

Handout 3: Introduction to **R**

Dave Armstrong
University of Wisconsin – Milwaukee
Department of Political Science

e: armstrod@uwm.edu

w: www.quantoid.net/ICPSR.php

Contents

1	Graphics Philosophies	2
2	The Plot Function	2
2.1	getting familiar with the function	2
2.2	Default Plotting Methods	4
2.3	Controlling the Plotting Region	6
2.4	Starting with a blank canvas	6
2.5	More Complicated Scatterplots	8
2.5.1	Adding a Legend	10
2.6	Identifying Points in the Plot	10
2.7	Adding in the Regression Line	11
2.8	Side-by-side Barplot	12
3	Fine Control of the Plotting Region with par	13

1 Graphics Philosophies

In **R** there are basically two philosophies about how graphs can be made.

- Painting - the traditional graphics system operates like painting, where elements can be added and layered onto a blank canvas. This is appealing because it allows you to build in the elements you see fit.
- Dominos - the lattice system operates rather more like dominos. Here, you spend a lot of time setting up the graphic command and then all of the action happens at once.

As you can imagine, these two different systems offer quite different solutions to creating high quality graphics. Depending on what exactly you're trying to do, some things are more difficult, or impossible, in one of these systems, but not in the other. We will certainly see examples of this as the course progresses. I have had a change of heart about these various systems recently. Originally, I did everything I possibly could in the traditional graphics system and then only later moved to the lattice system. This probably makes sense as the lattice system requires considerably more programming, but does lots more neat stuff.

We are going to spend our time today talking about the traditional graphics system. We will spend some time later talking about what lattice graphs can do and why we might want to use them, but for now, it's just the traditional graphics.

2 The Plot Function

For the most part, in the traditional graphics system, graphs are initially made with the `plot` function, though there are others, too. Then additional elements can be added as you see fit.

2.1 getting familiar with the function

Let's take a look at the help file for the `plot` command.

```
x: the coordinates of points in the plot. Alternatively, a
    single plotting structure, function or _any R object with a
    'plot' method_ can be provided.
```

```
y: the y coordinates of points in the plot, _optional_ if 'x' is
    an appropriate structure.
```

```
...: Arguments to be passed to methods, such as graphical
    parameters (see 'par'). Many methods will accept the
    following arguments:
```

```
'type' what type of plot should be drawn. Possible types are
```

- * 'p' for *p*oints,
- * 'l' for *l*ines,
- * 'b' for *b*oth,
- * 'c' for the lines part alone of 'b',
- * 'o' for both *o*verplotted,
- * 'h' for *h*istogram like (or high-density) vertical lines,
- * 's' for stair *s*teps,
- * 'S' for other *s*teps, see Details below,
- * 'n' for no plotting.

All other 'type's give a warning or an error; using, e.g., 'type = "punkte"' being equivalent to 'type = "p"' for S compatibility.

'main' an overall title for the plot: see 'title'.

'sub' a sub title for the plot: see 'title'.

'xlab' a title for the x axis: see 'title'.

'ylab' a title for the y axis: see 'title'.

'asp' the y/x aspect ratio, see 'plot.window'.

Now, we can load the Duncan data again and see how the plotting function works. The two following commands produce the same output within the plotting region, but have different axis labels.

```
plot

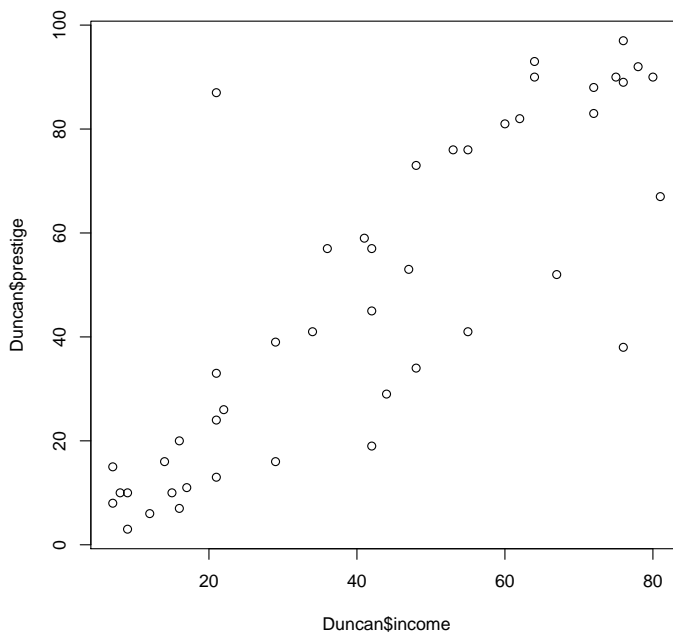
```
prestige ~ income, data=Duncan)
plot(Duncan$income, Duncan$prestige)
```


```

There are a couple of interesting features here.

- You can either specify the plot with a formula as the first argument, $y \sim x$ or with x, y as the first two arguments. The `data` argument only exists when you use the former method. Using the latter method, you have to provide the data in `dataset$variable` format, unless the data are attached.

Figure 1: Scatterplot of Income and Prestige from the Duncan data



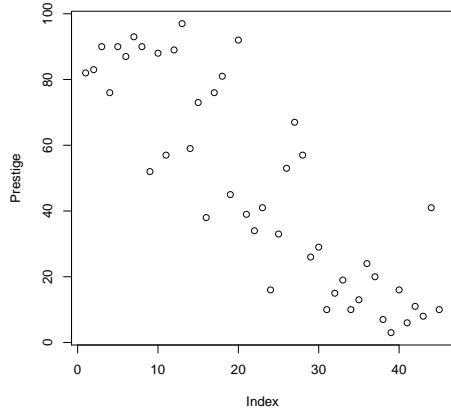
- Whatever names you have for **x** and **y** will be printed as the x- and y-labels. These can be controlled with the `xlab` and `ylab` commands.
- The plotting symbols, by default are open circles. This can also be controlled with the `pch` option.

2.2 Default Plotting Methods

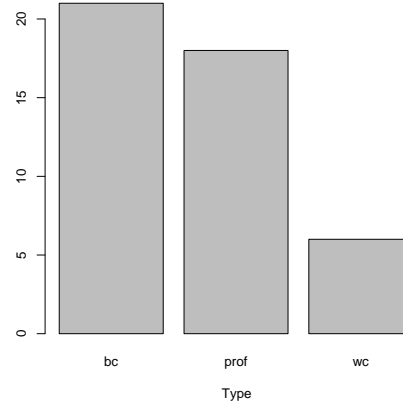
In **R**, there are default methods for plotting all types of variables. By *default method* I simply mean that **R** looks at the context in which you're asking for a graph and then makes what it thinks is a reasonable graph given the different types of data. All of the figures in 2 were called simply by using the `plot` command. **R** figured out by the types of variables being used what plot was most appropriate.

Figure 2: Default Plotting Methods

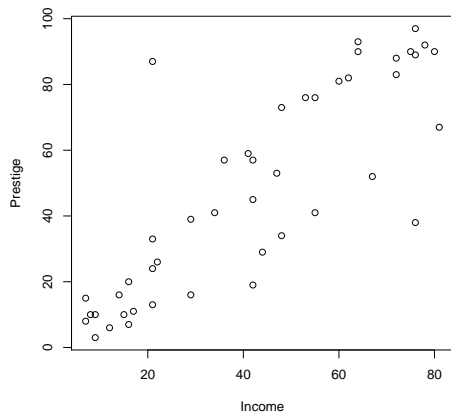
(a) One Numeric - scatterplot (with index as the x -axis)



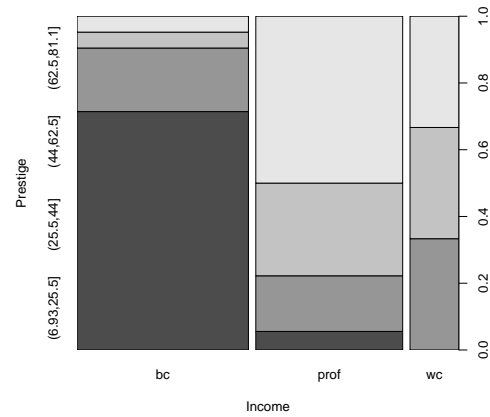
(b) One factor - histogram



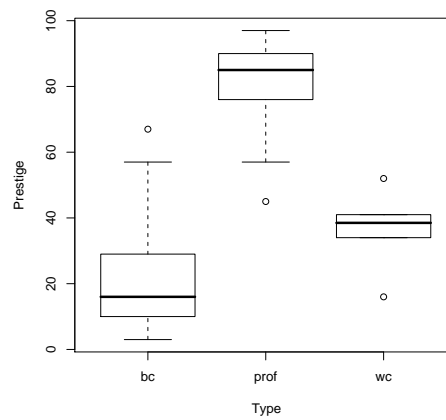
(c) Two Numeric Variables - scatterplot



(d) Two Factors - mosaic plot



(e) One Numeric, One Factor - boxplot



2.3 Controlling the Plotting Region

There are a number of commands we can use to control the size and features of the plotting region.

- We can control the limits of the x- and y-axis with `xlim` and `ylim`, respectively. Here, the limits are specified with a vector of indicating the desired minimum and maximum value of the axis.

```
plot(prestige ~ income, data=Duncan, xlim=c(0,100))
```

2.4 Starting with a blank canvas

In **R**, you can open a plotting window and set its dimensions without actually making the points appear in the space. You can do this by specifying `type='n'`. You can also remove the axes from the space, by issuing the command `axes=F`. Finally, you could remove any axis labels by adding the commands `xlab=''`, `ylab=''`. The command, then, would look something like this:

```
plot(prestige ~ income, data=Duncan, type="n", xlab='', ylab='', axes=F)
```

This will open a graphics window that is completely blank. One reason that you may want to do this is to be able to control, more precisely, the elements of the graph. Let's talk about adding some elements back in.

We can add the points by using the `points` command. You can see the arguments to the points command by typing `help(points)`. The first two arguments to the command are `x` and `y`. You can also change the plotting symbol and the color.

- Plotting Symbols are governed by the `pch` argument. Below is a description of some common plotting symbols.

○	△	+	×	◇	▽	⊠	*	⊕	⊗	⊘	⊙	⊚	⊛	■	●	▲	◆	●	●
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

- Color - Colors are controlled by the `col` command. (see help for `rainbow`, `colors`). The colors can be specified by `col = 'black'` or `col = 'red'`.
- Character Expansion is controlled by `cex`. This controls the size of the plotting symbols. The default is 1. Numbers in the range (0,1) make plotting symbols smaller than the default and values > 1 make the plotting symbols bigger than they would be otherwise.

There are lots of interesting things we can do with these. Let's start to build up a scatterplot of prestige and income from the blank slate we just spoke about.

```
plot(prestige ~ income, data=Duncan, type='n',  
     xlab='', ylab='', axes=F)
```

```
points(Duncan$income, Duncan$prestige, pch=16, col="blue")
```

We can also add the axes back in if we want with the command `axis`. There are a number of important arguments to `axis`, they are:

- `side` tells **R** which side you want the axis on, the options are 1=bottom, 2=left, 3=top, 4=right.
- `at` controls the points at which the tick marks are made on the axis. This has to be a vector of numbers which you can either put in manually or make with a sequence command `seq`. If `at` is omitted, then **R** decides how many tick marks there should be and where they should go.
- `labels` this is a vector of labels for the tick marks that must have the same number of elements as the `at` argument. If this is omitted, but `at` is specified, then the labels are just the numbers provided in `at`.

Let's add back in the x and y axes.

```
axis(1)  
axis(2)
```

What if we only want 4 tick-marks on each axis, evenly spaced from the minimum to maximum values? First, we need to make the `at` vectors for each axis. We can do this with the `seq` command which makes sequences of desired length.

```
x.at <- seq(min(Duncan$income), max(Duncan$income), length=4)  
y.at <- seq(min(Duncan$prestige), max(Duncan$prestige), length=4)  
plot(prestige ~ income, data=Duncan, type='n',  
     xlab='', ylab='', axes=F)  
points(Duncan$income, Duncan$prestige, pch=16, col="blue")  
axis(1, at=x.at)  
axis(2, at=y.at)
```

This doesn't look very pretty because there are lots of decimal places in the numbers. We can cut these off with the `round` command:

```
x.at <- round(seq(min(Duncan$income), max(Duncan$income), length=4), 0)  
y.at <- round(seq(min(Duncan$prestige), max(Duncan$prestige), length=4), 0)  
  
plot(prestige ~ income, data=Duncan, type='n',  
     xlab='', ylab='', axes=F)  
points(Duncan$income, Duncan$prestige, pch=16, col="blue")  
  
axis(1, at=x.at)  
axis(2, at=y.at)
```

We can add the box around the plotting region back in with the `box()` command. This command takes no arguments. Simply type `box()` at the command prompt and hit enter.

We can put the axis labels back in using the `mtext` command (which places text in the margins of the plot). This command has a number of useful arguments as well.

- `text` is the text string you want to include in the margin.
- `side` specifies the axis on which you want to put the text and follows the same convention as `axis`.
- `line` specifies the line on which you want to place the text. In the margins there are a number of lines. Line 0 is usually right at the edge of the plotting region. The tick labels are usually placed in line 1. If you want to separate the axis labels from the tick labels, you should put the text on line 3.

```
mtext("% males earnings > $3500/year", 1, line=3)
```

```
mtext("% of NORC raters indicating profession as\n 'good' or 'excellent'",  
      2, line=2)
```

One interesting thing to notice here is that in the second command, I split the text over two lines using the `\n` command which indicates a “new line”. Further the `line` argument states the line on which the text starts and works out. So, since my text goes over two lines, if I specify `line=3`, the text will go on lines 3 and 4. If I specify `line=2`, the text will go on lines 2 and 3.

Finally, we can add a title to the plot with the `title` command.

```
title("Prestige versus Income\n (Duncan data)")
```

2.5 More Complicated Scatterplots

Above, we simply plotted the points, so only two variables are involved here. What if we wanted to include a third variable? We could include information relating to occupation type by coloring the points different for different types of occupations and perhaps using different plotting symbols. To do this, we would have to specify the `points` command differently, but could use the rest of the commands we specified above to make the plot.

```
x.at <- round(seq(min(Duncan$income), max(Duncan$income), length=4), 0)  
y.at <- round(seq(min(Duncan$prestige), max(Duncan$prestige), length=4), 0)  
plot(prestige ~ income, data=Duncan, type='n',  
     xlab='', ylab='', axes=F)  
axis(1, at=x.at)  
axis(2, at=y.at)  
box()
```

Notice that the `points` command was left out here, so we just have a blank plot area. How, then, can we plot the points differently for professionals, blue collar workers and white collar workers? Well, think back to yesterday’s class. How could we filter the dataset so it only contained professional workers, that is the key.

```
Duncan_prof <- Duncan[Duncan$type == "prof", ]
Duncan_bc <- Duncan[Duncan$type == "bc", ]
Duncan_wc <- Duncan[Duncan$type == "wc", ]
```

Now, we have three different datasets, each only with one type of occupation. We can then simply use three `points` commands to include points from each dataset independently.

```
points(Duncan_bc$income, Duncan_bc$prestige, col='red',
       pch=1)
points(Duncan_prof$income, Duncan_prof$prestige, col='black',
       pch=2)
points(Duncan_wc$income, Duncan_wc$prestige, col='blue',
       pch=4)
```

There is actually a way to do this with one `points` command if we were really keen. It requires a bit of setup and some imagination, but let me point us in the right direction. First, let's make two vectors, one for the plotting symbols we want to use and one for the colors we want to use:

```
pch.vec <- c(1,2,4)
col.vec <- c('red', 'black', 'blue')
```

Now, let me take us one step further, we need a numeric (rather than factor) version of `type` from the Duncan data:

```
type_num <- as.numeric(Duncan$type)
```

Finally, let me remind you that we can access any element of `pch.vec` or `col.vec` as follows:

```
> pch.vec[1]
[1] 1
> pch.vec[2]
[1] 2
> pch.vec[3]
[1] 4
>
> col.vec[1]
[1] "red"
> col.vec[2]
[1] "black"
> col.vec[3]
[1] "blue"
```

Anyone see it yet? Well, we can do the following:

```
points(Duncan$income, Duncan$prestige, pch=pch.vec[type_num], col=col.vec[type_num])
```

Basically, what you are doing is you are specifying the plotting character and color for each point that's being plotted. You're doing this by allowing our vector of the three plotting types and three colors we want to use to be indexed by the occupation type. Nifty, right!? I think this really gives you a taste of the power of **R** graphics.

2.5.1 Adding a Legend

We know what the points mean, but we can't very well expect other people to know this unless we tell them. There is a function called `legend` that allows us to make a legend and stick it in the plot. The `legend` command has a number of arguments that should be specified.

- The first argument is the location of the plot. This can be specified in a number of ways. If you provide `x` and `y` coordinate values (with `x=#` and `y=#`), then this gives the coordinates for the top-left corner of the box containing the legend information. Otherwise, you can specify the location with `topleft`, `top`, `topright`, `right`, `bottomright`, `bottom`, `bottomleft`, `left` and `R` will put the legend near the edge of the plot in this position.
- The `legend` argument gives the text you want to be displayed for each point/line you're describing.
- The point symbol and color are given by a vector to `pch` and `col`.
- If instead of points, you have lines, you can give the argument `lty` with the different line types being used (more on this later).
- `inset` gives the fraction of the plot region between the edges of the legend and the box around the plotting region (default is 0).

We can include a legend in our plot as follows:

```
legend("bottomright", c("Blue-collar", "Professional", "White-collar"),
      pch=pch.vec, col=col.vec, inset=.01)
```

2.6 Identifying Points in the Plot

Points in the plot can be identified with the function `identify`. The `identify` will print a label next to points that you click on giving an indication of which point exactly is plotted. The important arguments are:

- Location is given by an `x` and `y` variable. These should be the same `x` and `y` variables you used to make the plot.
- The labels, given with the `label` option will provide `R` with text to print by each identified point.
- `n` gives the number of points we want to identify.

Let's try to identify the two most outlying points on our graph.

```
identify(x=Duncan$income, y=Duncan$prestige,
        labels=rownames(Duncan), n=2)
```

You can see here that the `minister` has more prestige on average than other jobs with similar percentage of males making over \$3500 and that the `conductor` (railroad) has less prestige than its income would suggest.

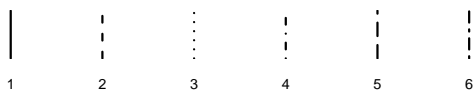
2.7 Adding in the Regression Line

We can see that the relationship between these two variables seems linear. Thus, we might want to stick in the OLS regression line. We could do this by first running the linear model and assigning its output to an object, then by plotting the line implied by that model. The command `abline` plots a line with a given intercept and slope. You can either specify the intercept and slope with the `a` and `b` arguments, or you can provide the command a (bivariate) linear model object and it will automatically take the intercept and slope parameters from there.

```
mod <- lm(prestige~income, data=Duncan)
abline(mod)
```

There are a number of ways to control how the line looks.

- `lty` controls the line-type. Below is a brief presentation of a few different line-types:



- `col` allows the line to be colored differently and works the same as above.
- `lwd` controls the width of the line. The default is 1 with numbers in the range (0,1) indicating thinner lines than the default and numbers > 1 indicating thicker lines than the default.

It looks like the `type` variable might be important because it appears that the line does not really run through the cloud of points. Adding the `type` variable will allow the intercept to differ for each type. To do this, we could estimate another model that includes occupation and then specify the slope and intercept ourselves as follows:

```
mod2 <- lm(prestige~type + income, data=Duncan)
b <- mod2$coef
a_bc <- b[1]
a_prof <- b[1] + b[2]
a_wc <- b[1] + b[3]
```

```
abline(a=a_bc, b=b[4], col=col.vec[1], lty=1)
abline(a=a_prof, b=b[4], col=col.vec[2], lty=2)
abline(a=a_wc, b=b[4], col=col.vec[3], lty=3)
```

Notice, that there are now three lines, each with the same slope, but different intercepts and generally, this looks “better” than the one-line model.

Finally, let’s say that we want to show what the prediction is for `minister`. To do this, we need to use the fitted values and the observation’s original values. We can also use the `arrows` function to draw a line down to the regression line from the point and then over to the y-axis from the regression line to indicate the prediction.

First, let’s look at the point’s values.

```

> Duncan[rownames(Duncan) == "minister", ]
      type income education prestige
minister prof      21         84      87

> mod2$fitted[names(mod2$fitted) == "minister"]
minister
54.05111

```

Now, we want to draw an arrow from the point (21,87) to the fitted value for `minister` on the y-axis, but the same x value (21). Using the `arrows` command, we can do this as follows:

```

arrows(21,87,21,54.05111, code=2, col="gray75", length=.1)
arrows(21,54.0511, 4.04, 54.0511, code=2, col="gray75", length=.1)

```

2.8 Side-by-side Barplot

R is especially good if you are very clear about what you want. It is less good for those grasping in the dark for the right method / data format, etc... A simple, but good, example of this is the barplot. The `barplot` command plots a set of heights as bars. The function will plot a separate set of bars for each row of the data. If there is only one row, then only one set of bars get plotted.

```

library(car)
data(Ornstein)
sectors <- table(Ornstein$sector)
barplot(sectors, las=2)

```

the `las=2` argument above makes the tick labels perpendicular to the axis. If you want to get proportions instead of frequencies, you can just make the table of frequencies into a table of proportions.

```

sectors.prop <- sectors/sum(sectors)
barplot(sectors.prop, las=2)

```

If we wanted to break this out by nation, we could do as follows:

```

sec.list <- by(Ornstein$sector, list(Ornstein$nation), table)
sec.tab <- do.call(rbind, sec.list)
barplot(sec.tab, col=c("red", "blue", "green", "orange"), beside=T)
legend(locator(1), legend=levels(Ornstein$nation), fill=c("red", "blue", "green", "orange"))

```

Again, if we wanted to plot proportions instead of frequencies, we need to calculate proportions in each row. We could do this “by hand”, for example with `sec.tab[1,]/sum(sec.tab[1,])`, but there is a function called `prop.table` which can do this for us:

```

sec.prop <- prop.table(sec.tab, 1)
barplot(sec.prop, beside=T, col=c("red", "blue", "green", "orange"))
legend(locator(1), legend=levels(Ornstein$nation), fill=c("red", "blue", "green", "orange"))

```

3 Fine Control of the Plotting Region with `par`

In the above command, how did I know the minimum value on the x-axis was 4.04? I knew this because it and a bunch of other interesting parameters of the plotting region are stored in and can be manipulated with `par()`. By typing `help(par)` you can see what sorts of things can be manipulated and by typing `par()` you can see how all of those parameters are currently set.

If we want to set, for example, the number of lines in the margins, close the plotting window and then issue the following command, `par(mar = c(5,5,5,5))`. This will put five lines on each side of the graph instead of the default (which I think is 5,4,4,2). This will open up a new window with the desired properties.

something