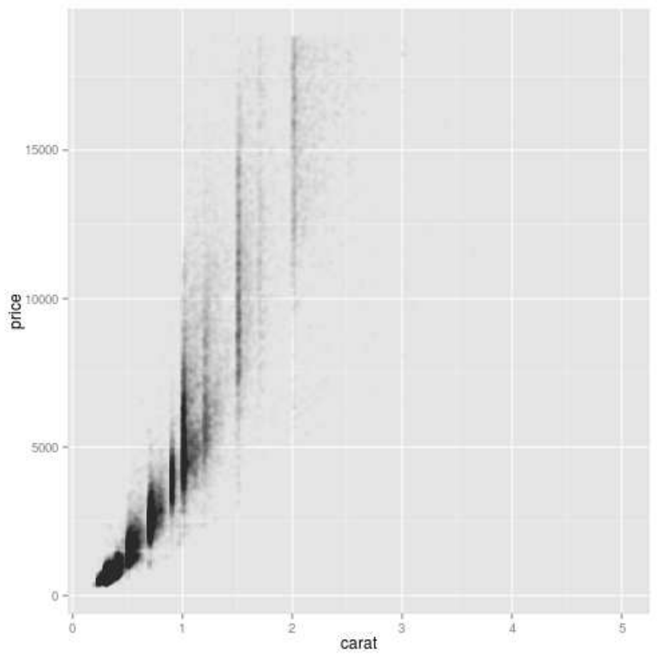
Colour, size and shape are all examples of aesthetic attributes, visual properties that affect the way observations are displayed. For every aesthetic attribute, there is a function, called a *scale*, which maps data values to valid values for that aesthetic. It is this scale that controls the appearance of the points and associated legend. For example, in the above plots, the colour scale maps J to purple and F to green. (Note that while I use British spelling, the ggplot2 also understands American.)

You can also manually set the aesthetics using `I()`, e.g., `colour = I("red")` or `size = I(2)`. This is not the same as mapping and is explained in more detail in Section~\ref{sub:setting-mapping}. For large datasets, like the diamonds data, semi-transparent points are often useful to alleviate some of the overplotting. To make a semi-transparent colour you can use the alpha aesthetic, which takes a value between 0 (completely transparent) and 1 (complete opaque). It's often useful to specify the transparency as a fraction, e.g., `1/10` or `1/20`, as the denominator specifies the number of points that must overplot to get a completely opaque colour.

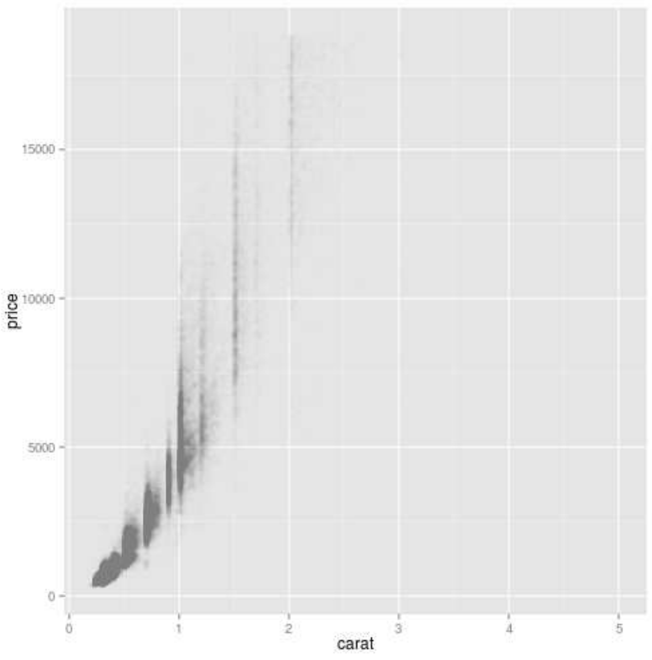
```
qplot(carat, price, data = diamonds, alpha = I(1/10))
```



```
qplot(carat, price, data = diamonds, alpha = I(1/100))
```



```
qplot(carat, price, data = diamonds, alpha = I(1/200))
```



Different types of aesthetic attributes work better with different types of variables. For example, colour and shape work well with categorical variables, while size works better with continuous variables. The amount of data also makes a difference: if there is a lot of data, like in the plots above, it can be hard to distinguish the different groups. An alternative solution is to use faceting.

Plot geoms

`qplot` is not limited to scatterplots, but can produce almost any kind of plot by varying the `geom`. Geom, short for geometric object, describes the type of object that is used to display the data. Some geoms have an associated statistical transformation, for example, a histogram is a binning statistic plus a bar geom. These different components are described in the next chapter. Here we'll introduce the most common and useful geoms, organised by the dimensionality of data that they work with. The following geoms enable you to investigate two-dimensional relationships:

- `geom = "point"` draws points to produce a scatterplot. This is the default when you supply both `x` and `y` arguments to `qplot()`.
- `geom = "smooth"` fits a smoother to the data and displays the smooth and its standard error.
- `geom = "boxplot"` produces a box-and-whisker plot to summarise the distribution of a set of points.
- `geom = "path"` and `geom = "line"` draw lines between the data points. Traditionally these are used to explore relationships between time and another variable, but lines may be used to join observations connected in some other way. A line plot is constrained to produce lines that travel from left to right, while paths can go in any direction.

For 1d distributions, your choice of geoms is guided by the variable type:

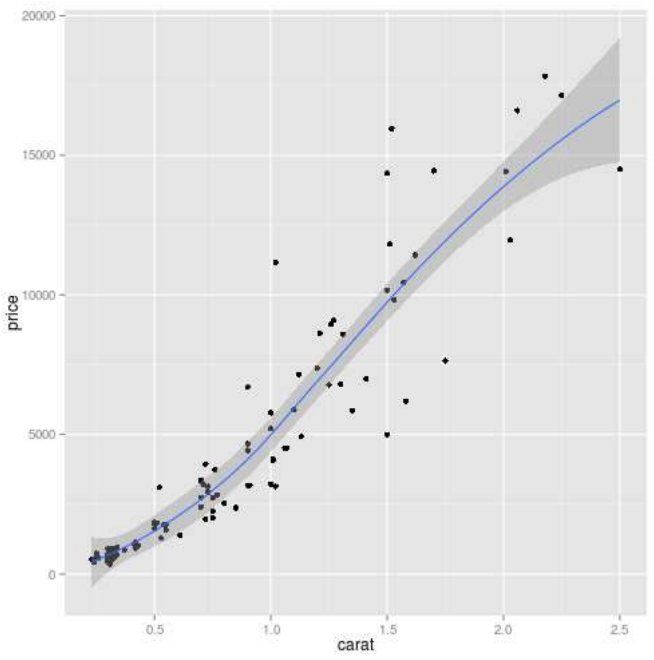
- For continuous variables, `geom = "histogram"` draws a histogram, `geom = "freqpoly"` a frequency polygon, and `geom = "density"` creates a density plot, \secref{sub:distribution}. The histogram geom is the default when you only supply an `x` value to `qplot()`.
- For discrete variables, `geom = "bar"` makes a bar chart.

Adding a smoother to a plot

If you have a scatterplot with many data points, it can be hard to see exactly what trend is shown by the data. In this case you may want to add a smoothed line to the plot. This is easily done using the `smooth` geom as shown in Figure~\ref{fig:qplot-smooth}. Notice that we have combined multiple geoms by supplying a vector of geom names created with `c()`. The geoms will be overlaid in the order in which they appear.

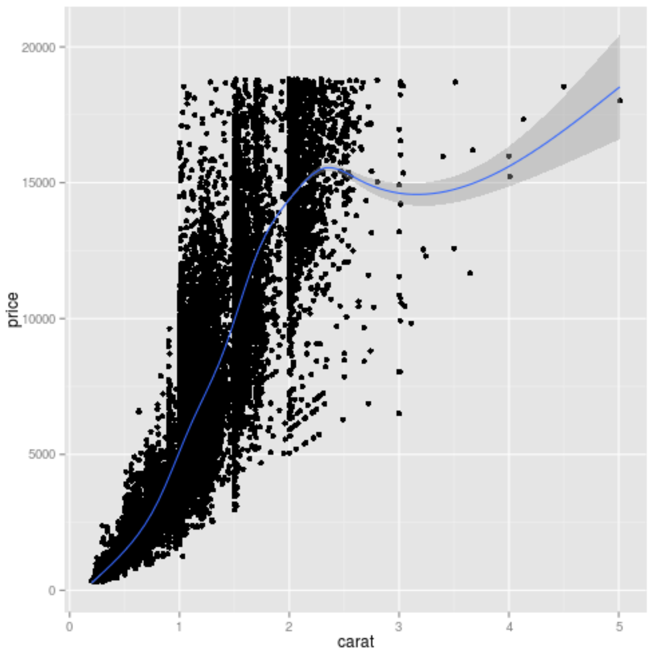
```
qplot(carat, price, data = dsmall, geom = c("point", "smooth"))
```

```
#> geom_smooth: method="auto" and size of largest group is <1000, so using loess. Use 'method = x' to change the sı
```



```
qplot(carat, price, data = diamonds, geom = c("point", "smooth"))
```

```
#> geom_smooth: method="auto" and size of largest group is >=1000, so using gam with formula: y ~ s(x, bs = "cs").
```



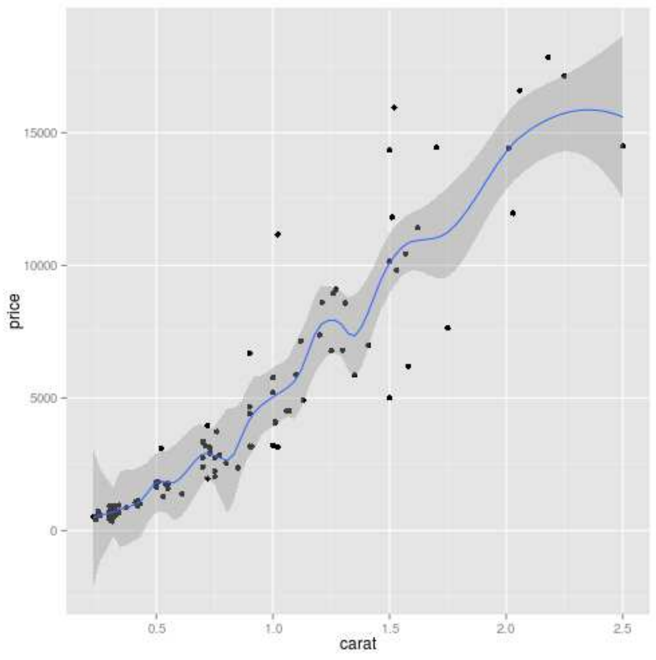
Despite overplotting, our impression of an exponential relationship between price and carat was correct. There are few diamonds bigger than three carats, and our uncertainty in the form of the relationship increases as illustrated by the point-wise confidence interval shown in grey. If you want to turn the confidence interval off, use `se = FALSE`.

There are many different smoothers you can choose between by using the `method` argument:

- `method = "loess"`, the default for small n, uses a smooth local regression. More details about the algorithm used can be found in `?loess`. The wiggliness of the line is controlled by the `span` parameter, which ranges from 0 (exceedingly wiggly) to 1 (not so wiggly), as shown in below.

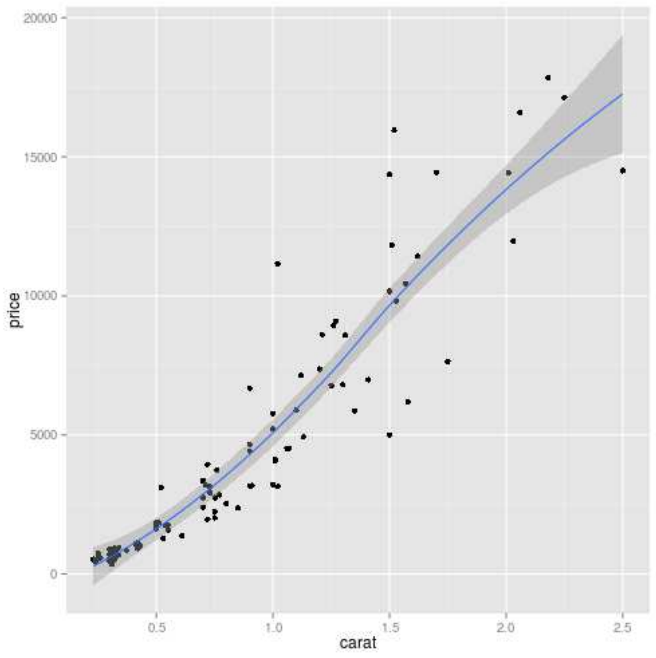
  ```
  qplot(carat, price, data = dsmall, geom = c("point", "smooth"),
    span = 0.2)
  ```

  ```
  #> geom_smooth: method="auto" and size of largest group is <1000, so using loess. Use 'method = x' to change t
  ```



```
qplot(carat, price, data = dsmall, geom = c("point", "smooth"),
  span = 1)
```

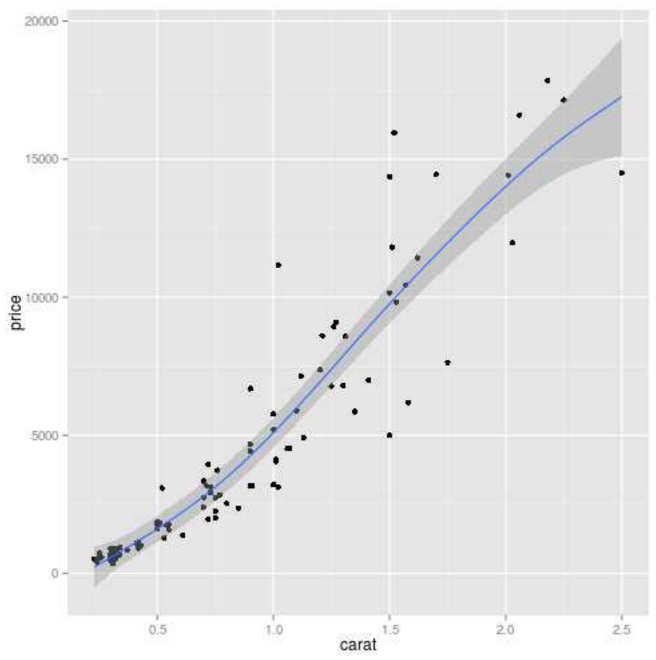
```
#> geom_smooth: method="auto" and size of largest group is <1000, so using loess. Use 'method = x' to change t
```



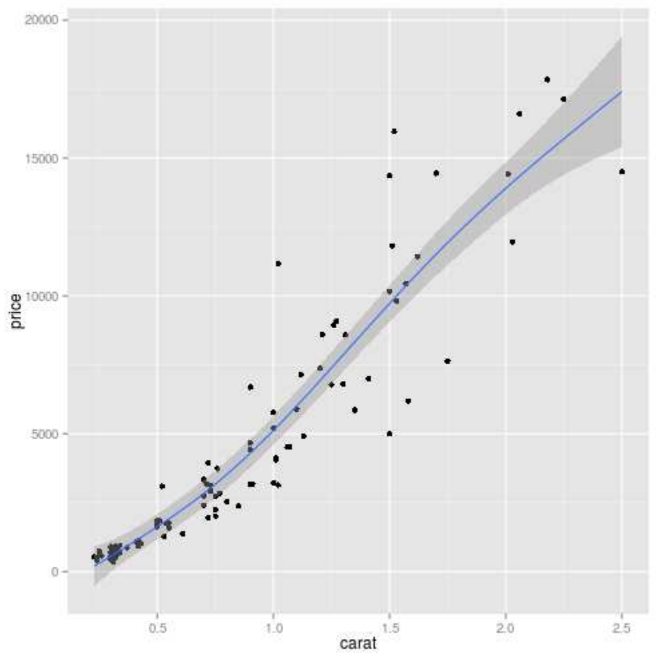
Loess does not work well for large datasets (it's O(n^2) in memory), and so an alternative smoothing algorithm is used when n is greater than 1,000.

- You could also load the `mgcv` library and use `method = "gam", formula = y ~ s(x)` to fit a generalised additive model. This is similar to using a spline with `lm`, but the degree of smoothness is estimated from the data. For large data, use the formula `y ~ s(x, bs = "cs")`. This is used by default when there are more than 1,000 points.

```
library(mgcv)
qplot(carat, price, data = dsmall, geom = c("point", "smooth"),
 method = "gam", formula = y ~ s(x))
```

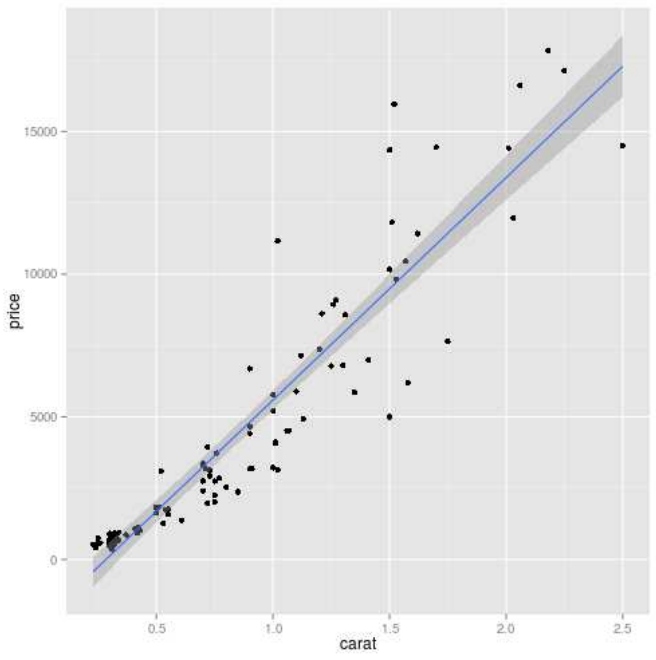


```
qplot(carat, price, data = dsmall, geom = c("point", "smooth"),
 method = "gam", formula = y ~ s(x, bs = "cs"))
```

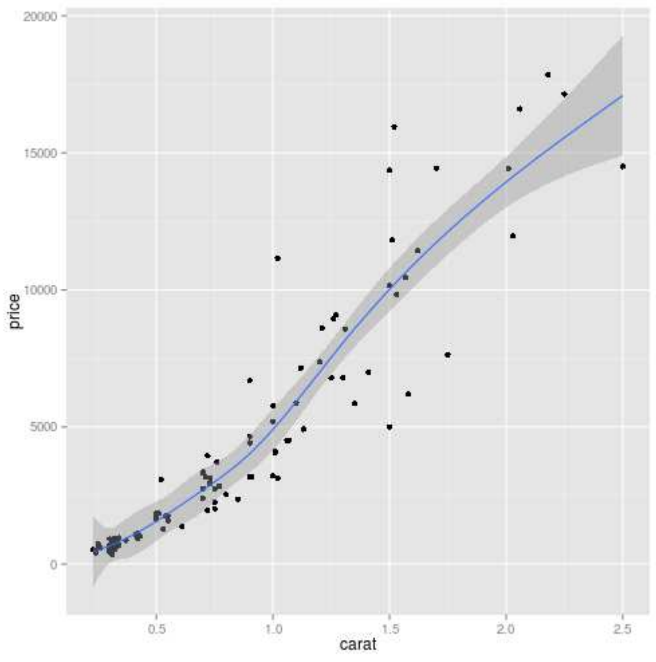


- `method = "lm"` fits a linear model. The default will fit a straight line to your data, or you can specify `formula = y ~ poly(x, 2)` to specify a degree 2 polynomial, or better, load the `splines` package and use a natural spline: `formula = y ~ ns(x, 2)`. The second parameter is the degrees of freedom: a higher number will create a wigglier curve. You are free to specify any formula involving `x` and `y`.

```
library(splines)
qplot(carat, price, data = dsmall, geom = c("point", "smooth"),
  method = "lm")
```



```
qplot(carat, price, data = dsmall, geom = c("point", "smooth"),
  method = "lm", formula = y ~ ns(x,5))
```

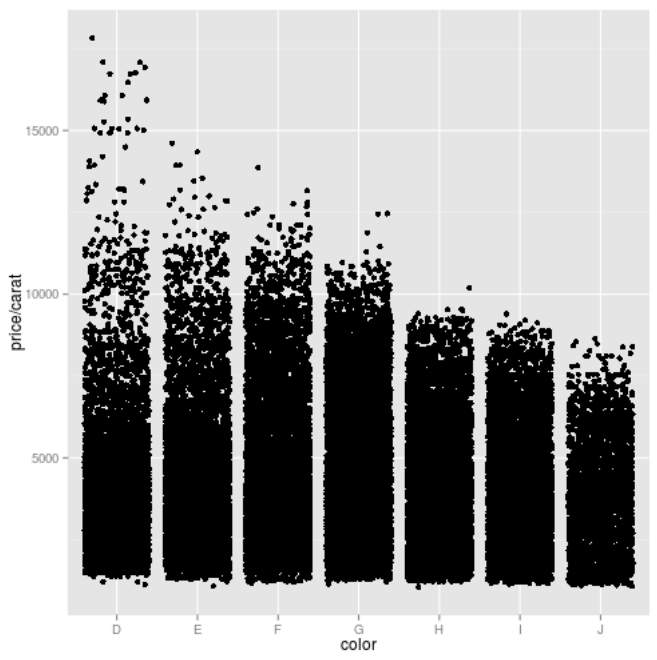


- `method = "rlm"` works like `lm`, but uses a robust fitting algorithm so that outliers don't affect the fit as much. It's part of the `MASS` package, so remember to load that first.

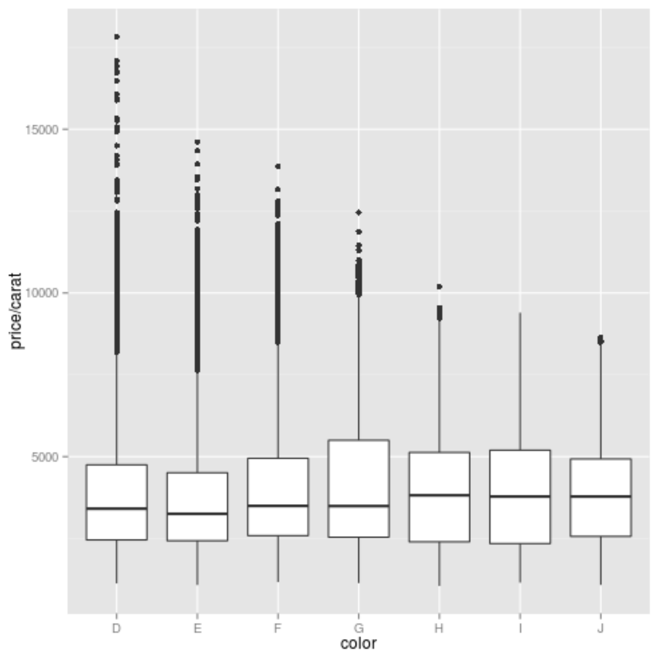
Boxplots and jittered points

When a set of data includes a categorical variable and one or more continuous variables, you will probably be interested to know how the values of the continuous variables vary with the levels of the categorical variable. Box-plots and jittered points offer two ways to do this. The following example explores how the distribution of price per carat varies with the colour of the diamond using jittering (`geom = "jitter"`, left) and box-and-whisker plots (`geom = "boxplot"`, right).

```
qplot(color, price / carat, data = diamonds, geom = "jitter")
```



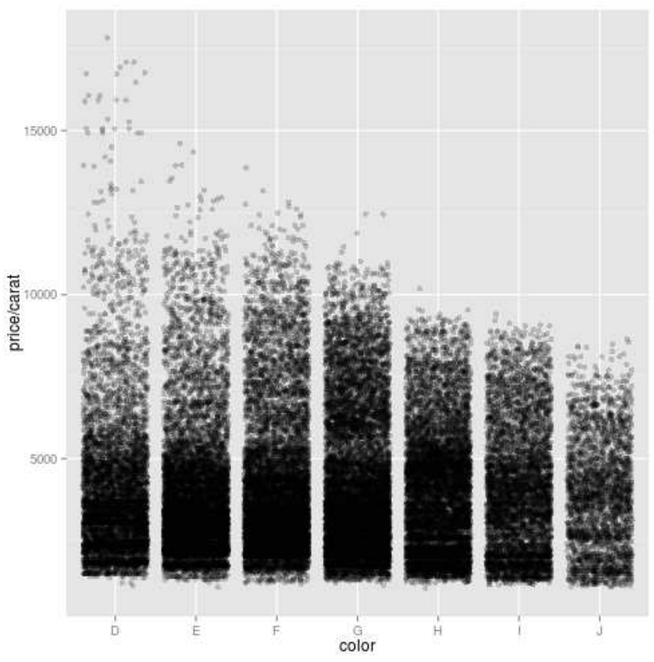
```
qplot(color, price / carat, data = diamonds, geom = "boxplot")
```



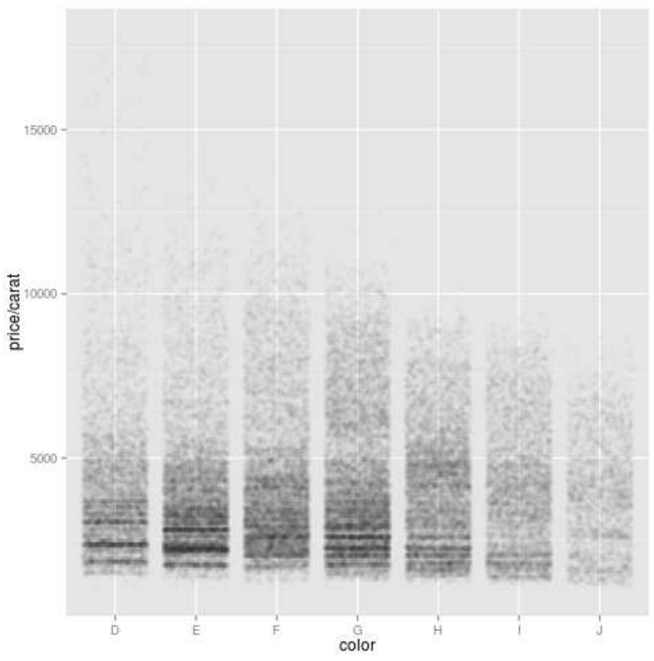
Each method has its strengths and weaknesses. Boxplots summarise the bulk of the distribution with only five numbers, while jittered plots show every point but can suffer from overplotting. In the example here, both plots show the dependency of the spread of price per carat on diamond colour, but the boxplots are more informative, indicating that there is very little change in the median and adjacent quartiles.

The overplotting seen in the plot of jittered values can be alleviated somewhat by using semi-transparent points using the `alpha` argument. The next example illustrates three different levels of transparency, which make it easier to see where the bulk of the points lie.

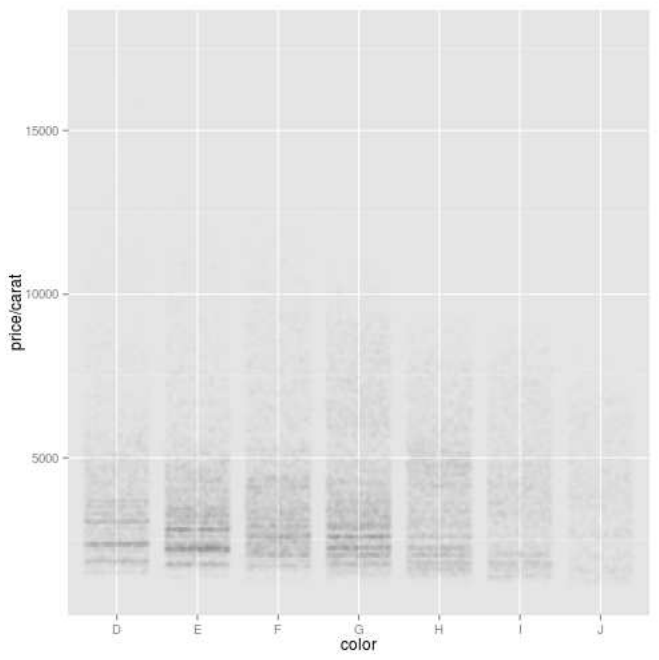
```
qplot(color, price / carat, data = diamonds, geom = "jitter",
  alpha = I(1 / 5))
```



```
qplot(color, price / carat, data = diamonds, geom = "jitter",
  alpha = I(1 / 50))
```



```
qplot(color, price / carat, data = diamonds, geom = "jitter",
  alpha = I(1 / 200))
```



This technique can't show the positions of the quantiles as well as a boxplot can, but it may reveal other features of the distribution that a boxplot cannot.

For jittered points, `qplot` offers the same control over aesthetics as it does for a normal scatterplot: `size`, `colour` and `shape`. For boxplots you can control the outline `colour`, the internal `fill` colour and the `size` of the lines.

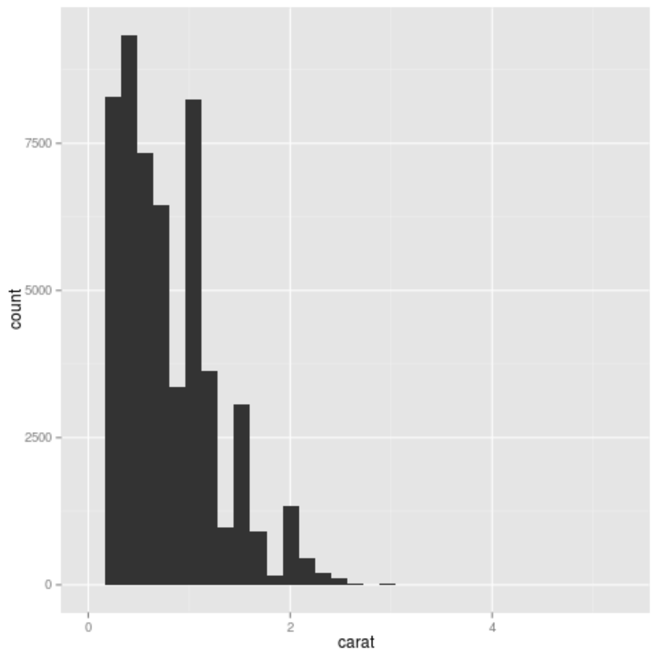
Another way to look at conditional distributions is to use faceting to plot a separate histogram or density plot for each value of the categorical variable.

Histogram and density plots

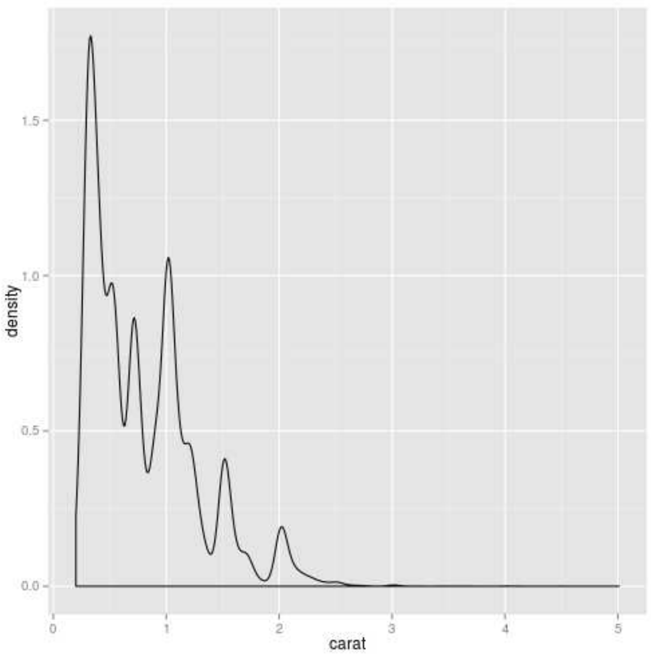
Histogram and density plots show the distribution of a single variable. They provide more information about the distribution of a single group than boxplots do, but it is harder to compare many groups (although we will look at one way to do so). The following code shows the distribution of carats with a histogram and a density plot.

```
qplot(carat, data = diamonds, geom = "histogram")
```

```
#> stat_bin: binwidth defaulted to range/30. Use 'binwidth = x' to adjust this.
```



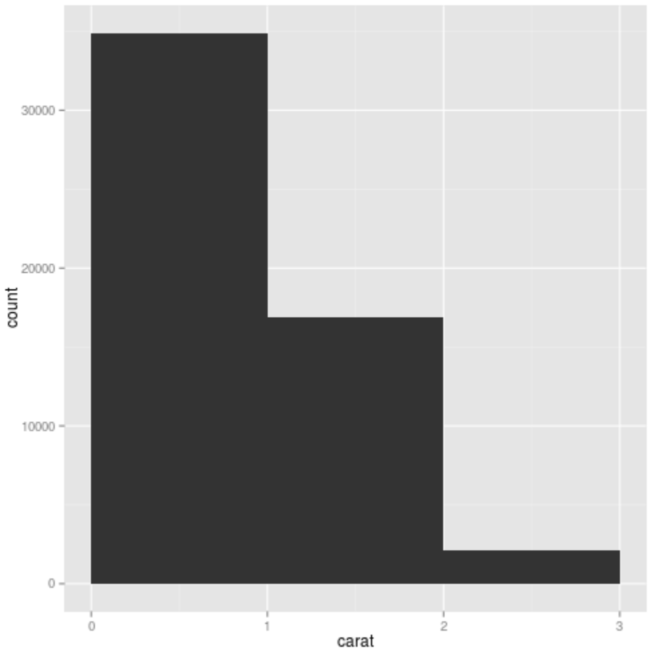
```
qplot(carat, data = diamonds, geom = "density")
```



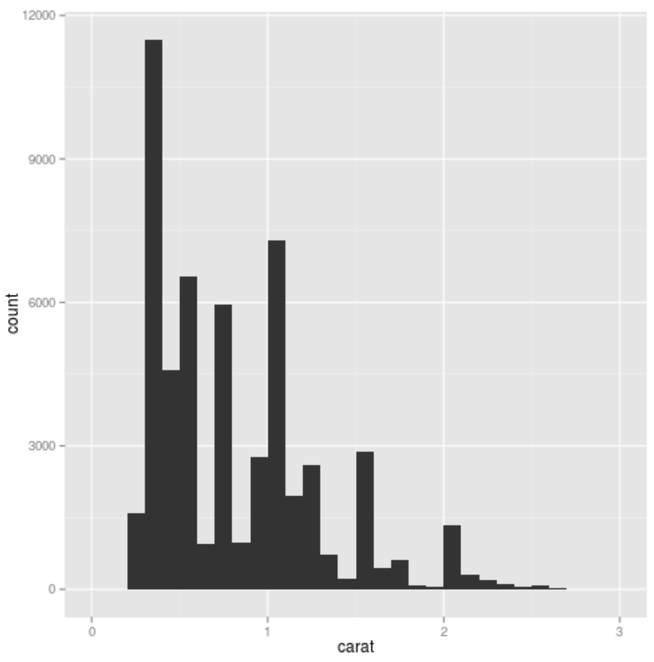
For the density plot, the `adjust` argument controls the degree of smoothness (high values of `adjust` produce smoother plots). For the histogram, the `binwidth` argument controls the amount of smoothing by setting the bin size. (Break points can also be specified explicitly, using the `breaks` argument.) It is **very important** to experiment with the level of smoothing. With a histogram you should try many bin widths: You may find that gross features of the data show up well at a large bin width, while finer features require a very narrow width.

In this following example, we experiment with three values of `binwidth`: 1.0, 0.1 and 0.01. It is only in the plot with the smallest bin width (right) that we see the striations we noted in an earlier scatterplot, most at “nice” numbers of carats. The full code is:

```
qplot(carat, data = diamonds, geom = "histogram", binwidth = 1,
  xlim = c(0,3))
```

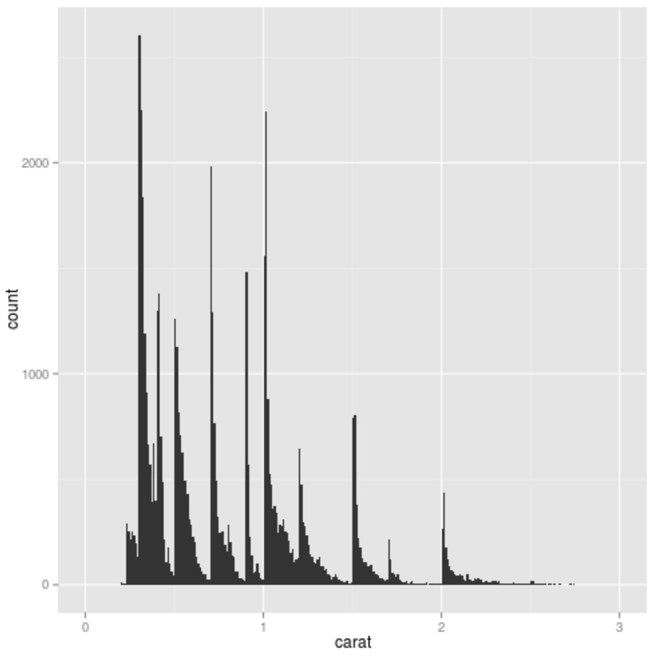


```
qplot(carat, data = diamonds, geom = "histogram", binwidth = 0.1,
  xlim = c(0,3))
```



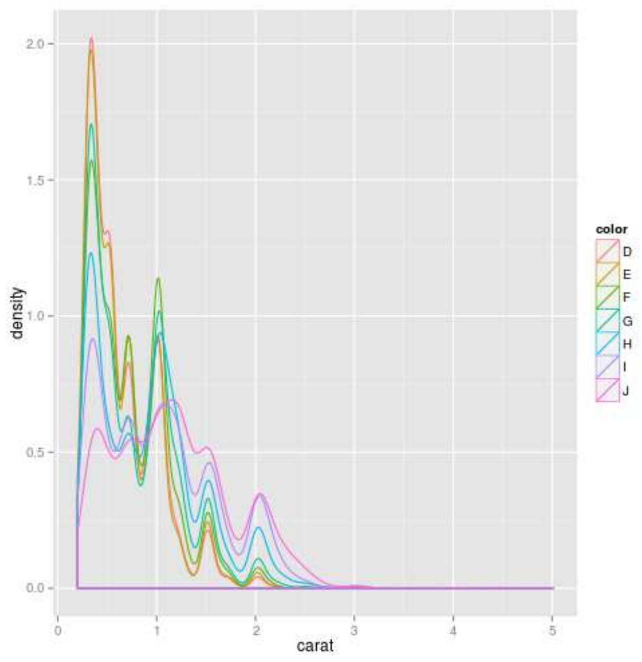
```
qplot(carat, data = diamonds, geom = "histogram", binwidth = 0.01,
  xlim = c(0,3))
```

```
#> Warning: position_stack requires constant width: output may be incorrect
```



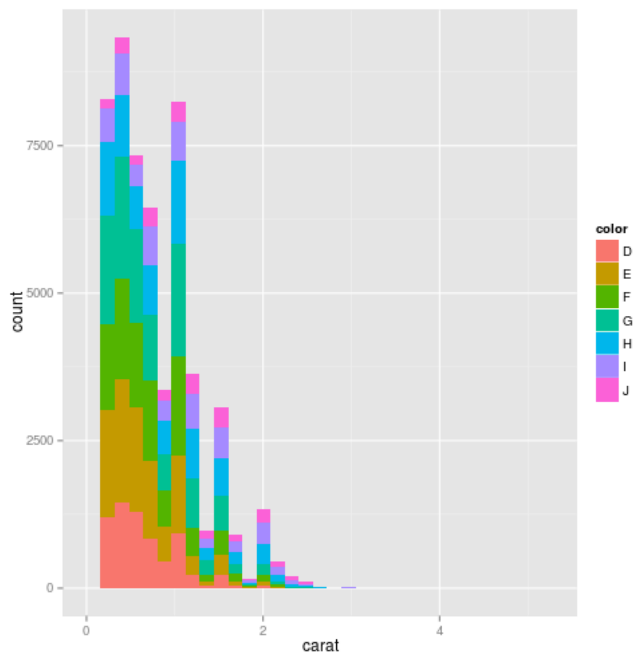
To compare the distributions of different subgroups, just add an aesthetic mapping, as in the following code. Mapping a categorical variable to an aesthetic will automatically split up the geom by that variable, so these commands instruct `qplot()` to draw a density plot and histogram for each level of diamond colour.

```
qplot(carat, data = diamonds, geom = "density", colour = color)
```



```
qplot(carat, data = diamonds, geom = "histogram", fill = color)
```

```
#> stat_bin: binwidth defaulted to range/30. Use 'binwidth = x' to adjust this.
```

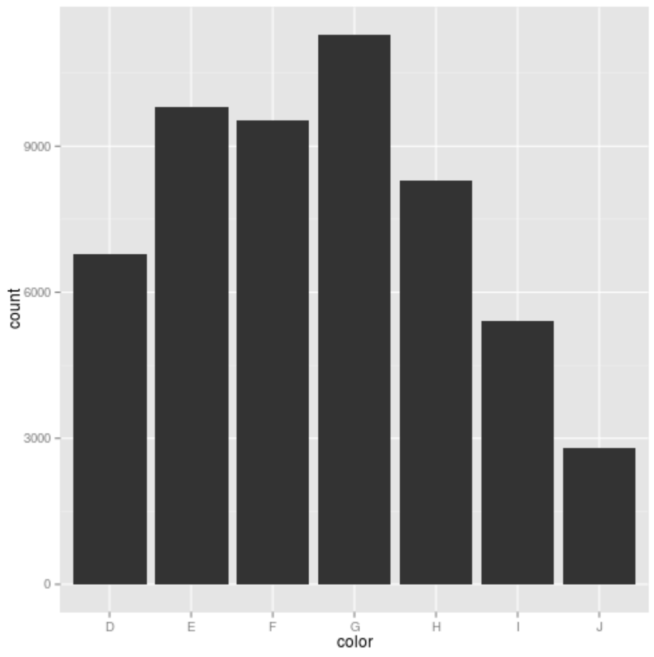


The density plot is more appealing at first because it seems easy to read and compare the various curves. However, it is more difficult to understand exactly what a density plot is showing. In addition, the density plot makes some assumptions that may not be true for our data; i.e., that it is unbounded, continuous and smooth.

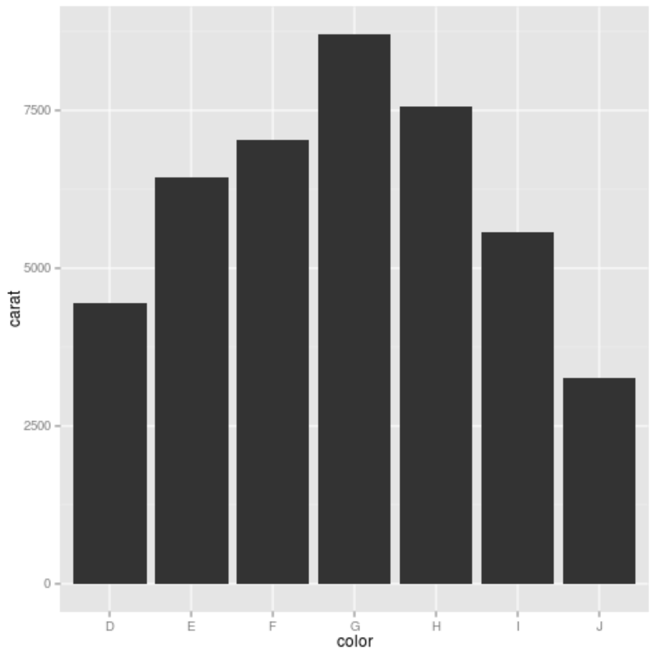
Bar charts

The discrete analogue of histogram is the bar chart, `geom = "bar"`. The bar geom counts the number of instances of each class so that you don't need to tabulate your values beforehand, as with `barchart` in base R. If the data has already been tabulated or if you'd like to tabulate class members in some other way, such as by summing up a continuous variable, you can use the `weight` geom. The first plot is a simple bar chart of diamond colour, and the second is a bar chart of diamond colour weighted by carat.

```
qplot(color, data = diamonds, geom = "bar")
```



```
qplot(color, data = diamonds, geom = "bar", weight = carat) +
  scale_y_continuous("carat")
```

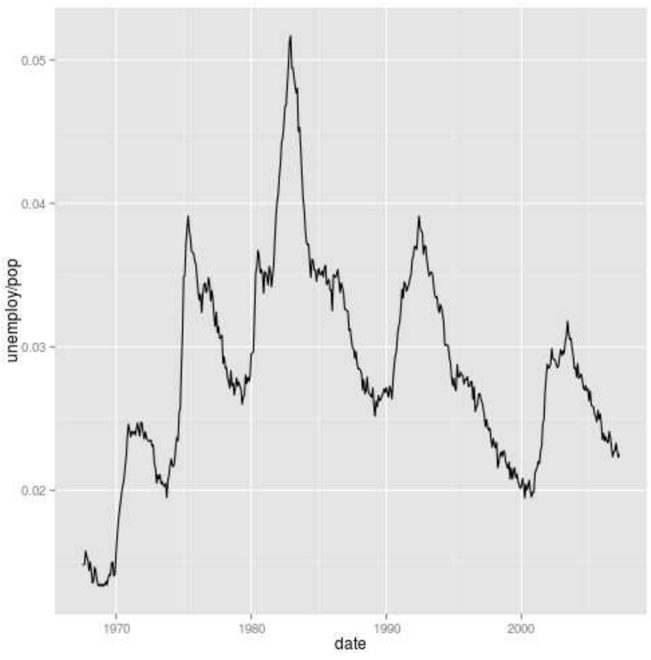


Time series with line and path plots

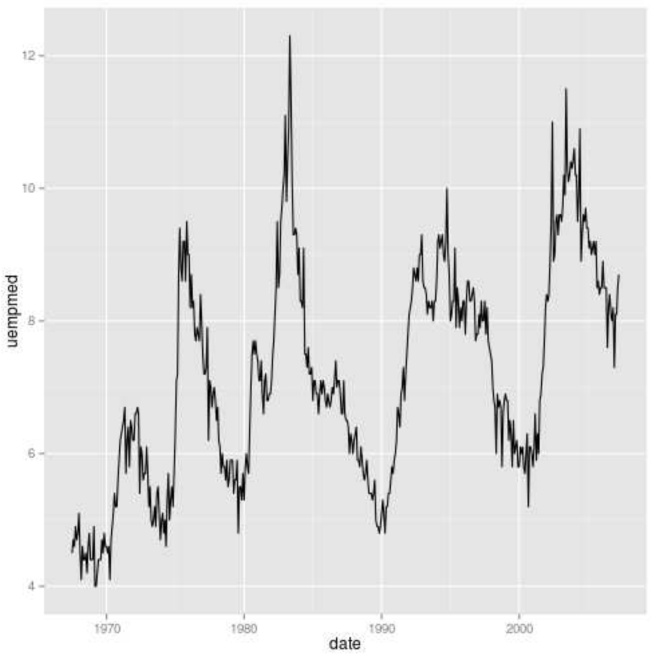
Line and path plots are typically used for time series data. Line plots join the points from left to right, while path plots join them in the order that they appear in the dataset (a line plot is just a path plot of the data sorted by x value). Line plots usually have time on the x-axis, showing how a single variable has changed over time. Path plots show how two variables have simultaneously changed over time, with time encoded in the way that the points are joined together.

Because there is no time variable in the diamonds data, we use the `economics` dataset, which contains economic data on the US measured over the last 40 years. The following examples shows two plots of unemployment over time, both produced using `geom = "line"`. The first shows an unemployment rate and the second shows the median number of weeks unemployed. We can already see some differences in these two variables, particularly in the last peak, where the unemployment percentage is lower than it was in the preceding peaks, but the length of unemployment is high.

```
qplot(date, unemploy / pop, data = economics, geom = "line")
```



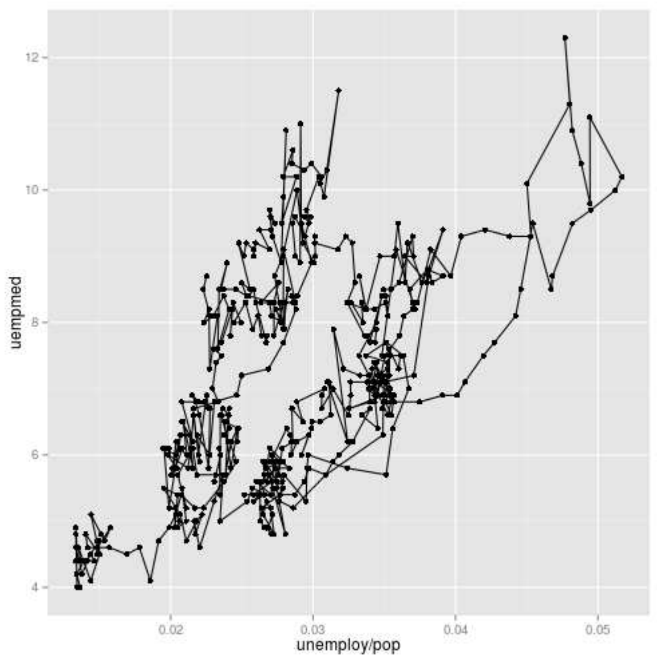
```
qplot(date, uempmed, data = economics, geom = "line")
```



To examine this relationship in greater detail, we would like to draw both time series on the same plot. We could draw a scatterplot of unemployment rate vs. length of unemployment, but then we could no longer see the evolution over time. The solution is to join points adjacent in time with line segments, forming a *path* plot.

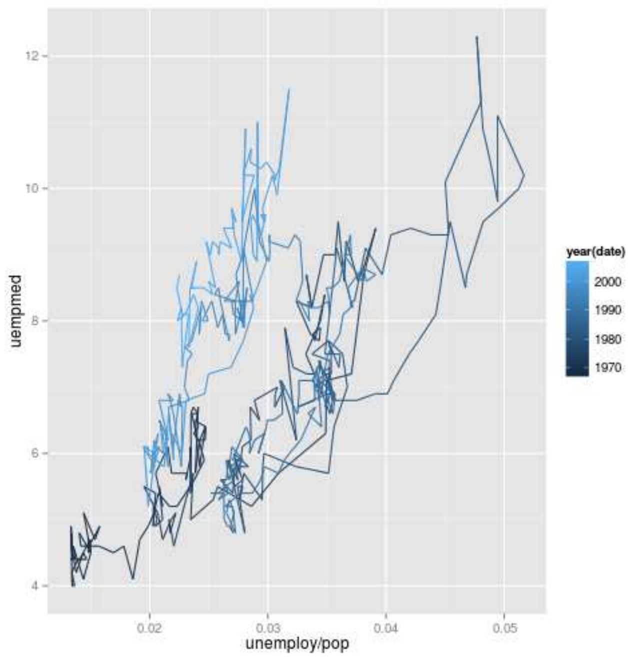
Below we plot unemployment rate vs. length of unemployment and join the individual observations with a path. Because of the many line crossings, the direction in which time flows isn't easy to see in the first plot. In the second plot, we apply the `colour` aesthetic to the line to make it easier to see the direction of time.

```
year <- function(x) as.POSIXlt(x)$year + 1900
qplot(unemploy / pop, uempmed, data = economics,
  geom = c("point", "path"))
```



```
qplot(unemploy / pop, uempmed, data = economics,
  geom = "path", colour = year(date)) + scale_area()
```

```
#> scale_area is deprecated. Use scale_size_area instead.
#>   Note that the behavior of scale_size_area is slightly different:
#>   by default it makes the area proportional to the numeric value. (Deprecated; last used in version 0.9.2)
```



We can see that percent unemployed and length of unemployment are highly correlated, although in recent years the length of unemployment has been increasing relative to the unemployment rate.

With longitudinal data, you often want to display multiple time series on each plot, each series representing one individual. To do this with `qplot()`, you need to map the `group` aesthetic to a variable encoding the group membership of each observation.

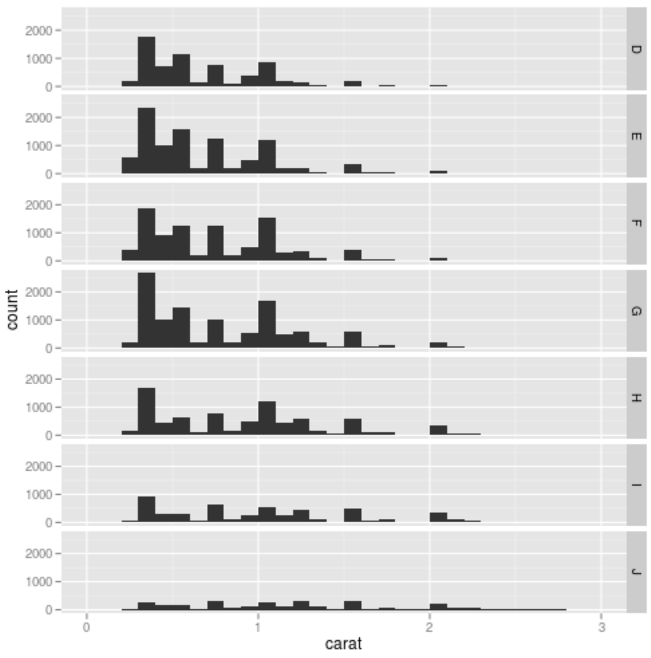
Faceting

We have already discussed using aesthetics (colour and shape) to compare subgroups, drawing all groups on the same plot. Faceting takes an alternative approach: It creates tables of graphics by splitting the data into subsets and displaying the same graph for each subset in an arrangement that facilitates comparison.

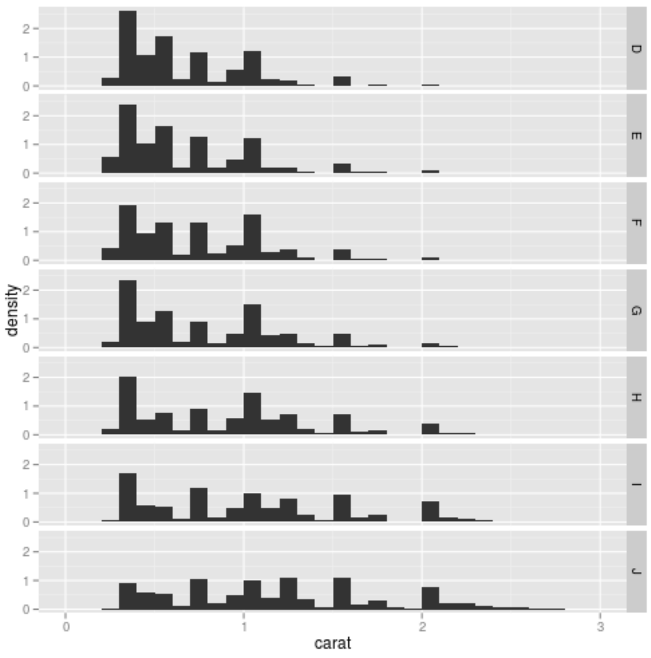
The default faceting method in `qplot()` creates plots arranged on a grid specified by a faceting formula which looks like `row_var ~ col_var`. You can specify as many row and column variables as you like, keeping in mind that using more than two variables will often produce a plot so large that it is difficult to see on screen. To facet on only one of columns or rows, use `.` as a place holder. For example, `row_var ~ .` will create a single column with multiple rows.

The following code illustrates this technique with two plots, sets of histograms showing the distribution of carat conditional on colour. The second set of histograms shows proportions, making it easier to compare distributions regardless of the relative abundance of diamonds of each colour. The `..density..` syntax is new. The y-axis of the histogram does not come from the original data, but from the statistical transformation that counts the number of observations in each bin. Using `..density..` tells ggplot2 to map the density to the y-axis instead of the default use of count.

```
qplot(carat, data = diamonds, geom = "histogram", binwidth = 0.1,
  xlim = c(0, 3)) + facet_grid(color ~ .)
```



```
qplot(carat, ..density.., data = diamonds, geom = "histogram",
  binwidth = 0.1, xlim = c(0, 3)) + facet_grid(color ~ .)
```



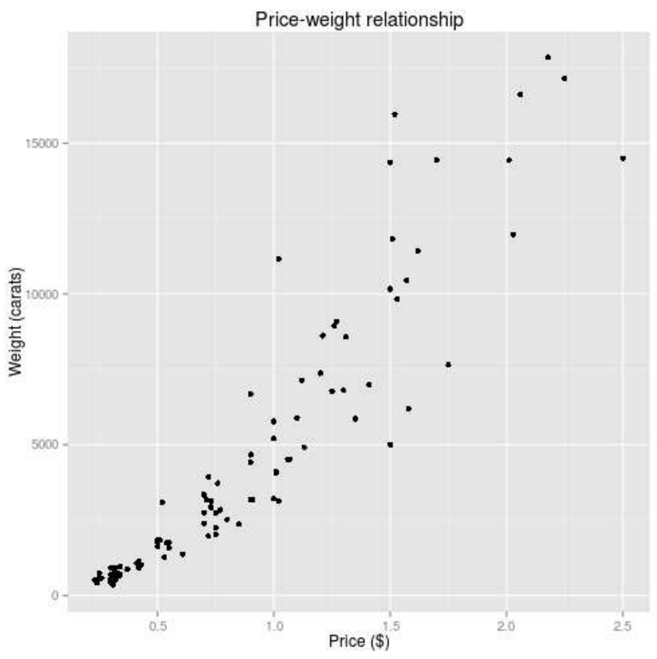
Other options

These are a few other `qplot` options to control the graphic's appearance. These all have the same effect as their `plot` equivalents:

- `xlim`, `ylim`: set limits for the x- and y-axes, each a numeric vector of length two, e.g., `xlim=c(0, 20)` or `ylim=c(-0.9, -0.5)`.
- `log`: a character vector indicating which (if any) axes should be logged. For example, `log="x"` will log the x-axis, `log="xy"` will log both.
- `main`: main title for the plot, centered in large text at the top of the plot. This can be a string (e.g., `main="plot title"`) or an expression (e.g., `main = expression(beta[1] == 1)`). See `?plotmath` for more examples of using mathematical formulae.
- `xlab`, `ylab`: labels for the x- and y-axes. As with the plot title, these can be character strings or mathematical expressions.

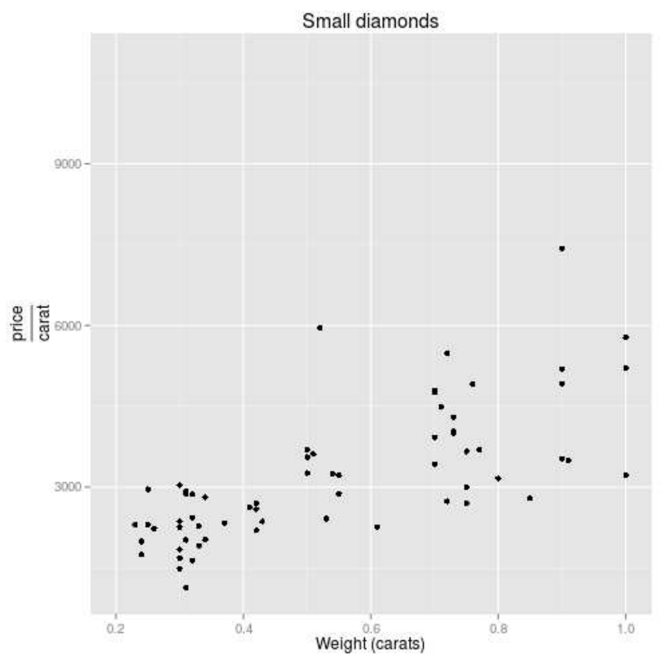
The following examples show the options in action.

```
qplot(
  carat, price, data = dsmall,
  xlab = "Price ($)", ylab = "Weight (carats)",
  main = "Price-weight relationship"
)
```

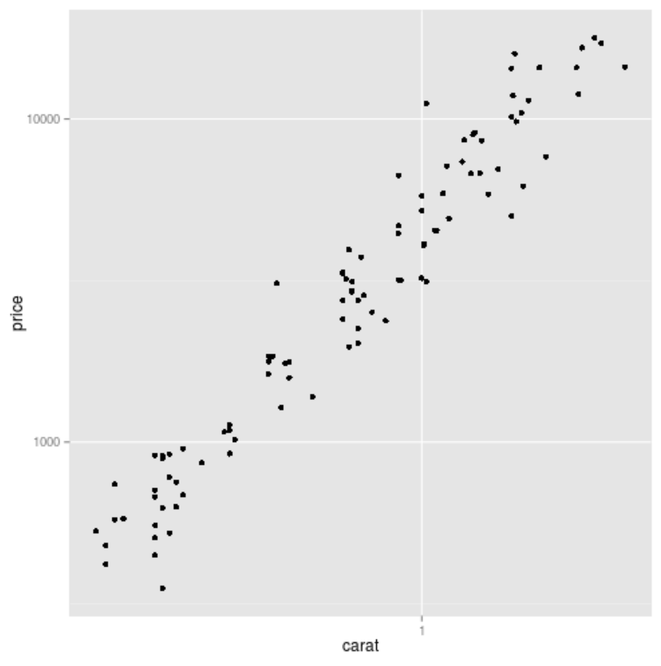


```
qplot(
   carat, price/carat, data = dsmall,
   ylab = expression(frac(price,carat)),
   xlab = "Weight (carats)",
   main="Small diamonds",
   xlim = c(.2,1)
)
```

```
#> Warning: Removed 35 rows containing missing values (geom_point).
```



```
qplot(carat, price, data = dsmall, log = "xy")
```



Differences from plot

There are a few important differences between `plot` and `qplot`:

- `qplot` is not generic: you cannot pass any type of R object to qplot and expect to get some kind of default plot. Note, however, that `autoplot()` is generic, and may provide a starting point for producing visualisations of arbitrary R objects.
- Usually you will supply a variable to the aesthetic attribute you're interested in. This is then scaled and displayed with a legend. If you want to set the value, e.g., to make red points, use `I()`: `colour = I("red")`.
- While you can continue to use the base R aesthetic names (`col`, `pch`, `cex`, etc.), it's a good idea to switch to the more descriptive ggplot2 aesthetic names (`colour`, `shape` and `size`). They're much easier to remember!
- To add further graphic elements to a plot produced in base graphics, you can use `points()`, `lines()` and `text()`. With `ggplot2`, you need to add additional layers to the existing plot.