16

Graphics with ggplot2

16.1 Introduction

As we discussed in Chapter 14, "Graphics Overview," the `ggplot2` package is an implementation of Wilkinson's Grammar of Graphics (hence the "gg" in its name). The last chapter focused on R's traditional graphics functions. Many plots were easy, but other plots were a lot of work compared to Stata. In particular, adding things like legends and confidence intervals were complicated.

The `ggplot2` package makes many of those things easier, as you will now see as we replicate many of the same graphs. The `ggplot2` package has both a shorter `qplot` function (also called `quickplot`) and a more powerful `ggplot` function. We will use both so you can learn the difference and choose whichever you prefer. Although less flexible overall, the built-in `lattice` package is also capable of doing these examples.

While traditional graphics come with R, you will need to install the `ggplot2` package. For details, see Chapter 2, "Installing and Updating R." Once installed, we need to load the package using the library function.

```
> library("ggplot2")

Loading required package: grid
Loading required package: reshape
Loading required package: proto
Loading required package: splines
Loading required package: MASS
Loading required package: RColorBrewer
Loading required package: colorspace
```

Notice that it requires the `grid` package. That is a completely different graphics system than the traditional graphics system. That means that the `par` function we used to set graphics parameters, like fonts, in the last chapter

R.A. Muenchen, J.M. Hilbe, *R for Stata Users*, Statistics and Computing, DOI 10.1007/978-1-4419-1318-0_16,

does not work with ggplot2, nor do any of the base functions that we have covered, including abline, arrows, axis, box, grid, lines, and text.

16.1.1 Overview qplot and ggplot

With the ggplot2 package, you create your graphs by specifying the following elements:

- Aesthetics: The aesthetics map your data to the graph, telling it what role each variable will play. Some variables will map to an axis, and some will determine the color, shape, or size of a point in a scatter plot. Different groups might have differently shaped or colored points. The size or color of a point might reflect the magnitude of a third variable. Other variables might determine how to fill the bars of a bar chart with colors or patterns; so, for example, you can see the number of males and females within each bar.
- Geoms: Short for geometric objects, geoms determine the objects that will represent the data values. Possible geoms include bar, box plot, error bar, histogram, jitter, line, path, point, smooth, and text.
- Statistics: Statistics provide functions for features like adding regression lines to a scatter plot, or dividing a variable up into bins to form a histogram.
- Scales: These match your data to the aesthetic features—for example, in a legend that tells us that triangles represent males and circles represent females.
- Coordinate system: For most plots this is the usual rectangular Cartesian coordinate system. However, for pie charts it is the circular polar coordinate system.
- Facets: These describe how to repeat your plot for each subgroup, perhaps creating a separate scatter plot for males and females. A helpful feature with facets is that they standardize the axes on each plot, making comparisons across groups much easier.

The qplot function tries to simplify graph creation by (a) looking a lot like the traditional plot function and (b) allowing you to skip specifying as many of the items above as possible. As with the plot function, main arguments to the qplot function are the x and y variables. You can identify them with the argument name "x=" or "y=" or you can simply supply them in that order. Unlike the plot function, the qplot function has a data argument. That means you do not have to attach the data frame to use short variable names. (However, to minimize our code, our examples will assume the data is attached.)

Finally, as you would expect, elements are specified by an argument. For example, geom="bar". A major difference between plot and qplot is that qplot will not automatically give you diagnostic plots for a model you have created. So it is much easier to get diagnostic plots using the plot function.

The ggplot function offers a complete implementation of the grammar of graphics. To do so, it gives up any resemblance to the plot function. It *requires* you to specify the data frame, since you can use different data frames in different layers of the graph. (The qplot function cannot.) Its options are specified by additional *functions* rather than the usual arguments. For example, rather than the geom="bar" format of qplot, they follow the form +geom_bar(options). The form is quite consistent, so if you know there is a geom named "smooth," you can readily guess how to specify it in either qplot or ggplot.

The simplicity that qplot offers has another limitation. Since it cannot plot in layers, it occasionally needs help interpreting what you want it to do with legends. For example, you could do a scatter plot for which size=q4. This would cause the points to have five sizes, from small for people who did not like the workshop to large for those who did. The qplot function would generate the legend for you automatically. However, what happens when you just want to specify the size of all points with size=4? It generates a rather useless legend showing one of the points and telling you it represents "4." Whenever you want to tell qplot to *i*nhibit the *i*nterpretation of values, you nest them within the "I()" function: size=I(4). As a mnemonic, think that I()=Inhibit unnecessary legends. The ggplot function does not need the I function since its level of control is fine enough to make your intentions obvious.

See Table 16.1 for a summary of the major differences between qplot and ggplot.

Although the ggplot2 is based on *The Grammar of Graphics*, the package differs in several important ways from the syntax described in that book. It depends on R's ability to transform data, so you can use log(x) or any other function within qplot or ggplot. It also uses R's ability to reshape or aggregate data, so the ggplot2 package does not include its own algebra for these steps. Also, ggplot2 displays axes and legends automatically, so there is no "guide" function.

For a more detailed comparison, see *ggplot2: Elegant Graphics for Data Analysis* by Hadley Wickham [55].

Now let us look at some examples. When possible, each is done using both qplot and ggplot. You can decide which you prefer.

16.1.2 Missing Values

By default, the ggplot2 package will display missing values. That would result in additional bars in bar charts and even entire additional plots when we repeat graphs for each level of a grouping variable. That might be fine in your initial analyses, but you are unlikely to want that in a plot for publication. We will use a version of our data set that has all missing values stripped out with

```
mydata100 <- na.omit(mydata100)
```

Table 16.1. Comparison of the `qplot` and `ggplot` functions.

	The `qplot` function	The `ggplot` function
Goal	Designed to be quick and as standard R as possible.	Designed as a full grammar of graphics system.

Aesthetics	Like most R functions:
	`qplot(x= , y= , fill= ,`
	`color= , shape= ,...)`
		You must specify the mapping between each graphical element, even x- and y-axes, and the variable(s):
		`ggplot(data= ,`
		`aes(x= , y= , fill= , color= ,`
		`shape= ,...) )`

ABline	`...geom="abline",` `intercept=a, slope=b)`	`+geom_abline(` `intercept=a, slope=b)`
Aspect ratio	Leave out for interactive adjustment. `+coord_equal(ratio=height/width)` `+coord_equal()` is square	Leave out for interactive adjustment. `+coord_equal(ratio=height/width)` `+coord_equal()` is square
Axis flipping	`+coord_flip()`	`+coord_flip()`

Axis labels	`...xlab="My Text")`
	Just like plot function.
		`+scale_x_discrete("My Text")`
		`+scale_y_discrete("My Text")`
		`+scale_x_continuous("My Text")`
		`+scale_y_continuous("My Text")`

Axis logarithmic	`+scale_x_log10()`
	`+scale_x_log2()`
	`+scale_x_log()`
		`+scale_x_log10()`
		`+scale_x_log2()`
		`+scale_x_log()`

Bars	`...geom="bar", position="stack"` or dodge.	`+geom_bar(position="stack")` or dodge.
Bar filling	`...posttest, fill=gender)`	`+aes( x=posttest, fill=gender )`
Data	Optional `data=` argument as with most R functions.	You must specify `data=` argument. `ggplot(data=mydata, aes(...`
Facets	`...,facets=gender ~ .)`	`+ facet_grid( gender ~ . )`
Greyscale	`+scale_fill_grey(start=0, end=1)` Change values to control grey.	`+scale_fill_grey(start=0, end=1)` Change values to control grey.
Histogram	`...geom="histogram", binwidth=1)`	`+geom_bar(binwidth=1)`
Density	`...geom="density")`	`+geom_density()`
Jitter	`..position=position_jitter()` Lessen jitter with e.g., `(width=.02)`.	`+geom_jitter(position=position_jitter()` Lessen jitter with e.g., `(width=.02)`.
Legend inhibit	Use `I()` function, e.g., `...geom="point", size=I(4) )`	Precise control makes `I()` function unnecessary.
Line	`...geom="line"`	`+geom_line()`
Line vertical	`...geom="vline", intercept=?)`	`+geom_vline(intercept=?)`
Line horiz.	`...geom="hline", intercept=?)`	`+geom_hline(intercept=?)`
Pie (polar)	`+coord_polar(theta="y")`	`+coord_polar(theta="y")`
Points	`...geom="point")` That is the default for two variables.	`+geom_point(size=2)` There is no default geom for `ggplot`. The default size is 2.
QQ plot	`...stat="qq")`	`+stat_qq=()`
Smooth	`...geom="smooth", method="lm")` Lowess is default method.	`+geom_smooth(method="lm")` Lowess is default method.
Smooth w/o Confidence	`...geom="smooth",` `method="lm", se=FALSE)`	`+geom_smooth(method="lm", se=FALSE)`
Titles	`...main="My Title")` Just like `plot` function.	`+opts(title="My Title" )`

See Section 10.5, "Missing Values" for other ways to address missing values.

16.1.3 Typographic Conventions

Throughout this book we have displayed R's prompt characters only when input was followed by output. The prompt characters helped us discriminate between the two. Each of our function calls will result in a graph, so there is no chance of confusing input with output. Therefore, we will dispense with

the prompt characters for the remainder of this chapter. This will make the code much cleaner to read because our examples of the `ggplot` function often end in a "+" sign. That is something you type. Since R prompts you with "+" at the beginning of a continued line, that looks a bit confusing at first.

16.2 Bar Plots

Let us do a simple bar chart of counts for our workshop variable (Fig. 16.1). Both of the following function calls will do it.

The `qplot` approach to Fig. 16.1 is

```
> attach(mydata100) # Assumed for all qplot examples


> qplot(workshop)
```

The `ggplot` approach to to Fig. 16.1 is

```
> ggplot(mydata100, aes(workshop) ) +
+   geom_bar()
```

Bars are the default geom when you give `qplot` only one factor, so we only need a single argument, workshop.

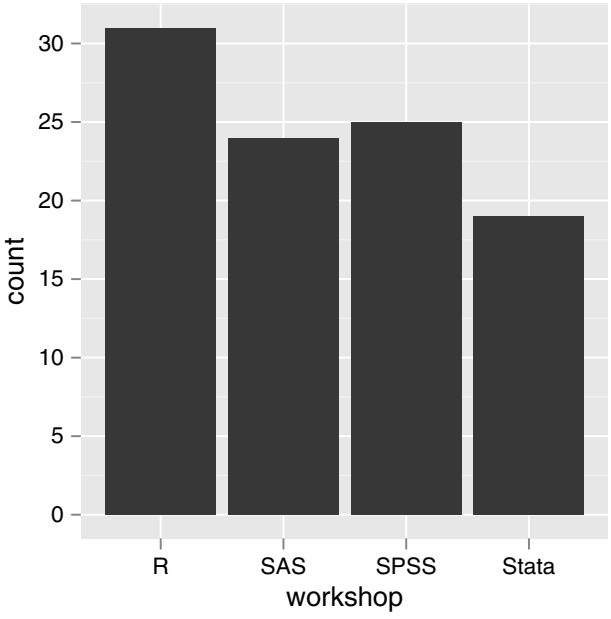

Fig. 16.1. A bar plot of workshop attendance.

The ggplot function call above requires three arguments:

1. Unlike most other R functions, it requires that you specify the data frame. As we will see later, that is because ggplot can plot multiple layers, and each layer can use a different data frame.
2. The aes function defines the *aes*thetic role that workshop will play. It maps workshop to the x-axis. We could have named the argument as in aes(x=workshop). The first two parameters to the aes function are x and y, in that order. To simplify the code, we will not bother listing their names.
3. The geom_bar function tells it that the geometric object, or geom, needed is a bar. Therefore, a bar chart will result. This function call is tied to the first one through the "+" sign.

We did that the same plot using traditional graphics bar plot function, but that required us to summarize the data using table(workshop). The ggplot2 package is more like Stata in this regard; it does that type of summarization for you.

If we want to change to a horizontal bar chart (Fig. 16.2), all we need to do is flip the coordinates. In the following examples, it is clear that we simply added the cord_flip function to the end of both qplot and ggplot. There is no argument to qplot like coord="flip".

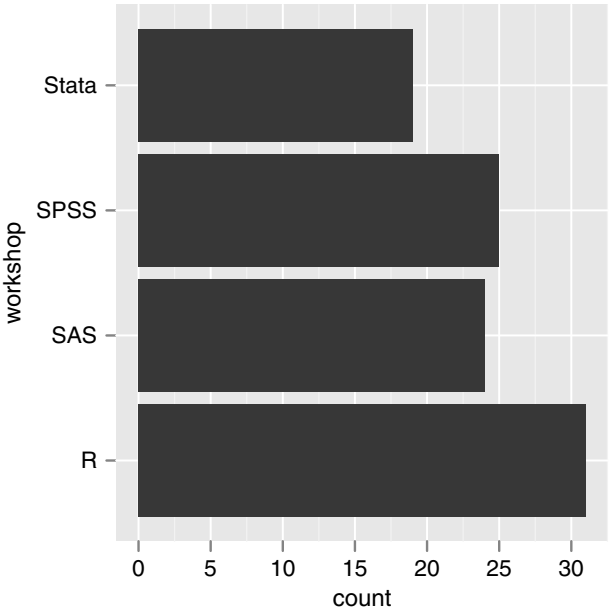

Fig. 16.2. A horizontal bar plot demonstrating the impact of the coord_flip function.

This brings up an interesting point. Both methods create the exact same graphics object. Even if there is a `qplot` equivalent, you can always add a `ggplot` function call to a `qplot` function call.

The `qplot` approach to Fig. 16.2 is

```
qplot(workshop) + coord_flip()
```

The `ggplot` approach to Fig. 16.2 is

```
ggplot(mydata100, aes(workshop) ) +
  geom_bar() + coord_flip()
```

You can create the usual types of grouped bar plots. Let us start with a simple stacked one (Fig. 16.3). You can use either function below. They contain two new arguments. Although we are requesting only a single bar, we must still supply a variable for the x-axis. The function call `factor("")` provides the variable we need, and it is simply an unnamed factor whose value is empty. We use the `factor` function to keep it from labeling the x-axis from 0 to 1, which it would do if the variable were continuous. The `fill=workshop` aesthetic argument tells the function to fill the bars with the number of students who took each workshop.

With `qplot`, we are clearing labels on the x-axis with `xlab=""`. Otherwise, the word "factor" would occur there from our `factor("")` statement.

The equivalent way to label `ggplot` is to use the `scale_x_discrete` function, also providing an empty label for the x-axis. Finally, the

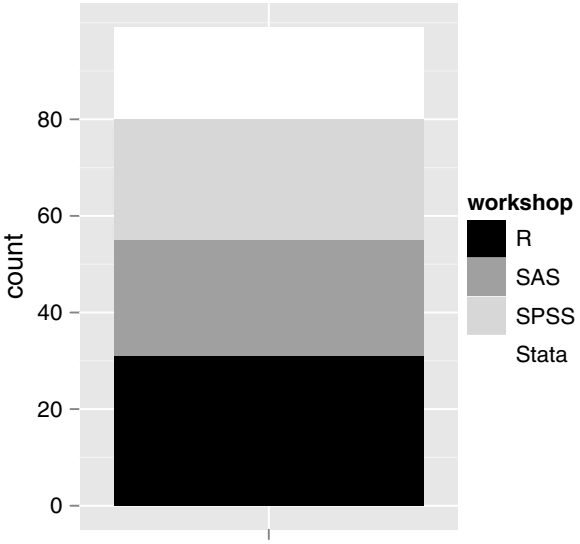

Fig. 16.3. A stacked bar plot of workshop attendance.

`scale_fill_grey` function tells each function to use shades of grey. You can leave this out, of course, and both functions will choose the same nice color scheme. The start and end values tell the function to go all the way to black and white, respectively. If you use just `scale_fill_grey()`, it will use four shades of grey.

The `qplot` approach to Fig. 16.3 is

```
qplot(factor(""), fill=workshop,
  geom="bar", xlab="") +
  scale_fill_grey(start=0, end=1)
```

The `ggplot` approach to Fig. 16.3 is

```
ggplot(mydata100,
  aes(factor(""), fill=workshop) ) +
  geom_bar() +
  scale_x_discrete("") +
  scale_fill_grey(start=0, end=1)
```

16.2.1 Pie Charts

One interesting aspect to the grammar of graphics concept is that a pie chart (Fig. 16.4) is just a single stacked bar chart (Fig. 16.3), drawn in polar coordinates. So we can use the same function calls that we used for the bar chart in the previous section, but convert to polar afterward using the `coord_polar` function.

This is a plot that only `ggplot` can do correctly. The `geom_bar(width=1)` function call tells it to put the slices right next to each other. If you included that on a standard bar chart, it would also put the bars right next to each other.

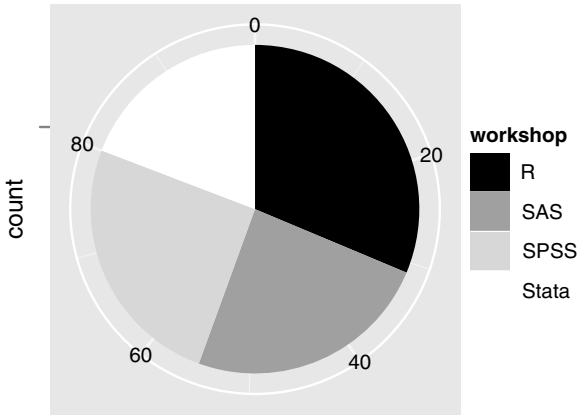

Fig. 16.4. A pie chart of workshop attendance.

```
ggplot(mydata100,
  aes( factor(""), fill=workshop ) ) +
  geom_bar( width=1 ) +
  scale_x_discrete("") +
  coord_polar(theta="y") +
  scale_fill_grey(start=0, end=1)
```

That is a lot of code for a simple pie chart! In the previous chapter, we created this graph with a simple

```
pie( table(workshop) )
```

So traditional graphics are the better approach in some cases. However, as we will see in the coming sections, the `ggplot2` package is the easiest to use for most things.

16.2.2 Bar Charts for Groups

Let us now look at repeating bar charts for levels of a factor, like gender. This requires having factors named for both the `x` argument and the `fill` argument. By default, the `position` argument stacks the `fill` groups—in this case, the workshops. That graph is displayed in the upper left frame of Fig. 16.5.

The `qplot` approach to Fig. 16.5, upper left is

```
qplot(gender, geom="bar",
  fill=workshop, position="stack") +
  scale_fill_grey(start=0, end=1)
```

The `ggplot` apprach to Fig. 16.5, upper left is

```
ggplot(mydata100, aes(gender, fill=workshop) ) +
  geom_bar(position="stack") +
  scale_fill_grey( start=0, end=1 )
```

Changing either of the above examples to:

```
position="fill"
```

makes every bar fill the y-axis, displaying the proportion in each group rather than the number. That type of graph is called a *spine plot* and it is displayed in the upper right of Fig. 16.5.

Finally, if you set

`position="dodge"` the filled segments appear beside one another, "dodging" each other. That takes more room on the x-axis, so it appears across the whole bottom row of Fig. 16.5.

We will discuss how to convey similar information using multiframe plots in Section 16.15, "Multiple Plots on a Page."

Fig. 16.5. A multiframe plot showing the impact of the various position settings.

16.3 Plots by Group or Level

One of the nicest features of the `ggplot2` package is its ability to easily plot groups within a single plot (Fig. 16.6). To fully appreciate all of the work it is doing for us, let us first consider how to do this with traditional graphics functions.

1. We would set up a multiframe plot, say for males and females.
2. Then we might create a bar chart on workshop, selecting `which( gender=="Female" )`.
3. Then we would repeat the step above, selecting the males.
4. We probably want to standardize the axes to better enable comparisons and do the plots again.
5. We would add a legend, making sure to manually match any color or symbol differences across the plots.
6. Finally, we would turn off the multiframe plot settings to get back to one plot-per-page.

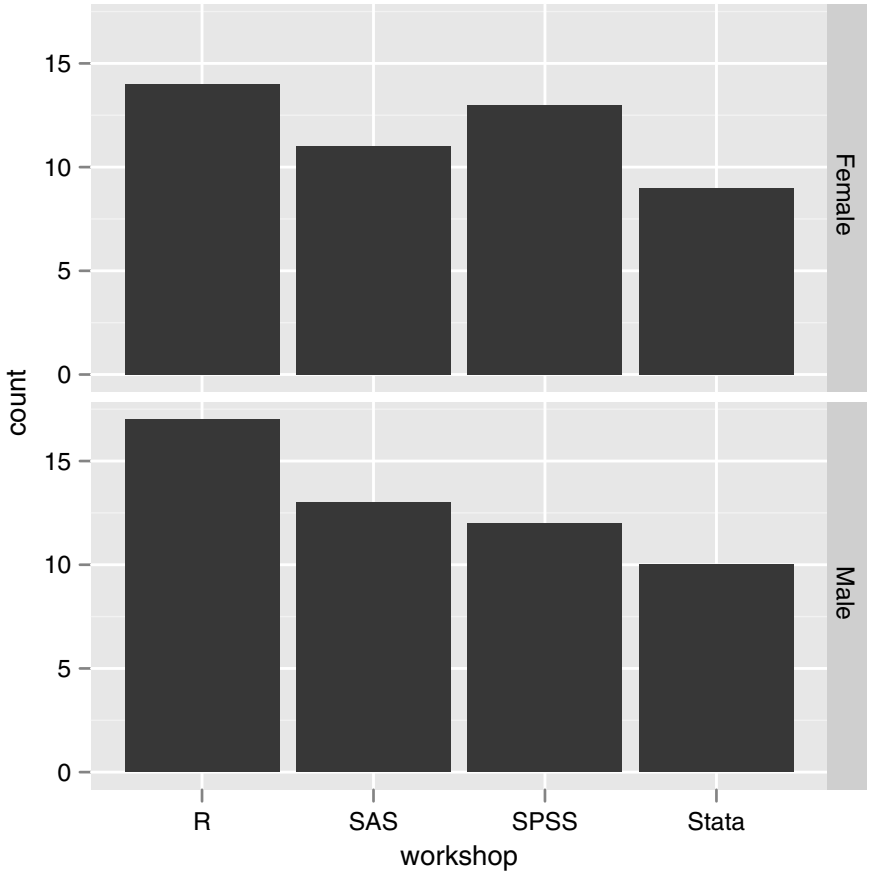

Fig. 16.6. A bar plot of workshop attendance with facets for the genders.

Thank goodness the `ggplot2` package can perform the equivalent of those tedious steps using either of the following simple function calls:

The `qplot` approach to Fig. 16.6 is

```
qplot(workshop, facets=gender ~ . )
```

The `ggplot` approach to Fig. 16.6 is

```
ggplot(mydata100, aes(workshop) ) +
  geom_bar() + facet_grid( gender ~ . )
```

The new feature is the `facets` argument in `qplot` and the `facet_grid` function in `ggplot`. The formula it uses is in the form "`rows~columns`". In this case, we have "`gender~.`" so we will get rows of plots for each gender and no columns. The "`.`" represents "1" row or column. If we instead did "`.~gender`", we would have one row and two columns of plots side-by-side.

You can extend this idea with the various rules for formulas described in Section 5.6.2, "Controlling Functions with Formulas." Given the constraints of space, the most you are likely to find useful is the addition of one more variable, such as

```
facets=workshop ~ gender
```

In our current example, that leaves us nothing to plot, but we will look at a scatter plot example of that later.

16.4 Presummarized Data

We mentioned earlier that the ggplot2 package assumed that your data needed summarizing, which is the opposite of some traditional R graphics functions. However, what if the data are already summarized? The qplot function makes it quite easy to deal with, as you can see in the program below. We simply use the factor function to provide the x argument and the c function to provide the data for the y argument. Since we are providing both x and y arguments, the qplot function will provide a default point geom, so we override that with geom="bar". The xlab and ylab arguments *lab*el the axes, which it would otherwise label with the factor and c functions themselves.

The qplot approach to Fig. 16.7 is

```
qplot( factor(c(1,2)), c(40, 60), geom="bar",
  xlab="myGroup", ylab="myMeasure")
```

The ggplot approach to this type of plot is somewhat different because it requires that the data be in a data frame. I find it much easier to create a temporary data frame containing the summary data. Trying to nest a data frame creation within the ggplot function will work, but you end up with so many parentheses that it can be a challenge getting it to work. The example program at the end of this chapter contains that example as well.

The following is the more complicated ggplot approach to Fig. 16.7. We are displaying R's prompts here to differentiate input from output.

```
> myTemp <- data.frame(
+   myGroup=factor( c(1,2) ),
+   myMeasure=c(40, 60)
+ )

> myTemp

  myGroup myMeasure
1       1        40
2       2        60
```

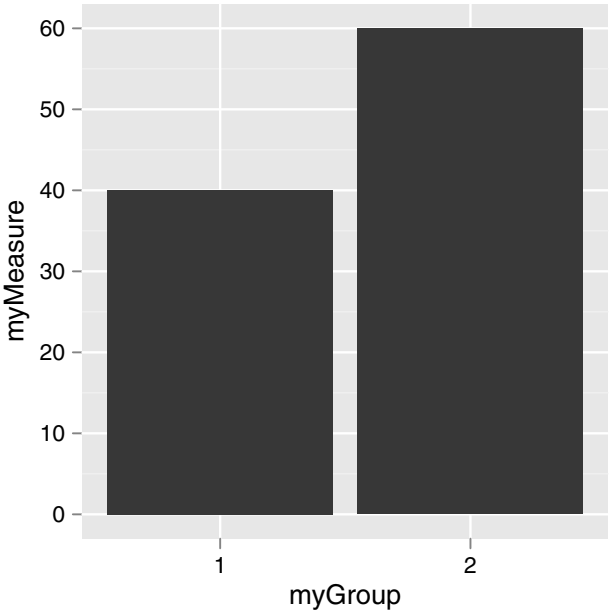

Fig. 16.7. A bar plot of presummarized data.

```
> ggplot(data=myTemp, aes(myX, myY) ) +
+   geom_bar()

> rm(myTemp) #Cleaning up.
```

16.5 Dot Charts

Dot charts are bar charts reduced to just points on lines, so you can take any of the above bar chart examples and turn them into dot charts (Fig. 16.8).

Dot charts are particularly good at packing in a lot of information on a single plot, so let us look at the counts for the attendance in each workshop, for both males and females. This example demonstrates how very different `qplot` and `ggplot` can be. It also shows how flexible `ggplot` is and that it is sometimes much easier to understand than `qplot`.

First, let us look at how `qplot` does it. The variable workshop is in the `x` position, so this is the same as saying `x=workshop`. If you look at the plot, workshop is on the y-axis. However, `qplot` requires an `x` variable, so we cannot simply say `y=workshop` and not specify an `x` variable. Next, it specifies `geom="point"` and sets the size of the points to `I(4)`, which is much larger than in a standard scatter plot. Remember that the `I()` function around the

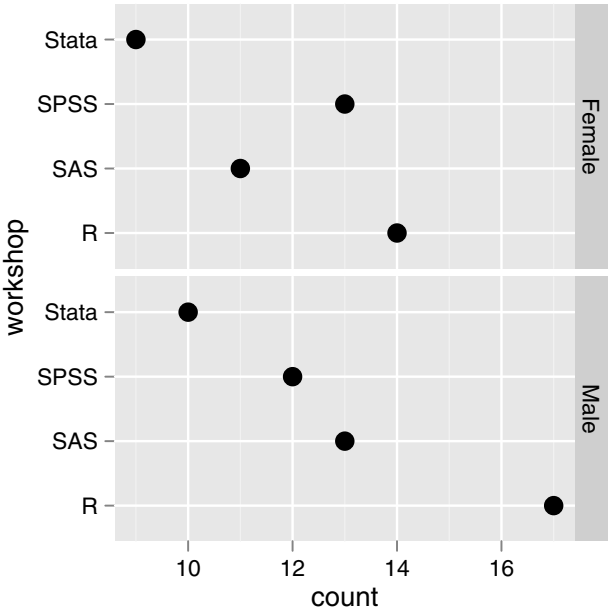

Fig. 16.8. A dot chart of workshop attendance with facets for the genders.

4 *i*nhibits *i*nterpretation, which in this case means that it stops `qplot` from displaying a legend showing which point size represents a "4." In this example, that is useless information. You can try various size values to see how it looks. The `stat="bin"` argument tells it to combine all of the values that it finds for each level of workshop as a histogram might do. So it ends up counting the number of observations in each combination of workshop and gender. The `facets` argument tells it to create a row for each gender. The `coord_flip` function rotates it in the direction we desire.

The `qplot` approach to Fig. 16.8 is

```
qplot(workshop, geom="point", size=I(4),
  stat="bin", facets=gender~.) +
 coord_flip()
```

Now let us see how `ggplot` does the same plot. The `aes` function supplies the x-axis variable and the y-axis variable uses the special "`..count..`" computed variable. That variable is also used by `qplot`, but it is the default `y` variable. The `geom_point` function adds points, bins them, and sets their size. The `coord_flip` function then reverses the axes. Finally, the `facet_grid` function specifies the same formula used earlier in `qplot`. Notice here that we did not need the `I()` function, as `ggplot` "knows" that the legend is not needed. If we were adjusting the point sizes based on a third variable, we would have

to specify the variable as an additional aesthetic. The syntax to `ggplot` is verbose, but more precise.

```
ggplot(mydata100,
   aes(workshop, ..count..) ) +
   geom_point(stat="bin", size=4) + coord_flip()+
   facet_grid( gender~. )
```

16.6 Adding Titles and Labels

Sprucing up your graphs with titles and labels is easy to do (Fig. 16.9). The `qplot` function adds them exactly like the traditional graphics functions do. You supply the main title with the `main` argument, and the x and y labels with `xlab` and `ylab`, respectively. There is no subtitle argument. As with all labels in R, the characters "\n" causes it to go to a new line, so "\nWorkshops" below will put just the word "Workshops" at the beginning of a new line.

The `qplot` approach to Fig.16.9 is

```
qplot(workshop, geom="bar",
  main="Workshop Attendance",
  xlab="Statistics Package \nWorkshops")
```

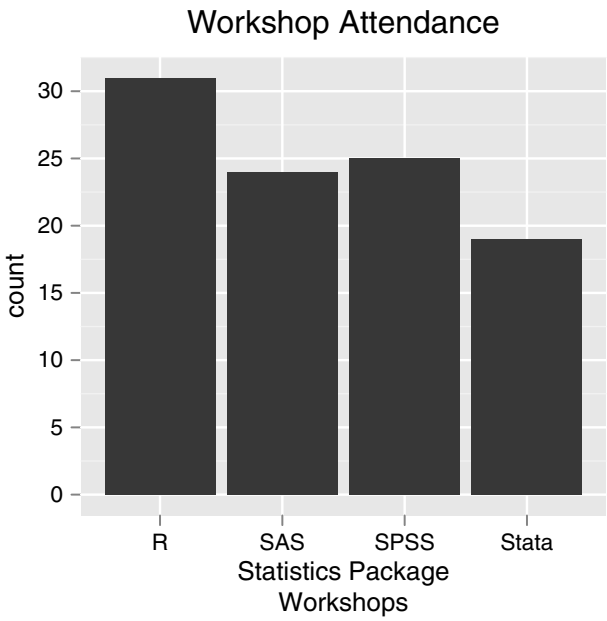

Fig. 16.9. A bar plot demonstrating titles and x-axis labels.

The ggplot approach to Fig.16.9 is

```
ggplot(mydata100, aes(workshop, ..count..) ) +
  geom_bar() +
  opts( title="Workshop Attendance" ) +
  scale_x_discrete("Statistics Package \nWorkshops")
```

Adding titles and labels in ggplot is slightly more verbose. The opts function sets various *opt*ions, one of which is title. The axis labels are attributes of the axes themselves. They are controlled by the functions,
scale_x_discrete, scale_y_discrete, and for continuous axes, they are controlled by the functions,
scale_x_continuous, scale_y_continuous, which are clearly named according to their function. We find it odd that you use different functions for labeling axes if they are discrete or continuous, but it is one of the trade-offs you make when getting all of the flexibility that ggplot offers.

16.7 Histograms and Density Plots

Many statistical methods make assumptions about the distribution of your data, or at least of your model residuals. Histograms and density plots are two effective plots to help you assess the distributions of your data.

16.7.1 Histograms

As long as you have 30 or more observations, histograms (Fig. 16.10) are a good way to examine continuous variables. You can use either of the following examples to create one. In qplot, the histogram is the default geom for continuous data, making it particulary easy to perform.

The qplot approach to Fig. 16.10 is

```
qplot(posttest)
```

The ggplot approach to Fig. 16.10 is

```
ggplot(mydata100, aes(posttest) ) +
  geom_histogram()
```

Both functions will print the number of bins it uses by default (30) to the R console (not shown). If you narrow the width of the bins, you will get more bars, showing more structure in the data (Fig. 16.11). If you prefer qplot, simply add the binwidth argument. If you prefer ggplot, add the geom_bar function with its binwidth argument. Smaller numbers result in more bars.

The qplot approach to Fig. 16.11 is

```
qplot(posttest, geom="histogram", binwidth=0.5)
```

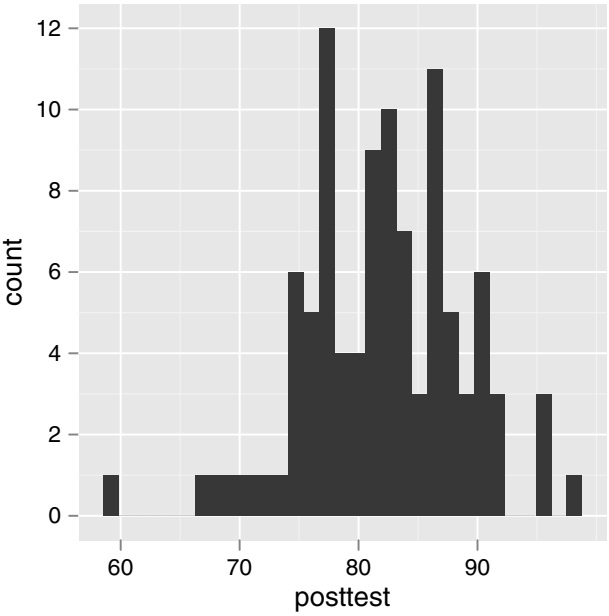

Fig. 16.10. A histogram of posttest.

The `ggplot` approach to Fig. 16.11 is

```
ggplot(mydata100, aes(posttest) ) +
  geom_bar( binwidth=0.5 )
```

16.7.2 Density Plots

If you prefer to see a density curve, just change the `geom` argument or function to `density` (Fig. 16.12).

The `qplot` approach to Fig. 16.12 is

```
qplot(posttest, geom="density" )
```

The `ggplot` approach to Fig. 16.12 is

```
ggplot(mydata100, aes(posttest) ) +
  geom_density()
```

16.7.3 Histograms with Density Overlaid

Overlaying the density on the histogram, as in Fig. 16.13, is only slightly more complicated. The variable that `qplot` or `ggplot` computes in the background

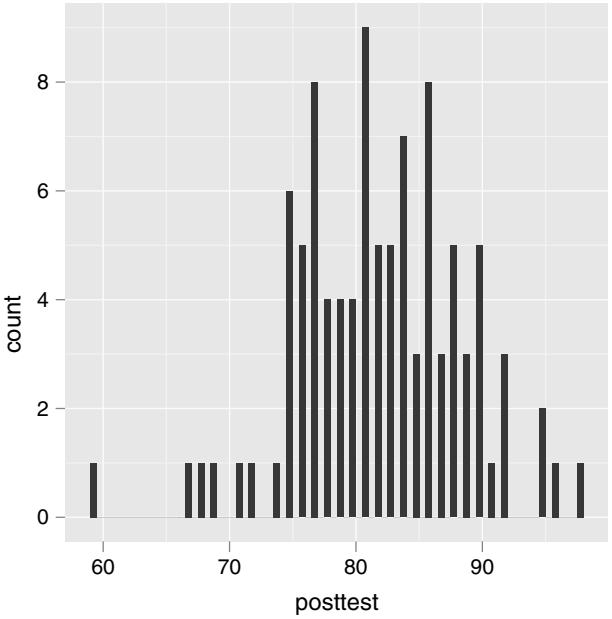

Fig. 16.11. A histogram of posttest with smaller bin widths.

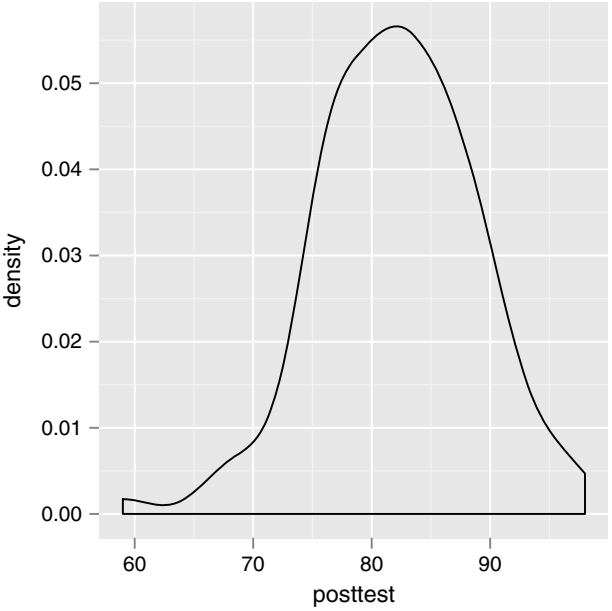

Fig. 16.12. A density plot of posttest.

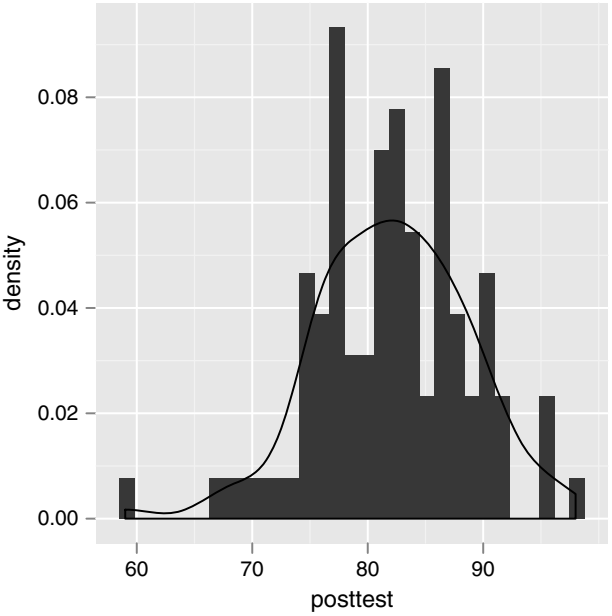

Fig. 16.13. A density plot overlaid on a histogram.

for the y-axis is named "`..density..`". To ask for both a histogram and the density, you must explicitly list `..density..` as the `y` variable. Then for `qplot`, you provide both `histogram` and `density` to the `geom` argument by combining them into a character vector using the `c` function.

For `ggplot`, you simply call both functions.

The `qplot` approach to Fig. 16.13 is

```
qplot(posttest, ..density..,
  geom=c( "histogram", "density" ) )
```

The `ggplot` approach to Fig. 16.13 is

```
ggplot(mydata100, aes(posttest, ..density..)) +
  geom_histogram() + geom_density()
```

16.7.4 Histograms for Groups, Stacked

What if we want to compare the histograms for males and females (Fig. 16.14)? Using base graphics, we had to set up a multiframe plot and learn how to control breakpoints for the bars so that they would be comparable. Using `ggplot2`, the facet feature makes the job trivial.

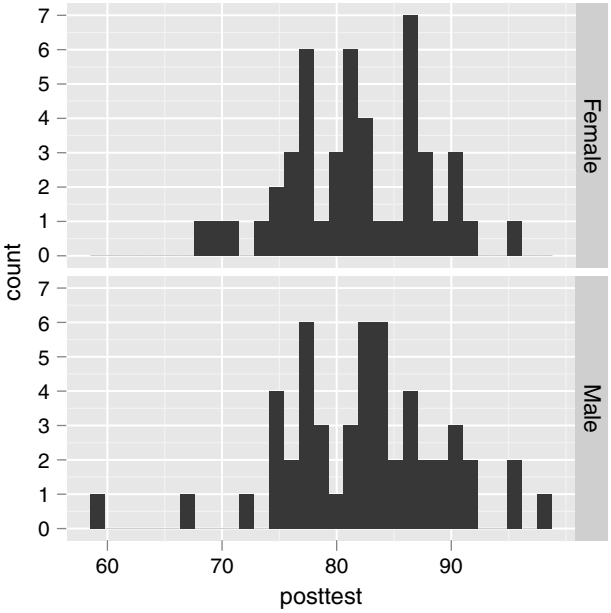

Fig. 16.14. Histograms of posttest with facets for the genders.

The `qplot` approach to Fig. 16.14 is

```
qplot(posttest, facets=gender~.)
```

The `ggplot` approach to Fig. 16.14is

```
ggplot(mydata100, aes(posttest) ) +
  geom_histogram() + facet_grid( gender ~ . )
```

16.7.5 Histograms for Groups, Overlaid

We can also compare males and females by filling the bars by gender as in Fig. 16.15. As earlier, if you leave off the `scale_fill_grey` function, the bars will come out in two colors rather than black and white.

The `qplot` approach to Fig. 16.15 is

```
qplot( posttest, fill=gender ) +
  scale_fill_grey(start = 0, end = 1)
```

The `ggplot` approach to Fig. 16.15 is

```
ggplot(mydata100, aes(posttest, fill=gender) ) +
  geom_bar() + scale_fill_grey( start=0, end=1 )
```

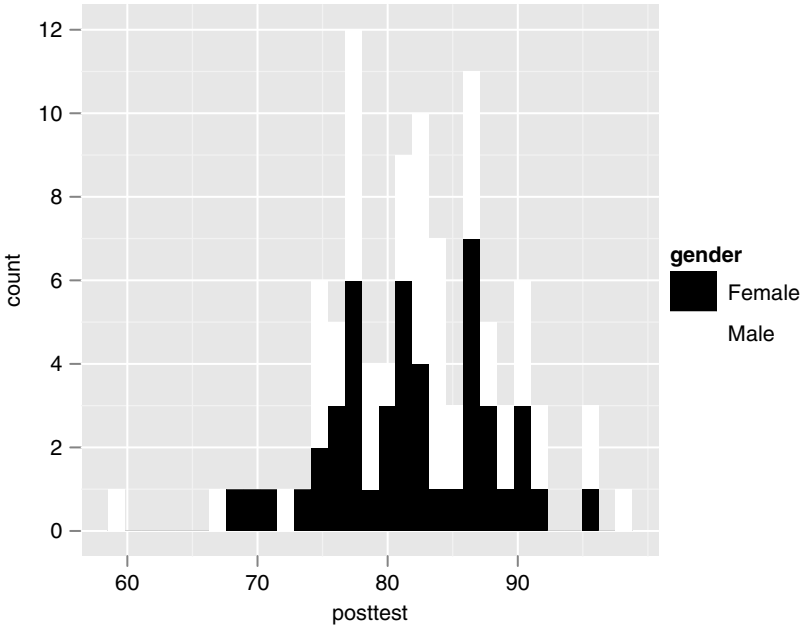

Fig. 16.15. A histogram with bars filled by gender.

16.8 Normal QQ Plots

We defined what a QQ plot is the previous chapter on traditional graphics. Creating them in the `ggplot2` package is straightforward (Fig. 16.16). If you prefer the `qplot` function, supply the `stat="qq"` argument. In `ggplot`, the similar `stat_qq` function will do the trick.

The `qplot` approach to Fig. 16.16 is

```
qplot( posttest, stat="qq" )
```

The `ggplot` approach to Fig. 16.16 is

```
ggplot( mydata100, aes(posttest) ) +
  stat_qq()
```

16.9 Strip Plots

Strip plots are scatter plots of single continuous variables, or a continuous variable displayed at each level of a factor like workshop. As with the single stacked bar chart, the case of a single strip plot still requires a variable on the x-axis (Fig. 16.17). As you see below, `factor("")` will suffice. The variable to actually plot is the `y` argument. Reversing the x and y variables will turn the

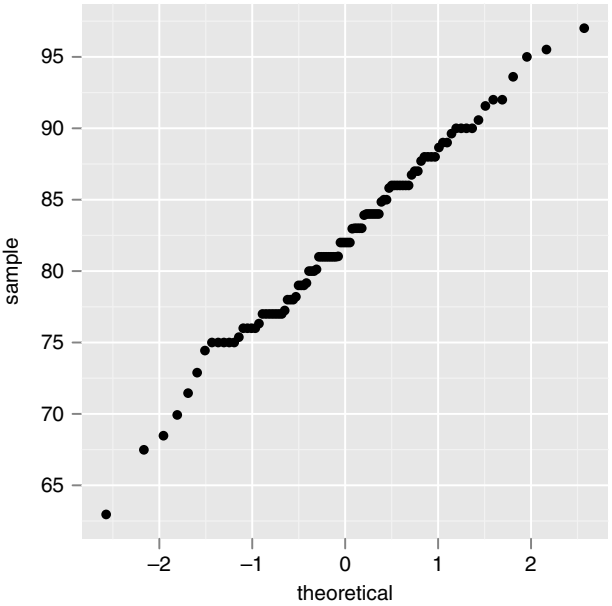

Fig. 16.16. A normal quantile–quantile plot of posttest.

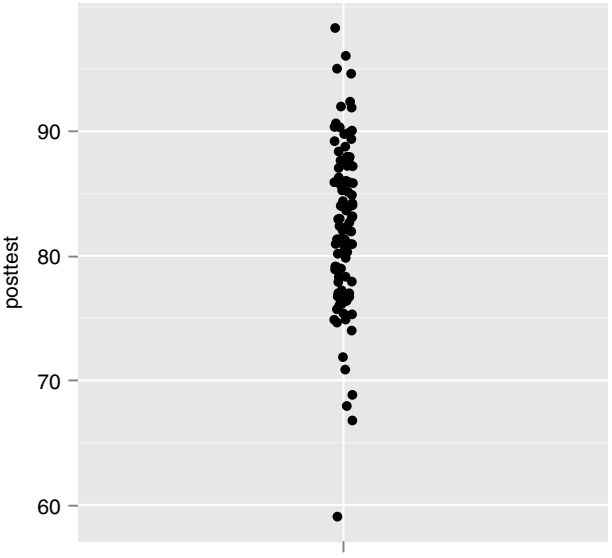

Fig. 16.17. A strip chart done using the `jitter` geom.

plot on its side, the default way the traditional graphics function, `stripchart`, does it. We prefer the vertical approach, as it matches the style of box plots and error bar plots when you use them to compare groups. The `geom="jitter"` adds some noise to separate points that would otherwise obscure other points by plotting on top of them. Finally, the `xlab=""` and `scale_x_discrete("")` labels erase what would have been a meaningless label about `factor("")` for `qplot` and `ggplot`, respectively.

This `qplot` approach does a strip plot with wider jitter than Fig 16.17:

```
qplot( factor(""), posttest, geom="jitter", xlab="")
```

This `ggplot` approach does a strip plot with wider jitter than Fig. 16.17:

```
ggplot(mydata100, aes(factor(""), posttest) ) +
  geom_jitter() +
  scale_x_discrete("")
```

The above two examples use an amount of jitter that is best for large data sets. For smaller data sets, it is best to limit the amount of jitter to separate the groups into clear strips of points. Unfortunately, this complicates the syntax.

The `qplot` function controls jitter width with the `position` argument, setting `position_jitter` with `width=scalefactor`.

The `ggplot` approach places that same parameter within its `geom_jitter` function call.

The `qplot` approach to Fig. 16.17 is

```
qplot( factor(""), posttest, data = mydata100, xlab="",
  position=position_jitter(width=.02))
```

The `ggplot` approach to Fig. 16.17 is

```
ggplot(mydata100, aes(factor(""), posttest) ) +
 geom_jitter(position=position_jitter(width=.02)) +
 scale_x_discrete("")
```

Placing a factor like workshop on the x-axis will result in a strip chart for each level of the factor (Fig. 16.18).

This `qplot` approach does a grouped strip plot with wider jitter than Fig 16.18, but its code is simpler:

```
> qplot(workshop, posttest, geom="jitter")
```

This `ggplot` approach does a grouped strip plot with wider jitter than Fig 16.18, but with simpler code:

```
> ggplot(mydata100, aes(workshop, posttest ) ) +
+   geom_jitter()
```

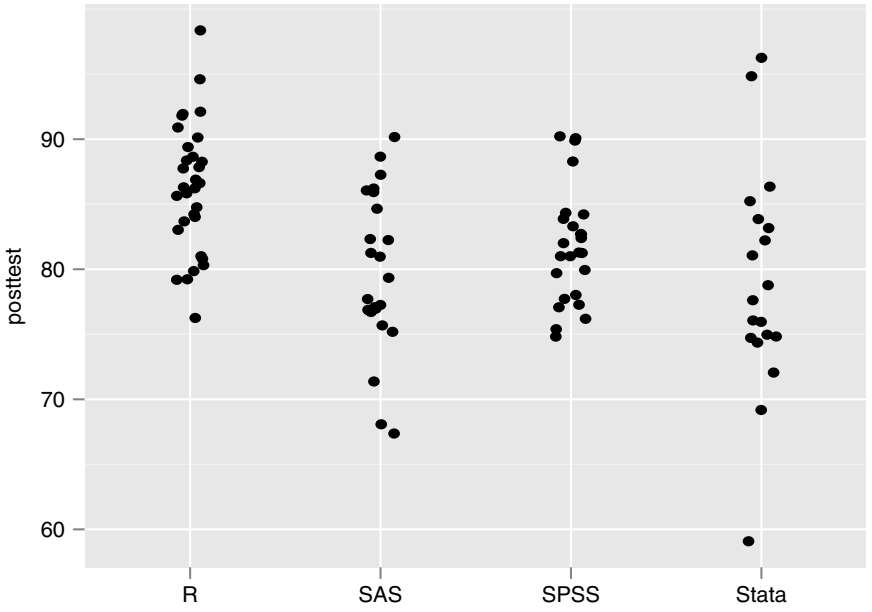

Fig. 16.18. A strip chart with facets for the workshops.

Limiting the amount of jitter for a grouped strip plot uses exactly the same parameters we used for a single strip plot.

The `qplot` approach to Fig. 16.18 is

```
qplot(workshop, posttest, data = mydata100, xlab="",
  position=position_jitter(width=.08))
```

The `ggplot` approach to Fig. 16.18 is

```
ggplot(mydata100, aes(workshop, posttest) ) +
 geom_jitter(position=position_jitter(width=.08)) +
 scale_x_discrete("")
```

16.10 Scatter Plots and Line Plots

The simplest scatter plot hardly takes any effort in `qplot`. Just list `x` and `y` variables in that order. You could add the `geom="point"` argument, but it is the default when you list two variables.

The `ggplot` function is slightly more complicated. Since it has no default geometric object to display, we must specify `geom_point()`.

The `qplot` approach to Fig. 16.19, upper left, is

```
qplot(pretest, posttest)
```

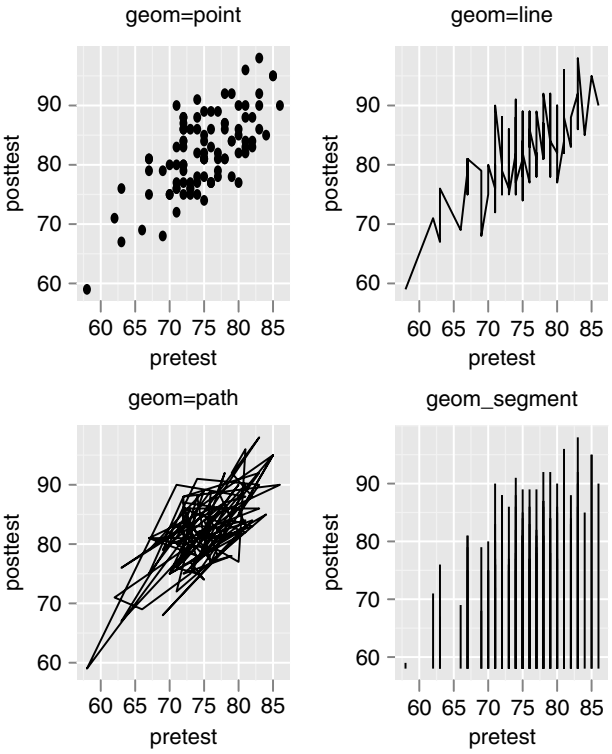

Fig. 16.19. A multiframe plot demonstrating various styles of scatter plots and line plots. The top two and the bottom left show different geoms. The bottom right is done a very different way, by drawing line segments from each point to the x-axis.

The `ggplot` approach to Fig. 16.19, upper left, is

```
ggplot(mydata100, aes(pretest, posttest) ) +
  geom_point()
```

We can connect the points using the line geom, as you see below. However, the result is different from what you get in traditional R graphics. The line connects the points in the order that they appear on the x-axis. That almost makes our data appear as a time series, when it is not.

The `qplot` approach to Fig. 16.19, upper right, is

```
qplot( pretest, posttest, geom="line")
```

The `qplot` approach to Fig. 16.19, upper right, is

```
ggplot(mydata100, aes(pretest, posttest) ) +
  geom_line()
```

Although the line geom ignored the order of the points in the data frame, the path geom will connect them in that order. You can see the result in the lower left quadrant of Fig. 16.19. The order of the points in our data set has no meaning, so it is just a mess!

The qplot approach to Fig. 16.19, lower left, is

```
qplot( pretest, posttest, geom="path")
```

The ggplot approach to Fig. 16.19, lower left, is

```
ggplot(mydata100, aes(pretest, posttest) ) +
  geom_path()
```

Now let us run a vertical line to each point. When we did that using traditional graphics, it was a very minor variation. In ggplot2, it is quite different but an interesting example. It is a plot that is much more clear using ggplot, so we will skip qplot for this one.

In the ggplot code below, the first line is the same as the above examples. Where it gets interesting is the geom_segment function. It has its own aes function, repeating the x and y arguments, but in this case, they are the beginning points for drawing line segments! It also has the arguments xend and yend, which tell it where to end the line segments. This may look overly complicated compared to the simple "type=h" argument from the plot function, but you could use this approach to draw all kinds of line segments. You could easily draw them coming from the top or either side, or even among sets of points. The "type=h" approach is a one trick pony. With that approach, adding features to a function leads to a very large number of options, and the developer is still unlikely to think of all of the interesting variations in advance.

The following is the code, and the resulting plot is in the lower right panel of Fig. 16.19.

```
ggplot(mydata100, aes(pretest, posttest) ) +
  geom_segment( aes(  pretest, posttest,
                  xend=pretest, yend=58) )
```

16.10.1 Scatter Plots with Jitter

We discussed the benefits of jitter in the previous chapter. To get a nonjittered plot of q1 and q4, we will just use qplot (Fig. 16.20, left).

```
qplot(q1,q4)
```

To add jitter, below are both the qplot and gglot approaches (Fig. 16.20, right). Note that the geom="point" argument is the default in qplot when two variables are used. Since that default is not shown, the fact that the position argument applies to it is not obvious. That relationship is clearer in the ggplot

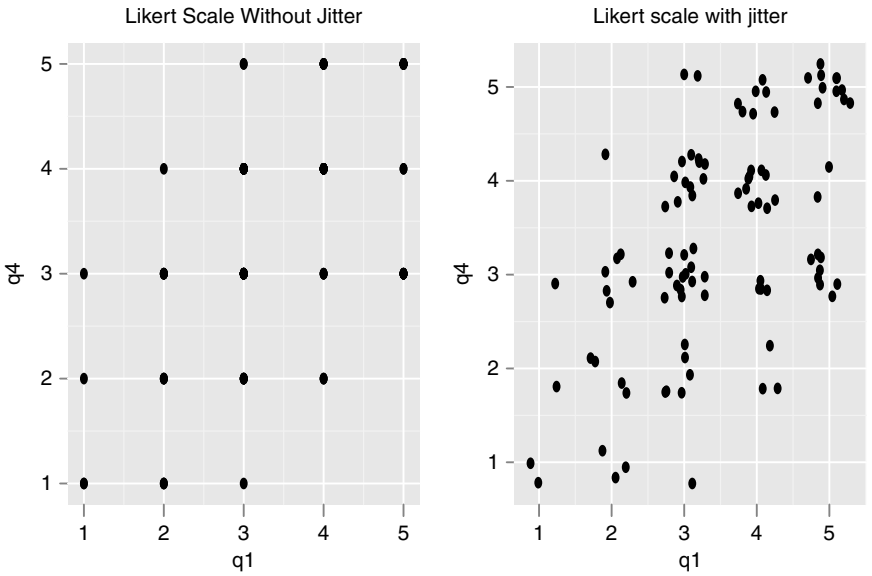

Fig. 16.20. A multiframe plot showing the impact of jitter on five-point Likert–scale data. The plot on the left is not jittered, so many points are obscured. The plot on the right is jittered, randomly moving points out from behind one another.

code, where the `position` argument is clearly part of the `geom_point` function. You can try various amounts of jitter to see which provides the best view of your data.

The `qplot` approach to Fig. 16.20, right, is

```
qplot(q1, q4, position=position_jitter(width=.3,height=.3))
```

The `ggplot` approach to Fig. 16.20, right, is

```
ggplot(mydata100, aes(x=q1, y=q2) ) +
 geom_point(position=position_jitter(width=.3,height=.3))
```

16.10.2 Scatter Plots for Large Data Sets

When plotting large data sets, points often obscure one another. The `ggplot2` package offers several ways to deal with this problem, including decreasing point size, adding jitter and/or transparency, displaying density contours, and replacing sets of points with hexagonal bins.

Scatter Plots with Jitter and Transparency

By adjusting the amount of jitter and the amount of transparency, you can find a good combination that lets you see through points into the heart of a dense scatter plot (Fig. 16.21).

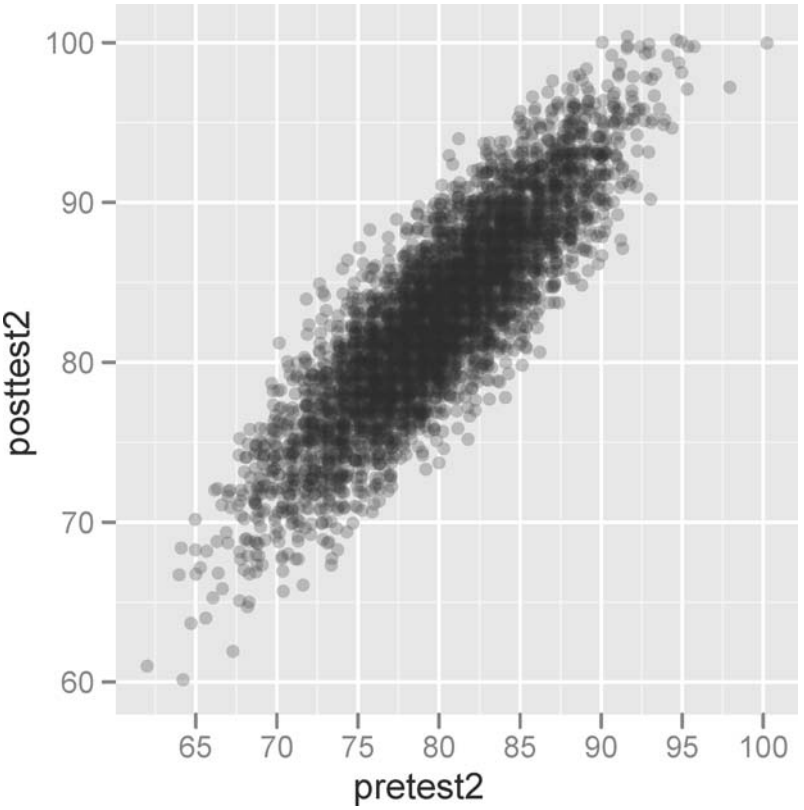

Fig. 16.21. A scatter plot demonstrating how transparency allows you to see many points at once.

Unfortunately, transparency is not yet supported in Windows metafiles. So if you are a Windows user, choose "Copy as bitmap" when cutting and pasting graphs into your word processor. For a higher-resolution image, route your graph to a file using the `png` function. For an example, see Section 14.6, "Graphics Devices." You can also use the `ggsave` function, which is part of the `ggplot2` package. For details, see Section 16.16, "Saving `ggplot2` Graphs to a File."

To get 5,000 points to work with, we generated them with the following:

```
pretest2  <- round( rnorm(n=5000,mean=80,sd=5) )

posttest2 <- round( pretest2 + rnorm(n=5000,mean=3,sd=3) )

pretest2[pretest2>100] <- 100

posttest2[posttest2>100] <- 100

temp=data.frame(pretest2,posttest2)
```

Now let us plot this data. This builds on our previous plots that used jitter and size. In computer terminology, controlling transparency is called *alpha compositing*. The `qplot` function makes this easy with a simple `alpha` argument. You can try various levels of transparency until you get the result you desire.

The `size` and `alpha` arguments could be set as variables. In which case, they would vary the point size or transparency to reflect the levels of the assigned variables. That would require a legend to help us interpret the plot. However, when you want to set them equal to fixed values, you can nest the numbers using the `I()` function. The `I()` function *i*nhibits the *i*nterpretation of its arguments. Without the `I()` function, the `qplot` function would print a legend saying that "size=2" and "alpha=0.15," which in our case is fairly useless information.

The `ggplot` function controls transparency with the `colour` argument to the `geom_jitter` function. That lets you control color and amount of transparency in the same option.

The `qplot` approach to Fig. 16.21 is

```
qplot(pretest2, posttest2, data=temp,
  geom="jitter", size=I(2), alpha=I(0.15),
  position=position_jitter(width=2) )
```

The `ggplot` approach to Fig. 16.21 is

```
ggplot(temp, aes(pretest2, posttest2),
  size=2, position = position_jitter(x=2,y=2) ) +
  geom_jitter(colour=alpha("black",0.15) )
```

Scatter Plots with Density Contours

A different approach to study a dense scatter plot is to draw density contours on top of the data (Fig. 16.22). With this approach, it is often better not to jitter the data, so that you can more clearly see the contours. You can do this with the `density2d` geom in `qplot` or the `geom_density` function in `ggplot`. The `size=I(1)` argument below reduces the point size to make it easier to see many points at once. As before, the `I()` function simply suppressed a superfluous legend.

The `qplot` approach to Fig. 16.22 is

```
qplot(pretest2, posttest2, data=temp,
  geom=c("point","density2d"), size = I(1) )
```

The `ggplot` approach to Fig. 16.22 is

```
ggplot(temp, aes( pretest2, posttest2) ) +
  geom_point( size=1 ) + geom_density_2d()
```

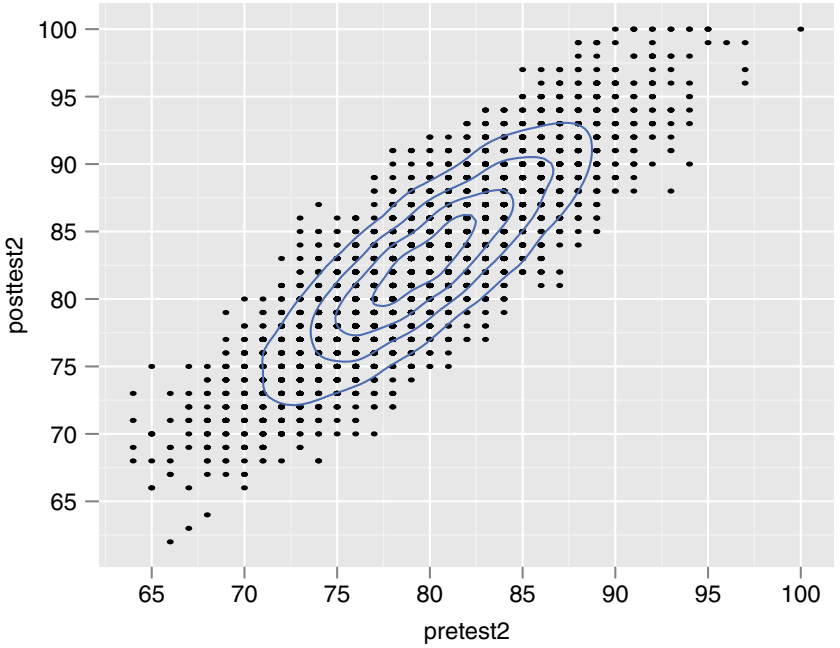

Fig. 16.22. This scatter plot shows an alternate way to see the structure in a large data set. These points are small, but not jittered, making more space for us to see the density contour lines.

16.10.3 Hexbin Plots

Another approach to plotting large data sets is to divide the plot surface into a set of hexagons and shade each hexagon to represent the number of points that fall within it; see Fig. 16.23. In that way, you can scale millions of points down into tens of bins.

In `qplot`, we can use the `hex` geom. In `ggplot`, we use the equivalent `geom_hex` function. Both use the `bins` argument to set the number of hexagonal bins you want. The default is 30; we use it here only so that you can see how to change it. As with histograms, increasing the number of bins may reveal more structure within the data.

The following function call uses `qplot` to create a color version of Fig. 16.23:

```
qplot(pretest2, posttest2, geom="hex", bins=30)
```

The following code uses `ggplot` to create the actual greyscale version of Fig. 16.23. The `scale_fill_continuous` function allows us to shade the plot using levels of grey. You can change the `low="grey80"` argument to other values to get the range of grey you prefer. Of course, you could add this function call to the above `qplot` call to get it to be grey instead of color.

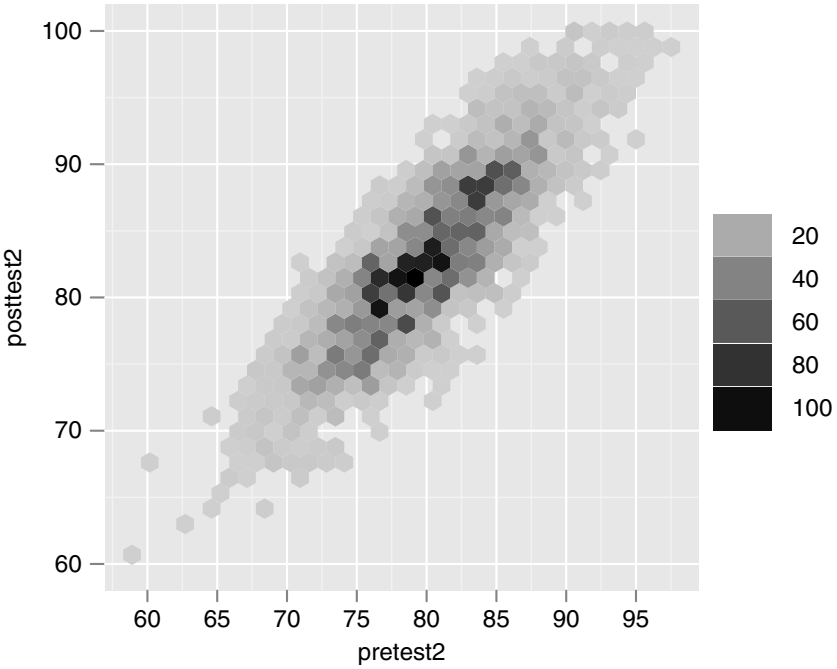

Fig. 16.23. A hexbin plot of pretest and posttest.

```
ggplot(temp, aes(pretest2, posttest2))  +
  geom_hex( bins=30 ) +
  scale_fill_continuous(
    low = "grey80", high = "black")
```

16.10.4 Scatter Plots with Fit Lines

While the traditional graphics `plot` function took quite a lot of extra effort to add confidence lines around a regression fit (Fig. 15.38), the `ggplot2` package makes that automatic. Unfortunately, the transparency used to create the confidence band is not supported when you cut and paste the image as a metafile in Windows. The image in Fig. 16.24 is a slightly lower resolution 600-dpi bitmap.

To get a regression line in `qplot`, simply specify `geom="smooth"`. However, that alone will replace the default of `geom="point"`, so if you want both, you need to specify `geom=c("point","smooth")`.

In `ggplot`, you use both the `geom_point` and `geom_smooth` functions. The default smoothing method is a lowess function, so if you prefer a linear model, include the `method=lm` argument.

The `qplot` approach to Fig. 16.24 is

```
qplot(pretest, posttest,
  geom=c("point","smooth"), method=lm )
```

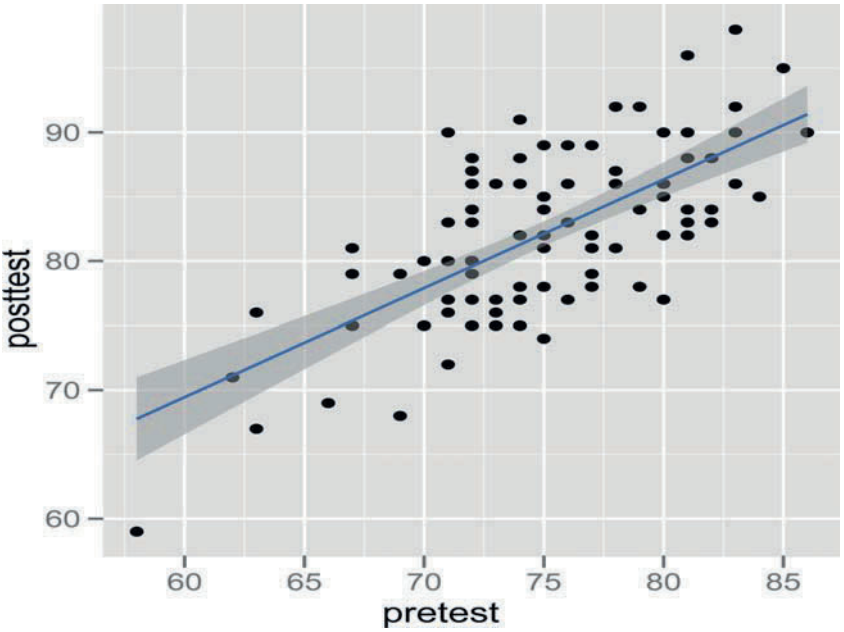

Fig. 16.24. A scatter plot with regression line and default confidence band.

The `ggplot` approach to Fig. 16.24 is

```
ggplot(mydata100, aes(pretest, posttest) ) +
  geom_point() + geom_smooth(method=lm)
```

Since the confidence bands appear by default, we have to set the `se` argument (standard error) to `FALSE` to turn it off.

The `qplot` approach to Fig. 16.25 is

```
qplot(pretest, posttest,
  geom=c("point","smooth"), method=lm, se=FALSE )
```

The `ggplot` approach to Fig. 16.25 is

```
ggplot(mydata100, aes(pretest, posttest) ) +
  geom_point() + geom_smooth(method=lm, se=FALSE)
```

16.10.5 Scatter Plots with Reference Lines

To place an arbitrary straight line on a plot, use the `abline` geom in `qplot`. You specify your slope and intercept using clearly named arguments. Here we are using intercept=0 and slope=1 since this is the line where posttest=pretest. If the students did not learn anything in the workshops,

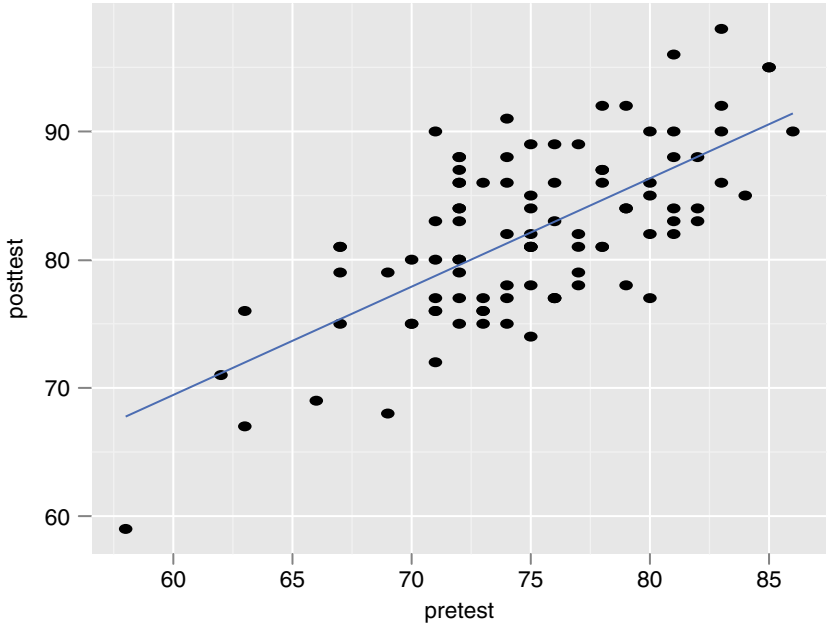

Fig. 16.25. A scatter plot with regression line with default confidence band removed.

the data would fall on this line (assuming a reliable test). The `ggplot` function adds the `abline` function with arguments for intercept and slope.

The `qplot` approach to Fig. 16.26 is

```
qplot(pretest, posttest,
  geom=c("point","abline"),
  intercept=0, slope=1 )
```

The `ggplot` approach to Fig. 16.26 is

```
ggplot(mydata100, aes(pretest, posttest) ) +
  geom_point()+ geom_abline( intercept=0, slope=1 )
```

Vertical or horizontal reference lines can help emphasize points or cutoffs. For example, if our students are required to get a score greater than 75 before moving on, we might want to display those cutoffs on our plot (Fig. 16.27).

In `qplot`, we can do this with the `xintercept` and `yintercept` arguments. In `ggplot`, the functions are named `geom_vline` and `geom_hline`, each with an `intercept` argument.

The `qplot` approach to Fig. 16.27 is

```
qplot(pretest, posttest,
 geom=c("point", "vline", "hline"),
 xintercept=75, yintercept=75)
```

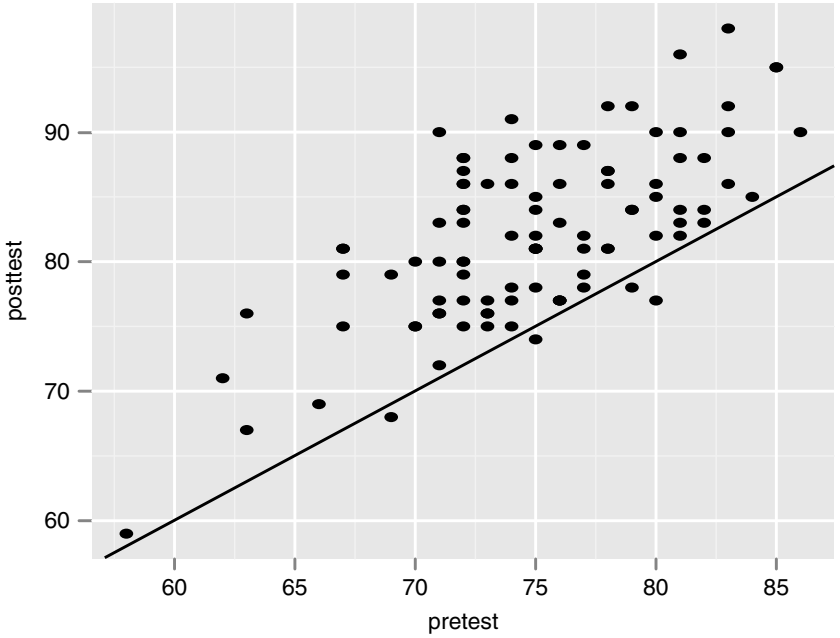

Fig. 16.26. A scatter plot with a line added where pretest=posttest. Most of the points lie above this line, showing that students did learn.

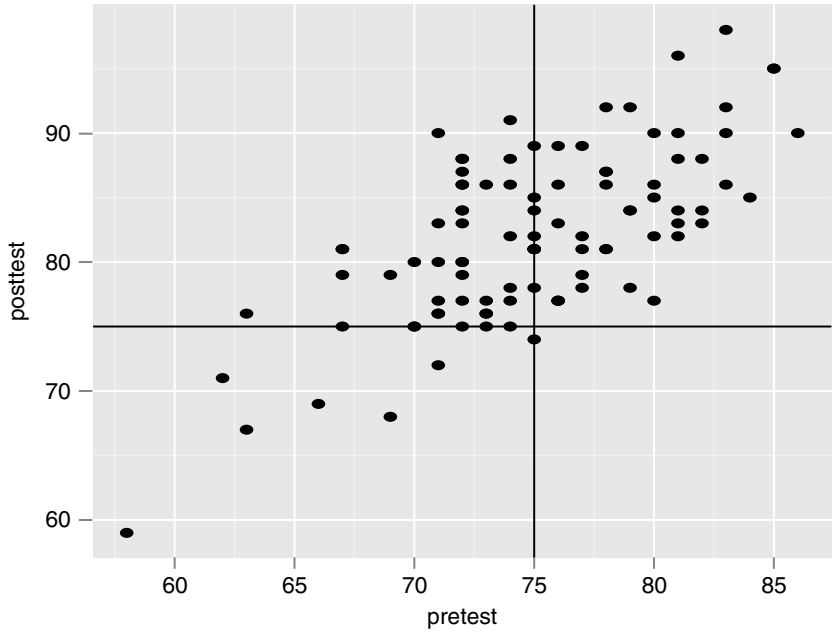

Fig. 16.27. A scatter plot with vertical and horizontal reference lines.

The ggplot approach to Fig. 16.27 is

```
ggplot(mydata100, aes(pretest, posttest)) +
  geom_point() +
  geom_vline( xintercept=75 ) +
  geom_hline( yintercept=75 )
```

To add a series of reference lines, we need to use the geom_vline or geom_hline functions (Fig. 16.28). The qplot example does not do much with qplot itself since it cannot create multiple reference lines. So for both examples, we use the identical geom_vline function. It includes the seq function to generate the *seq*uence of numbers we needed. Without it we could have used intercept=c(70,72,74,76,78,80). In this case, we did not save much effort, but if we wanted to add dozens of lines, the seq function would be much easier.

The qplot approach to Fig. 16.28 is

```
qplot(pretest, posttest, type="point") +
  geom_vline( intercept=seq(from=70,to=80,by=2) )
```

The ggplot approach to Fig. 16.28 is

```
ggplot(mydata100, aes(pretest, posttest)) +
  geom_point() +
  geom_vline( xintercept=seq(from=70,to=80,by=2) )
```

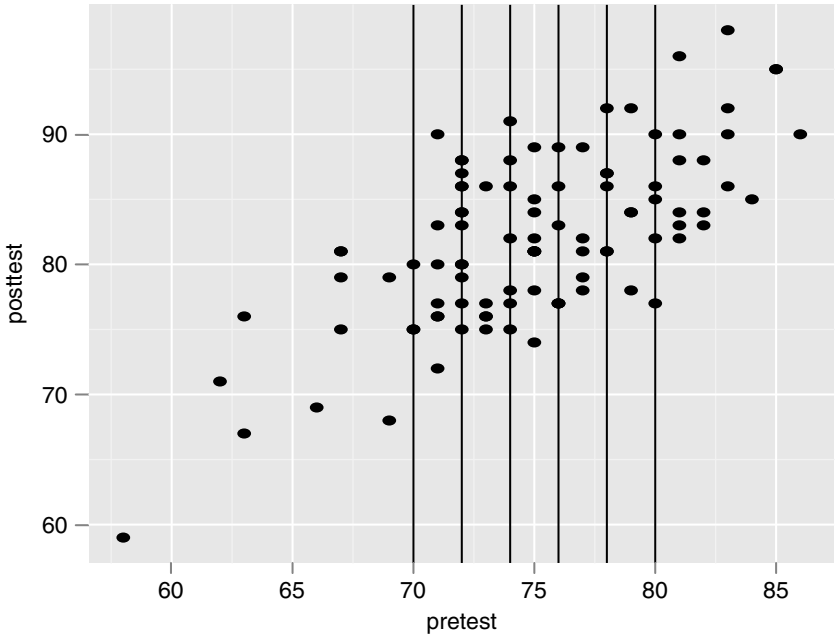

Fig. 16.28. A scatter plot with multiple vertical reference lines.

16.10.6 Scatter Plots with Labels Instead of Points

If you do not have much data or you are only interested in points around the edges, you can plot labels instead of plots symbols (Fig. 16.29). The labels can be identifiers such as ID numbers, people's names or row names, or they could be values of other variables of interest to add a third dimension to the plot.

You do this using the `geom="text"` argument in `qplot` or the `geom_text` function in `ggplot`. In either case, the `label` argument points to the values to use. Recall that in R, `row.names(mydata)` gives you the stored row names, even if these are just the sequential characters, "1," "2," and so on. We will store them in a variable named `mydata$id` and then use it with the `label` argument. The reason we do not use the form `label=row.names(mydata100)` is that the `ggplot2` package puts all of the variables it uses into a separate temporary data frame before running.

The `qplot` approach to Fig. 16.29 is

```
mydata100$id <- row.names(mydata100)
qplot(pretest, posttest, geom="text",
  label=mydata100$id )
```

The `ggplot` approach to Fig. 16.29 is

```
ggplot(mydata100, aes(pretest, posttest,
  label=mydata100$id ) ) + geom_text()
```

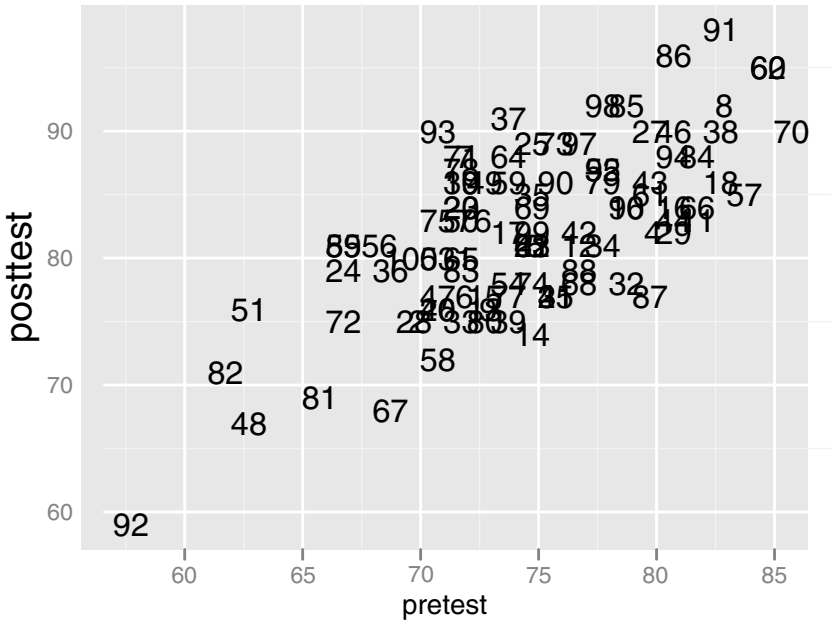

Fig. 16.29. A scatter plot with ID numbers plotted instead of points.

16.10.7 Changing Plot Symbols

You can use different plot symbols to represent levels of any third variable. Factor values, such as those representing group membership, can be displayed by different plot symbols (shapes) and/or colors. You could use a continuous third variable to shade the colors of each point or to vary the size of each point. The `ggplot2` package makes quick work of any of these options. Let us consider a plot of pretest versus posttest that uses different points for males and females (Fig. 16.30).

The `qplot` function can do this using the `shape` argument.

The `ggplot` function must bring a new variable into the `geom_point` function. Recall that aesthetics map variables into their roles, so we will nest `aes(shape=gender)` within the call to `geom_point`.

You can also set `color` and `size` by substituting either of those arguments for `shape`.

The `qplot` approach to Fig. 16.30 is

```
qplot(pretest, posttest, shape=gender)
```

The `ggplot` approach to Fig. 16.30 is

```
ggplot(mydata100, aes(pretest, posttest) ) +
  geom_point( aes(shape=gender) )
```

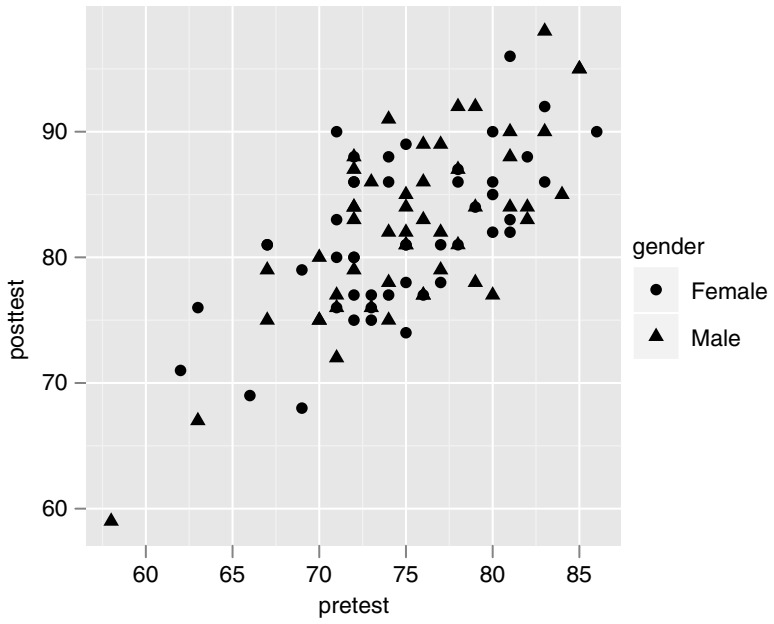

Fig. 16.30. A scatter plot with point shape determined by gender.

16.10.8 Scatter Plot with Linear Fits by Group

We have seen that the `smooth` geom adds a lowess or regression line and that `shape` can include group membership. If we do both of these in the same plot, we can get separate lines for each group as shown in Fig. 16.31.

The `qplot` approach to Fig. 16.31 is

```
qplot(pretest, posttest, geom=c("smooth","point"),
  method="lm", shape=gender)
```

The `ggplot` approach to Fig. 16.31 is

```
ggplot(mydata100,
  aes(x = pretest, y=posttest, shape=gender) ) +
  geom_smooth( method="lm" ) + geom_point()
```

16.10.9 Scatter Plots Faceted for Groups

Another way to compare groups on scatter with or without lines of fit is through facets (Fig. 16.32). As we have seen several times before, simply adding the `facets` argument to the `qplot` function allows you to specify `rows~columns` of categorical variables. So `facets=workshop~gender` is

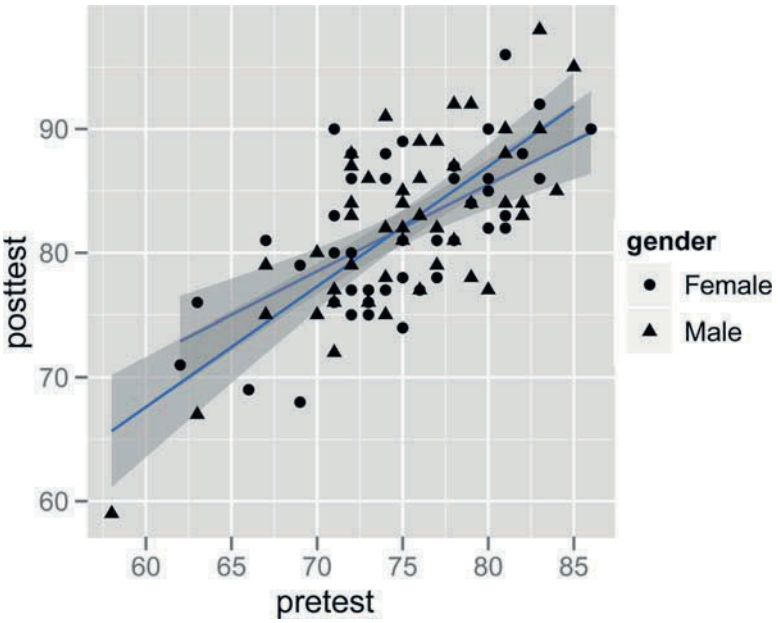

Fig. 16.31. A scatter plot with regression lines and point shape determined by gender.

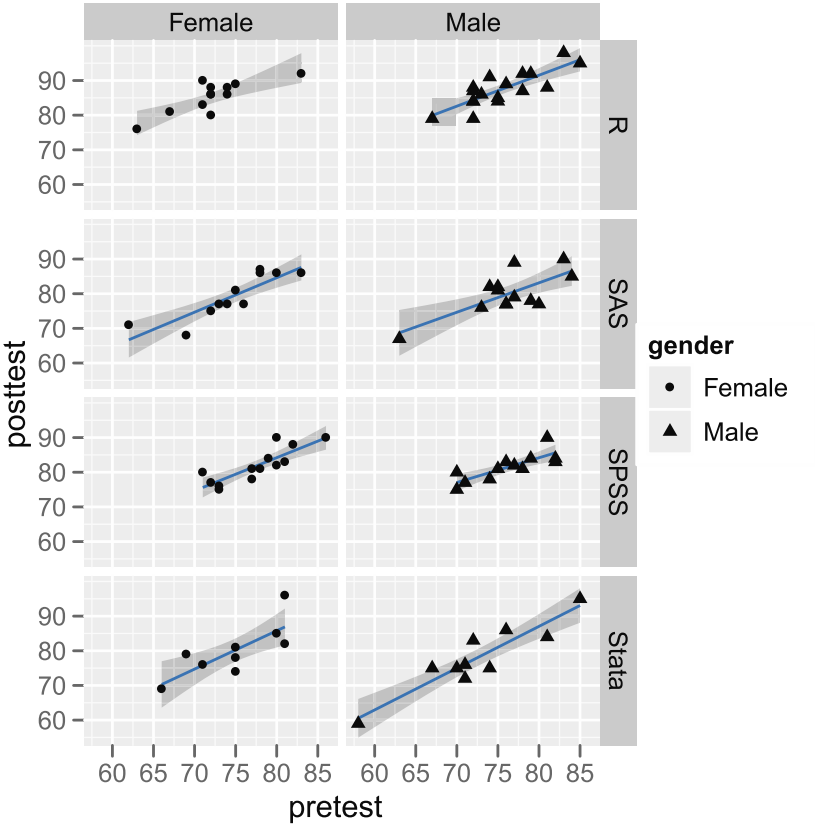

Fig. 16.32. A scatter plot with facets showing linear fits for each workshop and gender combination.

requesting a grid of plots for each workshop:gender combination, with workshop determining the rows and gender determining the columns.

The `ggplot` function works similarly, using the `facet_grid` function to do the same. If you have a continuous variable to condition on, you can use the `chop` function from the `ggplot2` package or the `cut` function that is built into R to break the variable into groups.

The `qplot` approach to Fig. 16.32 is

```
qplot(pretest, posttest, geom=c("smooth","point"),
  method="lm", shape=gender, facets=workshop ~ gender)
```

The `ggplot` approach to Fig. 16.32 is

```
ggplot(mydata100, aes( pretest, posttest) ) +
  geom_smooth( method="lm" ) + geom_point() +
  facet_grid( workshop ~ gender )
```

16.10.10 Scatter Plot Matrix

When you have many variables to plot, a scatter plot matrix is helpful (Fig. 16.33). You lose a lot of detail compared to a set of full-sized plots, but if your data set is not too large, you usually get the gist of the relationships.

The `ggplot2` package has a separate `plotmatrix` function for this type of plot. Simply entering the following will plot variables 3 through 8 against one another (not shown):

```
plotmatrix( mydata100[3:8] )
```

You can embellish the plots with many of the options we have covered earlier in this chapter. Shown below is an example of a scatter plot matrix

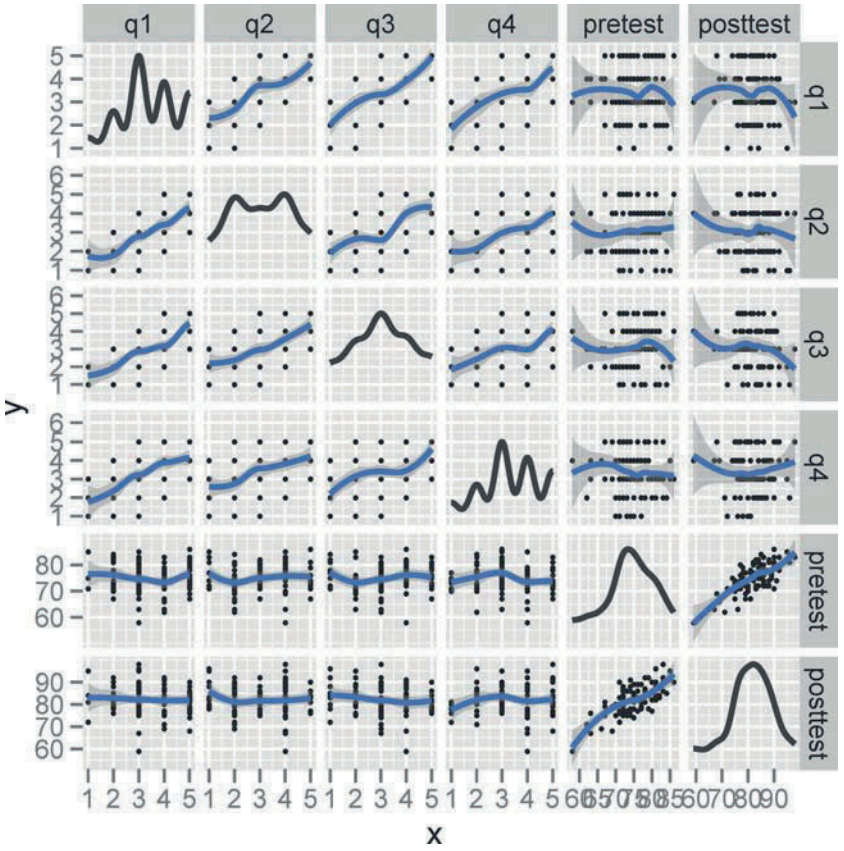

Fig. 16.33. A scatter plot matrix with lowess curve fits on the off-diagonal plots, and density plots on the diagonals.

(Fig. 16.33) with smoothed lowess fits for the entire data set (i.e., not by group). The density plots on the diagonals appear by default.

```
plotmatrix( mydata100[3:8] ) +
  geom_smooth()
```

The lowess fit generated some warnings but that is not a problem. It said, "There were 50 or more warnings (use warnings() to see the first 50)."

The next example gets fancier by assigning a different symbol shape and linear fits per group (plot not shown.)

```
plotmatrix( mydata100[3:8],
  aes( shape=gender ) ) +
  geom_smooth(method=lm)
```

16.11 Box Plots

We discussed what box plots are in the Chapter "Traditional Graphics," Section 15.11. We can recreate all those examples using the `ggplot2` package, except for the "notches" to indicate possible group differences, shown in the upper right of Fig. 15.45

The simplest type of box plot is for a single variable (Fig. 16.34). The `qplot` function uses the simple form of `factor("")` to act as its x-axis value.

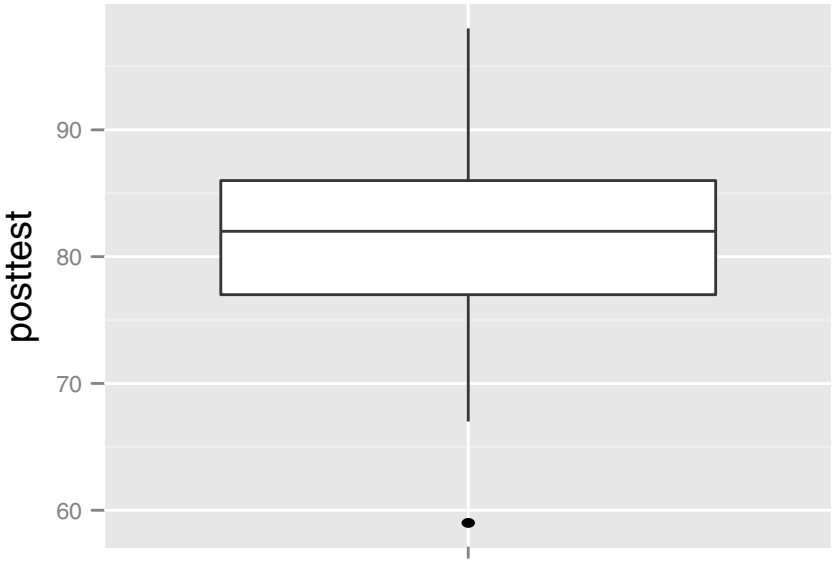

Fig. 16.34. A box plot of posttest.

The y value is the variable to plot: in this case, posttest. The geom of `boxplot` specifies the main display type. The `xlab=""` argument blanks out the label on the x-axis, which would have been a meaningless "`factor("")`".

The equivalent `ggplot` approach is almost identical with its ever-present `aes` arguments for x and y and the `geom_boxplot` function to draw the box. The `scale_x_discrete` function simply blanks out the x-axis label.

The `qplot` approach to Fig. 16.34 is

```
qplot(factor(""), posttest,
  geom="boxplot", xlab="")
```

The `ggplot` approach to Fig. 16.34 is

```
ggplot(mydata100,
  aes(factor(""), posttest) ) +
  geom_boxplot() +
  scale_x_discrete("")
```

Adding a grouping variable like workshop makes box plots much more informative (Fig. 16.35, ignore the overlaid strip plot points for now). These are the same function calls as above but with the `x` variable specified as workshop. We will skip showing this one in favor of the next.

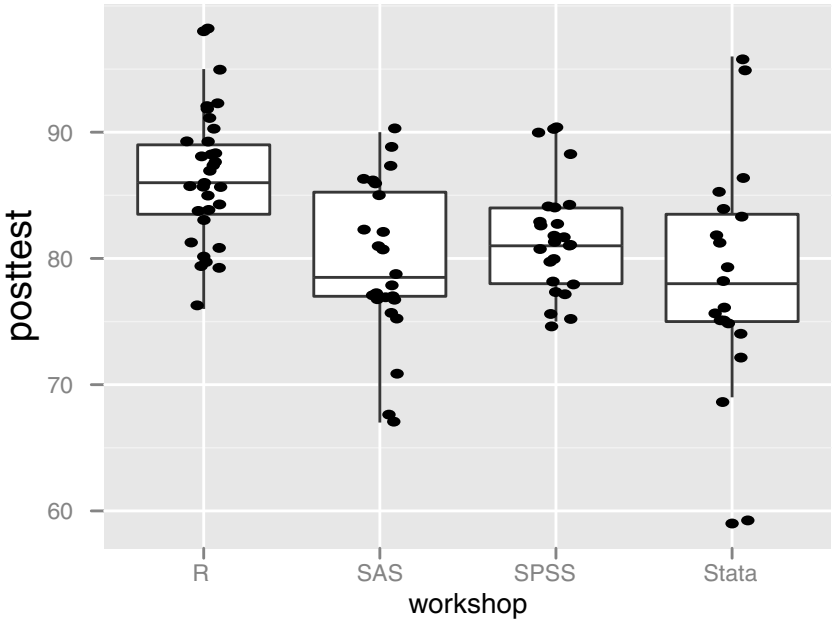

Fig. 16.35. A box plot comparing workshop groups on posttest, with jittered points on top.

The qplot approach to box plots (figure not shown) is

```
qplot(workshop, posttest, geom="boxplot" )
```

The ggplot approach to box plots (figure not shown) is

```
ggplot(mydata100,
  aes( workshop, posttest) ) +
  geom_boxplot()
```

Now we will do the same plot but with an added jittered strip plot on top of it (Fig. 16.35). This way we get the box plot information about the median and quartiles plus we get to see any interesting structure in the points that would otherwise have been lost. As you can see, the `qplot` now has jitter added to its `geom` argument, and `ggplot` has an additional `geom_jitter` function. Unfortunately, the amount of jitter that both functions provide by default is optimized for a much larger data set. So these next two sets of code do the plot shown in Fig. 16.35, but with much more jitter.

The qplot approach to Fig. 16.35 with more jitter added is

```
qplot(workshop, posttest,
  geom=c("boxplot","jitter") )
```

The ggplot approach to Fig. 16.35 with more jitter added is

```
ggplot(mydata100,
  aes(workshop, posttest )) +
  geom_boxplot() + geom_jitter()
```

The following is the exact code that created Fig. 16.35. The `qplot` function does not have enough control to request both the box plot and jitter while adjusting the amount of jitter.

```
ggplot(mydata100,
 aes(workshop, posttest )) +
 geom_boxplot() +
 geom_jitter(position=position_jitter(width=.1))
```

To add another grouping variable, you only need to only add the `fill` argument to either `qplot` or `ggplot`. Compare the resulting Fig. 16.36 to the result we obtained from traditional graphics, in the lower right panel of Fig. 15.45. The `ggplot2` version is superior in many ways. The genders are easier to compare for a given workshop, because they are now grouped side-by-side. The shading makes it easy to focus on one gender at a time to see how they changed across the levels of workshop. The labels are easier to read and did not require the custom sizing that we did earlier to make room for the labels. The `ggplot2` package usually does a better job with complex plots and makes quick work of them too.

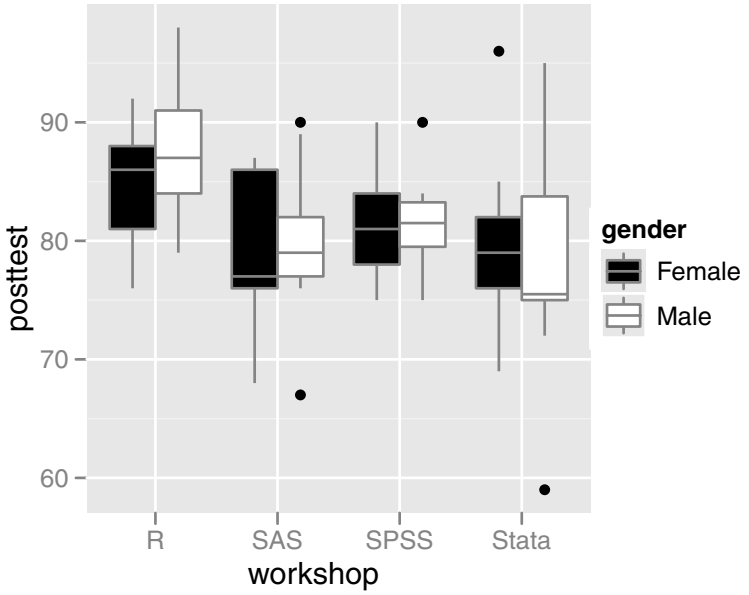

Fig. 16.36. A box plot comparing workshop and gender groups on posttest.

The `qplot` approach to Fig. 16.36 is

```
qplot(workshop, posttest,
  geom="boxplot", fill=gender ) +
  scale_fill_grey( start=0, end=1 )
```

The `ggplot` approach to Fig. 16.36 is

```
ggplot(mydata100,
  aes(workshop, posttest) ) +
  geom_boxplot( aes(fill=gender), colour="black") +
  scale_fill_grey( start=0, end=1 )
```

16.12 Error Bar Plots

Plotting means and 95% confidence intervals, as in Fig. 16.37, is a task that stretches what `qplot` was designed to do. As you can see from the two examples below, there is very little typing saved by using `qplot` over `ggplot`. In both cases, we are adding a jittered strip plot of points, as we did earlier in the section on strip plots (Section 16.9). Notice that we had to use the `as.numeric` function for our `x` variable: workshop. Since workshop is a factor, the software would not connect the means across the levels of `x`. Workshop

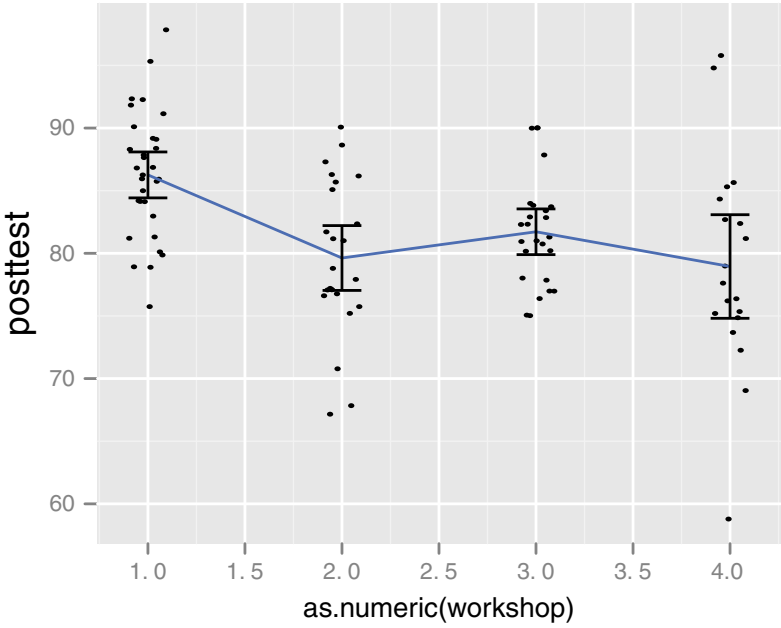

Fig. 16.37. An error bar plot with lines running through the means, with default axis labels.

is not a continuous variable, so that makes sense! Still, connecting the means with a line is a common approach, one that facilitates the study of higher level-interactions.

The key function for this plot is `stat_summary`, which we use twice. First, we use the argument `fun.y="mean"` to calculate the group means. We also use the `geom="smooth"` argument to connect them with a line. Next, we use `fun.data="mean_cl_normal"` to calculate *c*onfidence *l*imits for the means based on a normal distribution and display them with the `errorbar` geom. You can try various values for the `width` argument until you are satisfied with the error bar widths.

```
qplot( as.numeric(workshop), posttest) +
  geom_jitter(position=position_jitter(width=.1))  +
  stat_summary(fun.y="mean",
    geom="smooth", se=FALSE) +
  stat_summary(fun.data="mean_cl_normal",
    geom="errorbar", width=.2)

ggplot(mydata100,
  aes( as.numeric(workshop), posttest ) ) +
  geom_jitter(size=1,
    position=position_jitter(width=.1) )  +
```

```
stat_summary(fun.y="mean",
  geom="smooth", se=FALSE) +
stat_summary(fun.data="mean_cl_normal",
  geom="errorbar", width=.2)
```

Since we have a fairly small data set, replacing the `geom_jitter` function with just `geom_point(size=1)` creates a nice plot too (not shown).

16.13 Logarithmic Axes

If your data has a very wide range of values, working in a logarithmic scale is often helpful. In `ggplot2` you can approach this in three different ways. First, you can take the logarithm of the data before plotting:

```
qplot( log(pretest), log(posttest) )
```

Another approach is to use evenly placed tick-marks on the plot but have the axis values use logarithmic values such as 10^1, 10^2, and so on. This is what the `scale_x_log10` function does (similarly for the y-axis, of course). There are similar functions for natural logarithms, `scale_x_log` and base 2 logarithms, `scale_x_log2`:

```
qplot(pretest, posttest, data=mydata100) +
  scale_x_log10() + scale_y_log10()
```

Finally, you can have the tick marks spaced unevenly and use values on your original scale. The `coord_trans` function does that. Its arguments for the various bases of logarithms are `log10`, `log`, and `log2`.

```
qplot(pretest, posttest, data=mydata100) +
  coord_trans("log10", "log10")
```

With our data set, the range of values is so small that this last plot will not noticeably change the axes. Therefore, we do not show it.

16.14 Aspect Ratio

Changing the aspect ratio of a graph can be far more important than you might first think. Research has shown that when most of the lines or scatter on a plot are angled at 45°, people make more accurate comparisons to those parts that are not [6].

Unless you specify an aspect ratio for your graph, `qplot` and `ggplot` will match the dimensions of your output window and allow you to change those dimensions using your mouse, as you would for any other window.

If you are routing your output to a file however, it is helpful to be able to set it using code. You set the aspect ratio using the `coord_equal` function. If you leave it empty, as in `coord_equal()`, it will make the x- and y-axes of equal lengths. If you specify this while working interactively, you can stillreshape

your window, but the graph will remain square. Specifying a ratio parameter follows the form "height/width." For a mnemonic, think of how R specifies [rows,columns]. The following example would result in a graph that is four times wider than it is high (not shown):

```
qplot(pretest, posttest) + coord_equal(ratio=1/4)
```

16.15 Multiple Plots on a Page

In the previous chapter on traditional graphics, we discussed how to put multiple plots on a page. However, `ggplot2` uses the grid graphics system, so that method does not work. We saw the multiframe plot in Fig. 16.38 in the section on bar plots. Let us now look at how it was constructed. We will skip the bar plot details here and focus on how we combined them.

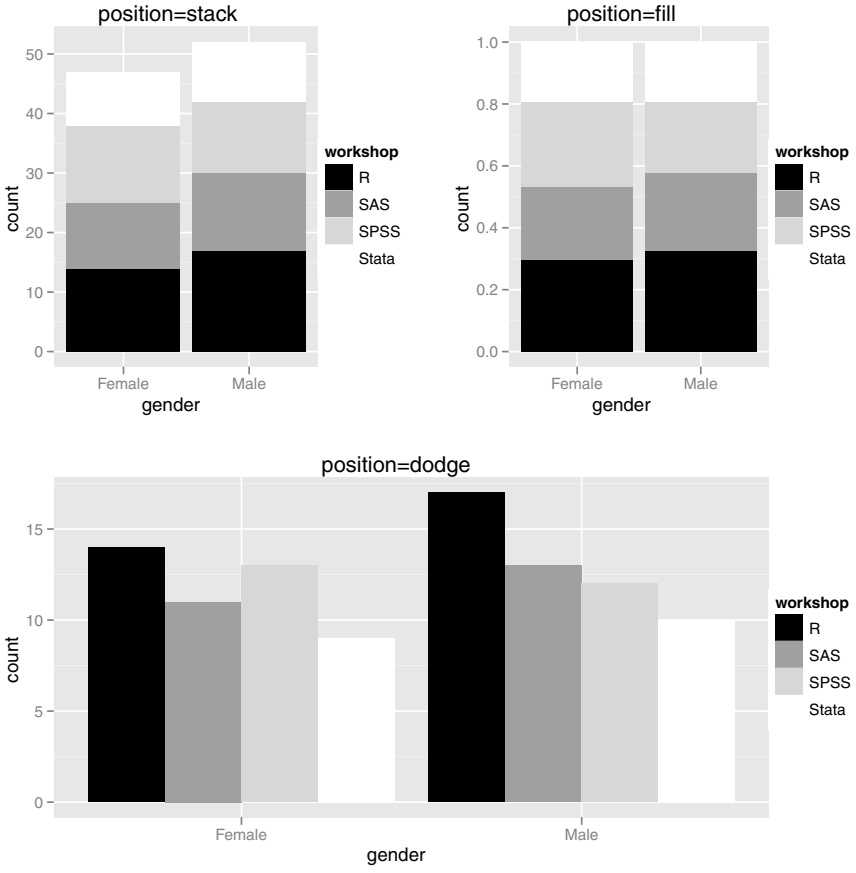

Fig. 16.38. A multiframe plot showing the impact of the various position settings.

We first clear the page with the `grid.newpage` function. This is an important step as otherwise plots printed using the following methods will appear on top of others.

```
grid.newpage()
```

Next, we use the `pushViewport` function to define the various frames called *viewports* in the grid graphics system. The `grid.layout` argument uses R's common format of (rows, columns). The following example sets up a two by two grid for us to use:

```
pushViewport( viewport(layout=grid.layout(2,2) ) )
```

In traditional graphics, you would now just do the graphs in order and they would find their place. However, in the grid system, we must save the plot to an object and then use the `print` function to print it into the viewport we desire. The object name of "p" is commonly used as an object name for the plot. Since there are many ways to add to this object, it is helpful to keep it short. To emphasize that this is something we get to name, we will use "myPlot."

The `print` function has a `vp` argument that lets you specify the *v*iew*p*ort's position in row(s) and column(s). In the following example, we will print the graph to row 1 and column 1:

```
myPlot <- ggplot(mydata100,
    aes(gender, fill=workshop) ) +
  geom_bar(position="stack") +
  scale_fill_grey(start = 0, end = 1) +
  opts( title="position=stack " )

print(myPlot, vp=viewport(
  layout.pos.row=1,
  layout.pos.col=1) )
```

The next plot prints to row 1 and column 2.

```
myPlot <- ggplot(mydata100,
    aes(gender, fill=workshop) ) +
  geom_bar(position="fill") +
  scale_fill_grey(start = 0, end = 1) +
  opts( title="position=fill" )

print(myPlot, vp=viewport(
  layout.pos.row=1,
  layout.pos.col=2) )
```

The third and final plot is much wider than the first two. So we will print it to row 2 in both columns 1 and 2. Since we did not set the aspect ratio explicitly, the graph will resize to fit the double-wide viewport.

```
myPlot <- ggplot(mydata100,
    aes(gender, fill=workshop) ) +
  geom_bar(position="dodge")  +
  scale_fill_grey(start = 0, end = 1) +
  opts( title="position=dodge" )

print(myPlot, vp=viewport(
  layout.pos.row=2,
  layout.pos.col=1:2) )
```

The next time you print a plot without specifying a viewport, the screen resets back to its previous full-window display. The code for the other multiframe plots is in the example program in Section 16.19.

16.16 Saving ggplot2 Graphs to a File

In Section 14.6, "Graphics Devices," we discussed various ways to save plots in files. Those methods work with the `ggplot2` package, and in fact they are the only way to save a multiframe plot to a file.

However, the `ggplot2` package has its own function that is optimized for saving single plots to a file. To save the last graph you created, with either `qplot` or `ggplot`, use the `ggsave` function. It will choose the proper graphics device from the file extension.

For example, the following function call will save the last graph created in an encapsulated postscript file:

```
> ggsave("mygraph.eps")

Saving 4.00" x 3.50" image
```

It will choose the width and height from your computer monitor and will report back those dimensions. If you did not get it right, you can change those dimensions and rerun the function. Alternately, you can specify the `width` and `height` arguments in inches or, for bitmapped formats like Portable Network Graphics (png), in dots-per-inch. See `help(ggsave)` for additional options.

16.17 An Example Specifying All Defaults

Now that you have seen some examples of both `qplot` and `ggplot`, let us take a brief look at the full power of `ggplot` by revisiting the scatter plot with a regression line (Fig. 16.39). We will first review both sets of code, exactly as described in Section 16.10.4.

First, done with `qplot`, it is quite easy and it feels similar to the traditional graphics `plot` function:

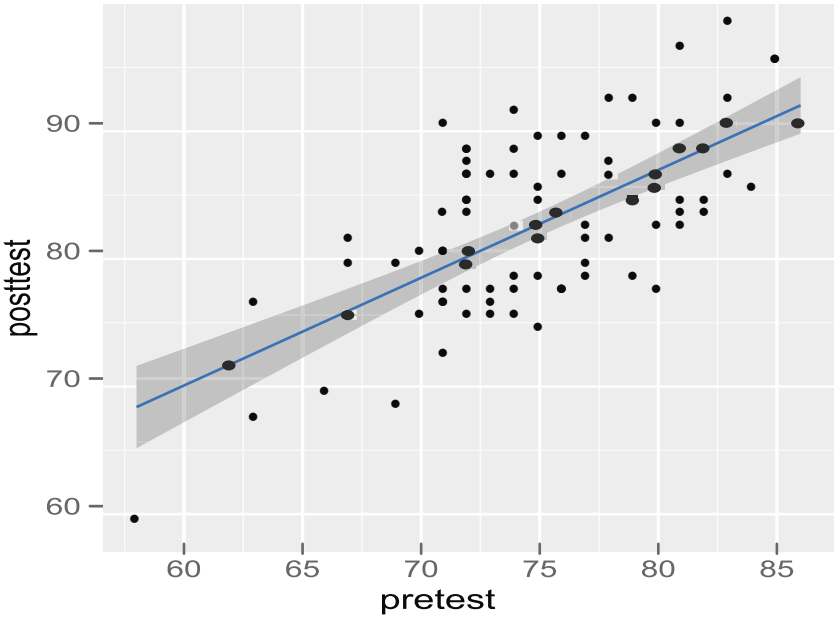

Fig. 16.39. This same scatter plot results from several types of programs shown in the text.

```
qplot(pretest, posttest,
  geom=c("point","smooth"), method="lm" )
```

Next, let us do it using `ggplot` with as many default settings as possible. It is not too much more typing, and it brings us into the grammar of graphics world. We see the new concepts of aesthetic mapping of variables and geometric objects, or geoms:

```
ggplot(mydata100, aes(pretest, posttest) ) +
  geom_point() +
  geom_smooth(method="lm")
```

Finally, here it is again in `ggplot` but with no default settings. We see that the plot is actually two layers: one with points and another with the smooth line. Each layer can use different data frames, variables, geometric objects, statistics, and so on. If you need graphics flexibility, `ggplot2` is the package for you!

```
ggplot() +
layer(
  data=mydata100,
  mapping=aes(pretest, posttest),
  geom="point",
```

```
  stat="identity"
) +
layer(
  data=mydata100,
  mapping=aes(pretest, posttest),
  geom="smooth",
  stat="smooth",
  method="lm"
) +
coord_cartesian()
```

16.18 Summary of Graphic Elements and Parameters

We have seen many ways to modify plots in the `ggplot2` package. The `ggopt` function is another way. You can set the parameters of all future graphs in the current session with the following function call. See `help(ggopt)` function for many more parameters.

```
ggopt(
  background.fill = "black",
  background.color ="white",
  axis.colour = "black"  #default axis fonts are grey.
  )
```

The `opts` function is useful for modifying settings for a single plot. For example, when colors, shapes, or labels make a legend superfluous, you can suppress it with

```
+ opts(legend.position="none")
```

See `help(opts)` for more examples.

The plots created with both `qplot` and `ggplot` make copious use of color. Since our examples did not really need color we supressed it with

```
...+ scale_fill_grey(start=0,end=1)
```

An alternate way of doing this the `theme_set` function. To use levels of grey, use:

```
theme_set( theme_grey() )
```

To limit colors to black and white, use:

```
theme_set( theme_bw() )
```

To return to the default colors, use:

```
theme_set()
```

Enter `help(theme_set)` for details.

16.19 Example Programs for ggplot2

Stata does not offer the grammar of graphics model. See previous the chapter for Stata graphics examples.

This program brings together the examples discussed in this chapter and a few variations that were not.

```
# Filename: GraphicsGG.R

setwd("/myRfolder")
load(file="mydata100.Rdata")
detach(mydata100) #In case I'm running repeatedly.

# Get rid of missing values for facets
mydata100 <- na.omit(mydata100)
attach(mydata100)
library(ggplot2)

# ---Barplots---

# Barplot - Vertical

qplot(workshop)

ggplot(mydata100, aes( workshop ) ) +
  geom_bar()

# Can also follow this form:
ggplot( mydata100 ) +
  aes( x=workshop ) +
  geom_bar()

# Barplot - Horizontal

qplot(workshop) + coord_flip()

ggplot(mydata100, aes( workshop ) ) +
  geom_bar() + coord_flip()

# Barplot - Single Bar Stacked

qplot(factor(""), fill=workshop,
  geom="bar", xlab="") +
  scale_fill_grey(start=0, end=1)
```

```
ggplot(mydata100,
  aes(factor(""), fill=workshop) ) +
  geom_bar() +
  scale_x_discrete("") +
  scale_fill_grey(start=0, end=1)

# Pie charts, same as stacked bar
  but polar coordinates

qplot( factor(""), fill=workshop,
  geom="bar", xlab="") +
  coord_polar(theta="y") +
  scale_fill_grey(start=0, end=1)

ggplot(mydata100,
  aes( factor(""), fill=workshop ) ) +
  geom_bar( width=1 ) +
  scale_x_discrete("") +
  coord_polar(theta="y") +
  scale_fill_grey(start=0, end=1)

# Barplots - Grouped
#   position=stack, fill, dodge,
#   for qplot, then ggplot

qplot(gender, geom="bar",
    fill=workshop, position="stack") +
  scale_fill_grey(start = 0, end = 1)

qplot(gender, geom="bar",
    fill=workshop, position="fill")  +
  scale_fill_grey(start = 0, end = 1)

qplot(gender, geom="bar",
    fill=workshop, position="dodge") +
  scale_fill_grey(start = 0, end = 1)

ggplot(mydata100, aes(gender, fill=workshop) ) +
  geom_bar(position="stack") +
  scale_fill_grey(start = 0, end = 1)

ggplot(mydata100, aes(gender, fill=workshop) ) +
  geom_bar(position="fill") +
  scale_fill_grey(start = 0, end = 1)
```

```
ggplot(mydata100, aes(gender, fill=workshop ) ) +
  geom_bar(position="dodge") +
  scale_fill_grey(start = 0, end = 1)


# Barplots - Faceted

qplot(workshop, facets=gender~.)

ggplot(mydata100, aes(workshop) ) +
  geom_bar() + facet_grid( gender~. )


# Barplots - Presummarized data

qplot( factor(c(1,2)), c(40, 60), geom="bar",
  xlab="myGroup", ylab="myMeasure")

myTemp <- data.frame(
  myGroup=factor( c(1,2) ),
  myMeasure=c(40, 60)
)
myTemp

ggplot(data=myTemp, aes(myGroup, myMeasure) ) +
  geom_bar()


# ---Dotcharts---

qplot(workshop, geom="point", size=I(4),
  stat="bin", facets=gender~.) +
 coord_flip()

ggplot(mydata100) +
   aes(x=workshop, y=..count.. ) +
   geom_point(stat="bin", size=4) + coord_flip()+
   facet_grid( gender~. )


# ---Adding Titles and Labels---

qplot(workshop, geom="bar",
  main="Workshop Attendance",
  xlab="Statistics Package \nWorkshops")
```

```
ggplot(mydata100) +
  aes(workshop, ..count..) +
  geom_bar() +
  opts( title="Workshop Attendance" ) +
  scale_x_discrete("Statistics Package \nWorkshops")

# Example not in book: labels of continuous scales.
ggplot(mydata100) +
  aes(pretest,posttest ) +
  geom_point() +
  scale_x_continuous("Test Score Before Training") +
  scale_y_continuous("Test Score After Training")  +
  opts( title="The Relationship is Linear" )

# ---Histograms and Density Plots---

# Simle Histogram
qplot(posttest)

ggplot(mydata100) +
  aes(posttest)   +
  geom_histogram()

# Histogram with more bars.
qplot(posttest, geom="histogram", binwidth=0.5)

ggplot(mydata100) +
  aes(posttest)   +
  geom_histogram( binwidth=0.5 )

# Density plot
qplot(posttest, geom="density")

ggplot(mydata100) +
  aes(posttest)   +
  geom_density()

# Histogram with Density

qplot(data=mydata100,posttest, ..density..,
  geom=c("histogram","density") )

ggplot(mydata100, aes(posttest, ..density.. ) ) +
  geom_histogram() + geom_density()
```

```
# Histogram - Separate plots by group

qplot(posttest, geom="histogram", facets=gender~.)
qplot(posttest, facets=gender~.)

ggplot(mydata100, aes(posttest) ) +
  geom_histogram() + facet_grid( gender~. )

# Histograms - Overlaid

qplot( posttest, fill=gender ) +
  scale_fill_grey(start = 0, end = 1)

ggplot(mydata100, aes(posttest, fill=gender) ) +
  geom_bar() +
  scale_fill_grey(start = 0, end = 1)


# ---QQ plots---

qplot(sample=posttest, stat="qq")

ggplot( mydata100, aes(sample=posttest) ) +
  stat_qq()


# ---Strip plots---

# With too much jitter for our small data set:

qplot( factor(""), posttest, geom="jitter", xlab="")

ggplot(mydata100, aes(factor(""), posttest) ) +
  geom_jitter() +
  scale_x_discrete("")


# Strip plot by group.
qplot(workshop, posttest, geom="jitter")

ggplot(mydata100, aes(workshop, posttest) ) +
  geom_jitter()

# Again, with limited jitter that fits our data better.
```

```
qplot( factor(""), posttest, xlab="",
  position=position_jitter(width=.02))

ggplot(mydata100, aes(factor(""), posttest) ) +
 geom_jitter(position=position_jitter(width=.02)) +
 scale_x_discrete("")


# Strip plot by group.
# Note that I am increasing the jitter width from
# .02 to .08 because there is only one fourth the
# room for each graph.

qplot(workshop, posttest, data = mydata100, xlab="",
  position=position_jitter(width=.08))

ggplot(mydata100, aes(workshop, posttest) ) +
 geom_jitter(position=position_jitter(width=.08)) +
 scale_x_discrete("")


# ---Scatter Plots---

# Simple scatter plot

qplot(pretest, posttest)
qplot(pretest, posttest, geom="point")

ggplot(mydata100, aes(pretest, posttest)) +
  geom_point()

# Scatter plot connecting points sorted on x.
qplot( pretest, posttest, geom="line")

ggplot(mydata100, aes(pretest, posttest) ) +
  geom_line()

# Scatter plot connecting points in data set order.

qplot( pretest, posttest, geom="path")

ggplot(mydata100, aes(pretest, posttest) ) +
  geom_path()
```

```
# Scatter plot with skinny histogram-like bars to X axis.

qplot(pretest,posttest,
  xend=pretest, yend=50,
  geom="segment")

ggplot(mydata100, aes(pretest, posttest) ) +
  geom_segment( aes(  pretest, posttest,
                 xend=pretest, yend=50) )

# Scatter plot with jitter
qplot(q1, q4) #First without

# Now with jitter.
qplot(q1, q4, position=
  position_jitter(width=.3,height=.3))

ggplot(mydata100, aes(x=q1, y=q2) ) +
 geom_point(position=
   position_jitter(width=.3,height=.3))

# Scatter plot on large data sets

pretest2   <-
  round( rnorm( n=5000, mean=80, sd=5) )
posttest2  <-
  round( pretest2 + rnorm( n=5000, mean=3, sd=3) )
pretest2[pretest2>100] <- 100
posttest2[posttest2>100] <- 100
temp=data.frame(pretest2,posttest2)

# Small, jittered, transparent points.

qplot(pretest2, posttest2, data = temp,
  size = I(1), colour = I(alpha("black", 0.15)),
  geom = "jitter")

# Or in the next version of ggplot2:

qplot(pretest2, posttest2, data = temp,
  size = I(1), alpha = I(0.15),
  geom = "jitter")


ggplot(temp, aes(pretest2, posttest2),
```

```
  size=2, position = position_jitter(x=2,y=2) ) +
  geom_jitter(colour=alpha("black",0.15) )

ggplot(temp, aes(pretest2, posttest2)+
 geom_point(colour=alpha("black",0.15),
   position=position_jitter(width=.3,height=.3)) )

# Hexbin plots

qplot(pretest2, posttest2, geom="hex", bins=30)

ggplot(temp, aes(pretest2, posttest2))  +
  geom_hex( bins=30 )


# This works too:

ggplot(temp, aes(pretest2, posttest2))  +
  stat_binhex(bins = 30)


# Using density contours and small points.

qplot(pretest2, posttest2, data=temp,
  geom=c("point","density2d"), size = I(1) )


# geom_density_2d was renamed geom_density2d

ggplot(temp, aes( x=pretest2, y=posttest2) ) +
 geom_point( size=1 ) + geom_density2d()

rm(pretest2,posttest2,temp)

# Scatter plot with regression line,
  95% confidence intervals.

qplot(pretest, posttest,
  geom=c("point","smooth"), method=lm )

ggplot(mydata100, aes(pretest, posttest) ) +
  geom_point() + geom_smooth(method=lm)

# Scatter plot with regression line
  but NO confidence intervals.
```

```
qplot(pretest, posttest,
  geom=c("point","smooth"),
  method=lm, se=FALSE )

ggplot(mydata100, aes(pretest, posttest) ) +
  geom_point() +
  geom_smooth(method=lm, se=FALSE)

# Scatter with x=y line

qplot(pretest, posttest,
  geom=c("point","abline"),
  intercept=0, slope=1 )

ggplot(mydata100, aes(pretest, posttest) ) +
  geom_point()+
  geom_abline(intercept=0, slope=1)

# Scatter with vertical or horizontal lines

# When the book was written, qplot required the
# values to be equal. Now it does not using
# xintercept and yintercept.

qplot(pretest, posttest,
 geom=c("point", "vline", "hline"),
 xintercept=75, yintercept=75)

ggplot(mydata100, aes(pretest, posttest)) +
  geom_point() +
  geom_vline( xintercept=75 ) +
  geom_hline( yintercept=75 )


# Scatter plot with a set of vertical lines

qplot(pretest, posttest, type="point") +
  geom_vline( xintercept=seq(from=70,to=80,by=2) )

ggplot(mydata100, aes(pretest, posttest)) +
  geom_point() +
  geom_vline( xintercept=seq(from=70,to=80,by=2) )

# Scatter plotting text labels
```

```
qplot(pretest, posttest, geom="text",
  label=rownames(mydata100) )

ggplot(mydata100,
  aes(pretest, posttest,
  label=rownames(mydata100) ) ) +
  geom_text()

# Scatter plot with different
  point shapes for each group.

qplot(pretest, posttest, shape=gender)

ggplot(mydata100, aes(pretest, posttest) ) +
  geom_point( aes(shape=gender ) )

# Scatter plot with regressions fit for each group.

qplot(pretest, posttest,
  geom=c("smooth","point"),
  method="lm", shape=gender)

ggplot(mydata100,
  aes(pretest, posttest, shape=gender) ) +
  geom_smooth( method="lm" ) + geom_point()

# Scatter plot faceted for groups

qplot(pretest, posttest,
  geom=c("smooth", "point"),
  method="lm", shape=gender,
  facets=workshop~gender)

ggplot(mydata100,
  aes(pretest, posttest, shape=gender) ) +
  geom_smooth( method="lm" ) + geom_point() +
  facet_grid( workshop~gender )

# Scatter plot matrix

plotmatrix( mydata100[3:8] )

# Small points & lowess fit.
plotmatrix( mydata100[3:8], aes( size=1 ) ) +
```

```
  geom_smooth() +
  opts(legend.position="none")

# Shape and gender fits.
plotmatrix( mydata100[3:8],
  aes( shape=gender ) ) +
  geom_smooth(method=lm)

# ---Box Plots---

# box plot of one variable

qplot(factor(""), posttest,
  geom="boxplot", xlab="")

ggplot(mydata100,
  aes(factor(""), posttest) ) +
  geom_boxplot() +
  scale_x_discrete("")

# Box plot by group

qplot(workshop, posttest, geom="boxplot" )

ggplot(mydata100,
  aes(workshop, posttest) ) +
  geom_boxplot()

# Box plot by group with jitter

# First, with default jitter,
# that is too much for our small data set

qplot(workshop, posttest,
  geom=c("boxplot","jitter") )

ggplot(mydata100,
  aes(workshop, posttest )) +
  geom_boxplot() + geom_jitter()

# Again, with a smaller amount of jitter.

ggplot(mydata100,
 aes(workshop, posttest )) +
 geom_boxplot() +
```

```
 geom_jitter(position=
   position_jitter(width=.1))

# Box plot for two-way interaction.

qplot(workshop, posttest,
  geom="boxplot", fill=gender ) +
  scale_fill_grey(start = 0, end = 1)

ggplot(mydata100,
  aes(workshop, posttest) ) +
  geom_boxplot( aes(fill=gender),
    colour="grey50") +
  scale_fill_grey(start = 0, end = 1)

# Error bar plot

# This is the code for qplot.

qplot( as.numeric(workshop), posttest) +
  geom_jitter(position=
    position_jitter(width=.1))  +
  stat_summary(fun.y="mean",
    geom="smooth", se=FALSE) +
  stat_summary(fun.data="mean_cl_normal",
    geom="errorbar", width=.2)

# This is the code for ggplot.

ggplot(mydata100,
  aes( as.numeric(workshop), posttest ) ) +
  geom_jitter(size=1,
    position=position_jitter(width=.1) )  +
  stat_summary(fun.y="mean",
    geom="smooth", se=FALSE) +
  stat_summary(fun.data="mean_cl_normal",
    geom="errorbar", width=.2)

# This does away with the jitter and looks nice.

ggplot(mydata100,
  aes( as.numeric(workshop), posttest ) ) +
  geom_point(size=1)  +
  stat_summary(fun.y="mean",
    geom="smooth", se=FALSE) +
```

```
  stat_summary(fun.data="mean_cl_normal",
    geom="errorbar", width=.2)

# This uses large points for the means.

ggplot(mydata100,
  aes( workshop, posttest ) ) +
  geom_point(size=1) +
  stat_summary(fun.y="mean",
    geom="point", size=3) +
  stat_summary(fun.data="mean_cl_normal",
    geom="errorbar", width=.2)

# ---Logarithmic Axes---

# Change the variables

qplot( log(pretest), log(posttest) )

ggplot(mydata100,
  aes( log(pretest), log(posttest) ) ) +
  geom_point()

# Change axis labels

qplot(pretest, posttest, log="xy")

ggplot(mydata100,
    aes( x=pretest, y=posttest) ) +
  geom_point() +
  scale_x_log10() +
  scale_y_log10()

# Change axis scaling

# Tickmarks remain uniformly spaced,
# because scale of our data is too limited.

qplot(pretest, posttest, data=mydata100)  +
  coord_trans(x="log10", y="log10")

ggplot(mydata100,
    aes( x=pretest, y=posttest) ) +
  geom_point() +
  coord_trans(x="log10", y="log10")
```

```
# ---Aspect Ratio---

# This forces x and y to be equal.
qplot(pretest, posttest) + coord_equal()

# This sets aspect ratio to height/width.
qplot(pretest, posttest) +
  coord_equal(ratio=1/4)

#---Multiframe Plots: Bar Chart Example---

# Clears the page, otherwise new plots
# will appear on top of old.

grid.newpage()

# Sets up a 2 by 2 grid to plot into.
pushViewport(
  viewport( layout=grid.layout(2,2) )
)

# Bar Chart dodged in row 1, column 1.
myPlot <- ggplot(mydata100,
    aes(gender, fill=workshop) ) +
  geom_bar(position="stack") +
  scale_fill_grey(start = 0, end = 1) +
  opts( title="position=stack " )
print(myPlot, vp=viewport(
  layout.pos.row=1,
  layout.pos.col=1) )

# Bar Chart stacked, in row 1, column 2.
myPlot <- ggplot(mydata100,
    aes(gender, fill=workshop) ) +
  geom_bar(position="fill") +
  scale_fill_grey(start = 0, end = 1) +
  opts( title="position=fill" )
print(myPlot, vp=viewport(
  layout.pos.row=1,
  layout.pos.col=2) )

# Bar Chart dodged, given frames,
# in row 2, columns 1 and 2.
myPlot <- ggplot(mydata100,
```

```
    aes(gender, fill=workshop) ) +
  geom_bar(position="dodge")  +
  scale_fill_grey(start = 0, end = 1) +
  opts( title="position=dodge" )
print(myPlot, vp=viewport(
  layout.pos.row=2,
  layout.pos.col=1:2) )

dev.off()

#---Multiframe Scatter Plots---

# Clears the page, otherwise new plots
  will appear on top of old.
grid.newpage()

# Sets up a 2 by 2 grid to plot into.
pushViewport(
  viewport( layout=grid.layout(2,2) )
)

# Scatter plot of points
myPlot <- qplot(pretest, posttest,
  main="geom=point")
print(myPlot, vp=viewport(
  layout.pos.row=1,
  layout.pos.col=1) )

myPlot <- qplot( pretest, posttest,
          geom="line", main="geom=line" )
print(myPlot, vp=viewport(
  layout.pos.row=1,
  layout.pos.col=2) )

myPlot <- qplot( pretest, posttest,
          geom="path", main="geom=path" )
print(myPlot, vp=viewport(
  layout.pos.row=2,
  layout.pos.col=1) )

myPlot <- ggplot( mydata100,
  aes(pretest, posttest) ) +
  geom_segment( aes(x=pretest, y=posttest,
                    xend=pretest, yend=58) ) +
  opts( title="geom_segment" )
```

```
print(myPlot,
  vp=viewport(
    layout.pos.row=2,
    layout.pos.col=2 )
)


# ---Multiframe Scatter Plot for Jitter---

grid.newpage()

pushViewport(
  viewport( layout=grid.layout(1,2) )
)

# Scatter plot without
myPlot <- qplot( q1, q4,
          main="Likert Scale Without Jitter")
print(myPlot, vp=viewport(
  layout.pos.row=1,
  layout.pos.col=1) )

myPlot <- qplot(q1, q4,
     position=position_jitter(
       width=.3,height=.3),
     main="Likert scale with jitter")

print(myPlot,
  vp=viewport( layout.pos.row=1,
               layout.pos.col=2 )
)


# ---Detailed Comparison of qplot and ggplot---

qplot(pretest, posttest,
  geom=c("point","smooth"), method="lm" )

# Or ggplot with default settings:

ggplot(mydata100, aes(x=pretest, y=posttest) ) +
  geom_point() +
  geom_smooth(method="lm")
```

```
# Or with all of the defaults displayed:
ggplot() +
layer(
  data=mydata100,
  mapping=aes(x=pretest, y=posttest),
  geom="point",
  stat="identity"
) +
layer(
  data=mydata100,
  mapping=aes(x=pretest, y=posttest),
  geom="smooth",
  stat="smooth",
  method="lm"
) +
coord_cartesian()
```