Pretty plots with `par`

BISC 888–1 Simon Fraser University

Sean C. Anderson
sean@seananderson.ca

November 6, 2013

Base plotting functions let you draw with data. If you can imagine it, you can make it with base graphics functions in R.

The price you pay for this flexibility is a lack of structured grammar like `ggplot2`, a lot of sometimes-complex code, and a lot of time coding.

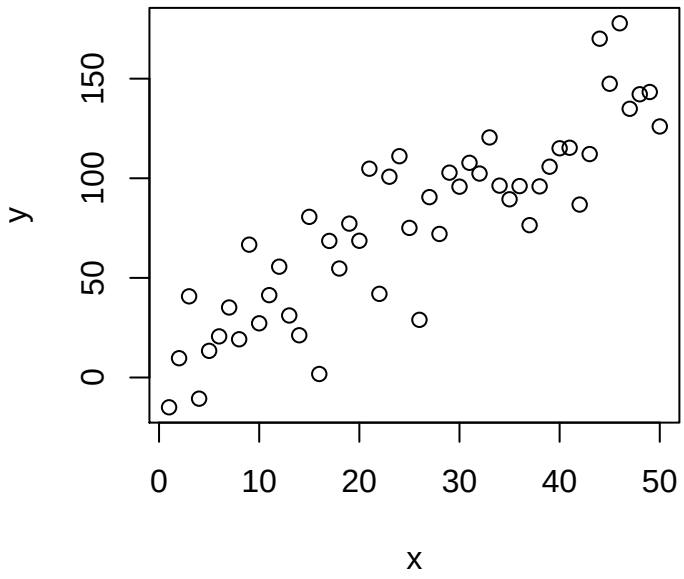
Explore data rapidly with `ggplot2`, and if you reach the limits of what you can do with `ggplot2`, or you find a plot you want to customize beyond what you can easily do with `ggplot2`, then create it with base graphics.

1 Making one panel look good

In this section, we're going to start with a default base graphics plot and thoroughly customize it.

Let's start with a basic scatterplot. We'll generate some data and make a default plot.

```
set.seed(2)
x <- seq_len(50)
y <- x * 3 + rnorm(50, sd = 20)
plot(x, y)
```



First, let's set up `par`. We'll cover some common adjustments here, but there are many options we won't touch on. It is well worth reading the help file `?par` frequently. If you want to be able to quickly produce highly-customized R graphics, you will need to be intimately familiar with the `par` options.

We're going to reduce the character expansion value `cex`. This will make everything a bit smaller. The `cex` value you choose will depend on the aesthetics of the plot and the output (e.g. PDF) dimensions. Typically, values around 0.8 will work. Lower values can be useful if you are working with small plots or plots with lots of panels.

```
par(cex = 0.8)
```

We're also going to set `mgp`, which controls the spacing of the axis titles, axis number labels, and the axis line itself. The first two values (axis title and label and label spacing) are useful to set.

```
par(mgp = c(2, 0.6, 0))
```

We're going to reduce the tick length by setting `tck`.

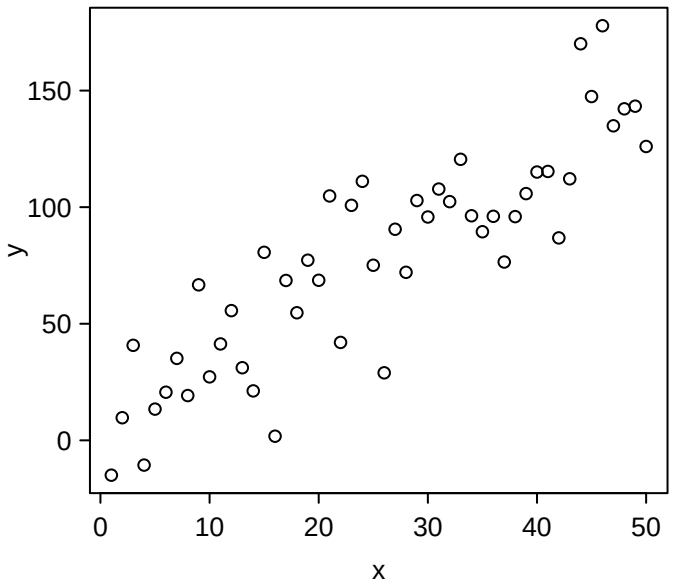
```
par(tck = -0.02)
```

We're going to make all the axis text horizontal by setting `las` (label axis style).

```
par(las = 1)
```

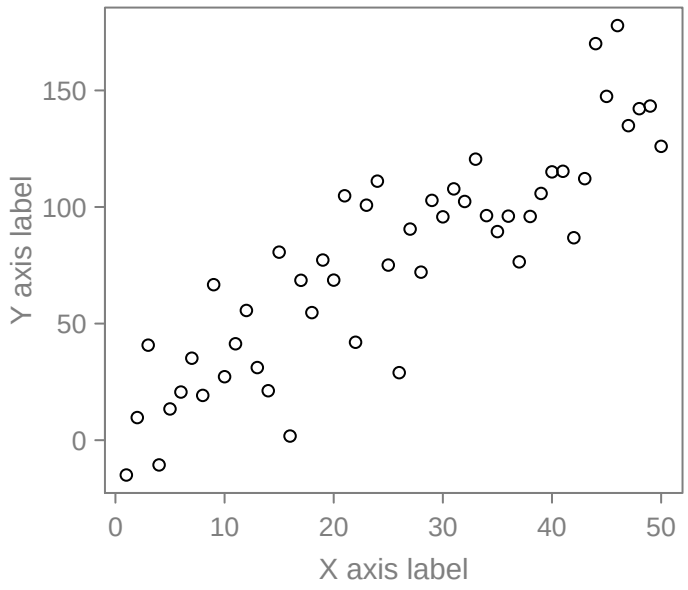
And now remake the plot:

```
plot(x, y)
```



We're getting there. Now let's emphasize the data by lightening the axes and axis labels. There are multiple ways to do this, but I'm going to show you the most flexible way: starting with a blank plot and building it up piece by piece:

```
plot(x, y, type = "n", axes = FALSE, ann = FALSE)
points(x, y)
axis_col <- "grey50"
box(col = axis_col)
axis(1, col = axis_col, col.axis = axis_col)
axis(2, col = axis_col, col.axis = axis_col)
mtext("X axis label", side = 1, col = axis_col, line = 2.0, cex = 0.9)
mtext("Y axis label", side = 2, col = axis_col, las = 3,
  line = 2.2, cex = 0.9)
```



2 Adding a colour dimension

This is going to be much more involved than in `ggplot`, but bear with me. It's also very flexible. First we're going to bring in a colour palette. Then we're going to match cut that colour palette up according to our data. Then we're going to join that colour palette to the data we want to plot.

`RColorBrewer` is an excellent package for colour palettes. We're going to start with one of those. Since the data we're going to colour is continuous we're going to pick a continuous colour scale.

First, let's create some fake data we want to colour by. We'll call this column `z`. We'll combine, `x`, `y`, and `z` in a data frame.

```
d <- data.frame(x, y, z = 50:1)
```

Now, we'll load the `RColorBrewer` library and read in a palette.

```
library(RColorBrewer)
pal <- brewer.pal(9, "YlOrRd")
```

Now, we're going to create a data frame that matches colours with values of `z` that we want to assign colours at.

```
pal_df <- data.frame(pal = pal, cuts = seq(min(d$z), max(d$z),
    length.out = length(pal)), stringsAsFactors = FALSE)
pal_df
```

```
##       pal   cuts
## 1 #FFFFCC  1.000
## 2 #FFEDA0  7.125
## 3 #FED976 13.250
## 4 #FEB24C 19.375
## 5 #FD8D3C 25.500
## 6 #FC4E2A 31.625
## 7 #E31A1C 37.750
## 8 #BD0026 43.875
## 9 #800026 50.000
```

And we'll use `findInterval` to find the rows of cut values that match the values of `z`:

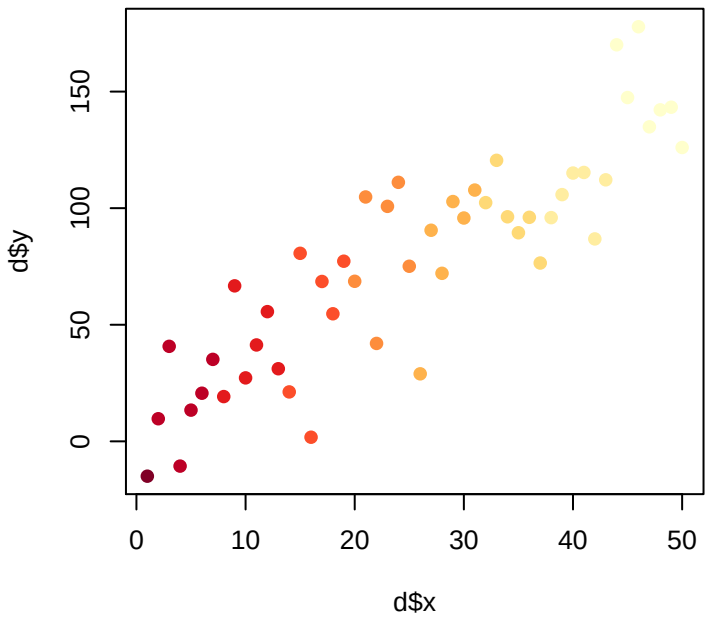
```
pal_indices <- findInterval(d$z, pal_df$cuts)
```

Then we'll use those indices to pick out the colour palette values:

```
d$col <- pal_df$pal[pal_indices]
```

Now we can plot with those colours:

```
plot(d$x, d$y, col = d$col, pch = 19)
```



3 Adding lines and polygons

We're going to work through an example adding a linear model fit and shaded confidence intervals to our scatter plot.

There are many ways to do this (and for this simple case, many much simpler ways), but this is a general example that can work across nearly any type of linear or non-linear model with minimal modification.

First, we'll fit the model:

```
m <- lm(y ~ x, data = d)
```

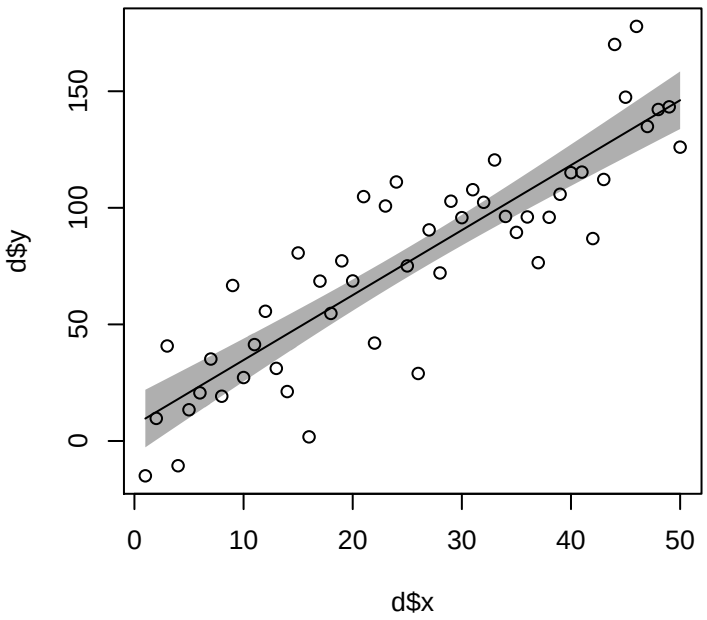
Now, we'll create a new prediction data frame and generate values of `x` that we want to predict across. We'll predict across a sequence of `x` values that are close enough to each other that our confidence interval curve will look smooth.

We'll get both the fitted line and the 95% confidence intervals by adding and subtracting 1.96 times the standard error on the predictions.

```
pred_df <- data.frame(x = seq(min(x), max(x), length.out = 100))
pred_df$fit <- predict(m, newdata = pred_df)
se <- predict(m, newdata = pred_df, se = TRUE)$se.fit
pred_df$lower <- pred_df$fit + 1.96 * se
pred_df$upper <- pred_df$fit - 1.96 * se
```

Then we'll plot the data again and add the line with `lines` and the confidence interval with `polygon`. The first argument to `polygon` is a sequence of x-values from left to right and the second argument is a sequence of y-values from left to right. Therefore, some of the vectors need to be reversed with the `rev` function. We'll also use the `with` function, which allows us to avoid writing out `pred_df$` repeatedly. It allows us to access the column names directly in that line of code.

```
plot(d$x, d$y)
with(pred_df, lines(x, fit))
with(pred_df, polygon(c(x, rev(x)), c(lower, rev(upper)),
  border = NA, col = "#00000050"))
```


d$y
0
50
100
150
0
10
20
30
40
50
d$x