

# Handout: Introduction to **R**

Dave Armstrong  
University of Wisconsin – Milwaukee  
Department of Political Science

e: [armstrod@uwm.edu](mailto:armstrod@uwm.edu)  
w: [www.quantoid.net/ICPSR.php](http://www.quantoid.net/ICPSR.php)

July 25, 2011

## Contents

<b>1</b>	<b>Very Basic Function Writing</b>	<b>2</b>
<b>2</b>	<b>Lattice Graphics</b>	<b>2</b>
2.1	Plotting by groups . . . . .	3
2.2	Other Lattice Graphs . . . . .	7
2.2.1	Histograms . . . . .	7
2.2.2	Density Plots . . . . .	8
2.2.3	Histograms with Density Overlay . . . . .	9

# 1 Very Basic Function Writing

Lattice graphs often require us to write functions to do anything but the simplest things in plots. This is not especially burdensome, but it will be worth seeing how functions work first. We define a function as follows:

```
my.function <- function(x){  
do something with or to x  
}
```

Here, `x` is just a place-holder and wherever it shows up in the function, substitute the value(s) you provide to the function as its argument. For example, let's consider writing a function to calculate the mean.

```
my.mean.fun <- function(fred){  
sum(fred, na.rm=T)/sum(!is.na(fred))  
}  
> my.mean.fun(Duncan$income)  
[1] 41.86667  
> mean(Duncan$income)  
[1] 41.86667
```

Notice that the argument here is `fred`. I will generally not have a variable named `fred`, but that doesn't matter. In the function, `fred` stands in for whatever variable I put as an argument to the function, in this case `Duncan$income`. Everywhere in the function you see `fred`, `Duncan$income` gets substituted and the calculations proceed. Now, on to lattice graphs.

# 2 Lattice Graphics

While you can certainly make lattice graphs for the same types of data that we made traditional graphics, the real strength of lattice graphs is with dependent data (i.e., data organized by group). Examples of this would be time-series cross-sectional data (e.g., countries/states/individuals over time) or hierarchically structured data (e.g., students within classrooms within schools). Lattice graphs perform a type of repeated calculation - they make the same plot for each group and present them all in a very nice-looking display.

The work-horse for lattice graphs is the command `xyplot`. This is the lattice analog to the `plot` command in the traditional graphics environment. To make the commands available, you first have to load the lattice library:

```
library(lattice)
```

Now, let's just make a scatterplot of income and prestige from the Duncan data.

```
library(car)  
data(Duncan)  
xyplot(prestige ~ income, data=Duncan)
```

This plot looks very similar, but it has a couple of different defaults from the `plot` command. Specifically, the default color for the plotting symbols is a shade of blue and there are tick marks on each axis, though they're only labeled on the bottom and left axes. Many of the same `par` arguments work here, so we can change the plotting symbol and color if we want with `pch` and `col`.

```
xyplot
```

## 2.1 Plotting by groups

Now, let's consider generating this plot, but by the groups defined by occupation type:

```
xyplot
```

Notice that the only difference here is the `|` (the pipe character) separating `income` from the grouping variable `type`. I also changed the plotting symbol back to an open circle. When a grouping variable is used, **R** makes the first plot in the lower left-hand corner and then fills in the rows of the plotting region until it's done. If you want to change this behavior (so it fills in the upper left-hand cell first), you can issue the argument `as.table=T`.

```
xyplot
```

Now, let's try to remake the same graph that we made before - with different plotting symbols based on type. We could make the original plot (without lines or a legend as follows):

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

xyplot
```

Remember, that one way we could do this with `plot` was to create a vector of plotting symbols - one for each observation. This is what I've done here. Adding the lines gets more complicated. The complication here comes from the fact that in lattice graphs, you must do everything at once. To do this, you have to define a function (hence the reason the function-writing day came before the lattice day).

Here, we need to specify the panel function argument to `xyplot`. The panel function is a function of the `x` and `y` arguments to the `xyplot` command. We could remake the graph above by specifying the following panel function:

```
xyplot
```

Inside the panel function, we use sub-functions that generate the points, lines, etc... that we require to make the graphs. In this case, we used the function `panel.points` which adds points to each panel (here there is just one) according to the grouping variable and the values of `x` and `y` (more on this a bit later). There are a bunch of these types of panel utilities, a listing of many of them is below:

```
panel.lines(...)
panel.points(...)
panel.segments(...)
panel.text(...)
panel.rect(...)
panel.arrows(...)
panel.polygon(...)
```

These generally work in a similar fashion to their traditional graphics counterparts.

If you think back to lecture 3, we made a plot that had three different lines in it. We can do that here in a similar fashion. We still need to know the slopes and intercepts from the model and can get those 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]
```

We can then use the `panel.abline` sub-command in the panel function to include a line:

```
xyplot(prestige ~ income, data=Duncan, panel=function(x,y){
  panel.points(x,y, pch=pch.vec[as.numeric(Duncan$type)],
    col=col.vec[as.numeric(Duncan$type)])
  panel.abline(a=a_bc, b=b[4], lty=1, col="black")
  panel.abline(a=a_prof, b=b[4], lty=2, col="red")
  panel.abline(a=a_wc, b=b[4], lty=3, col="blue")
})
```

Notice here that `panel.abline` works the same as `abline`. The difference is that the former works inside a panel function and the latter works on its own to add something to an existing plot.

Adding a legend to the graph is a bit trickier here. It requires us to set up a function ahead of time and then we can include that function as an argument to the graphing command. To specify how the legend looks, we need to use the `key` argument. This is a list that contains information about the text, points and lines displayed. The arguments to `key` that you may want to specify are `space`, `text`, `lines` and `points`. Let's take a look at the key we'll use for our plot and we can discuss:

```

key=list(space="top",
         text=list(c("Blue Collar", "Professional", "White Collar")),
         points=list(pch=pch.vec, col=col.vec),
         lines=list(lty=c(1,2,3), col=col.vec))

```

The argument to `key` is a list. The list has a number of elements. The first is `space` which tells **R** where you want to put the legend. I picked top because I think that's less intrusive than right or left and I like it better than on the bottom, but you can change this to either `'left'`, `'right'` or `'bottom'` to suit your taste. The second element here is `text`, which is a list with a vector of the text we want to print. You could also control the color of the text in here if you wanted. The next element is `points` - which is a list with arguments controlling how the points get printed. Here I am telling **R** to use the `pch.vec` vector of plotting symbols and the `col.vec` vector of colors to control the points. Finally, I specify the element `lines` which is a list of arguments controlling the look of the lines. Here I tell **R** to use the line types 1, 2, and 3 and the same colors as the points. The whole plotting function now looks as follows:

```

xyplot

```
prestige ~ income, data=Duncan, panel=function(x,y){
  panel.points(x,y, pch=pch.vec[as.numeric(Duncan$type)],
              col=col.vec[as.numeric(Duncan$type)])
  panel.abline(a=a_bc, b=b[4], lty=1, col="black")
  panel.abline(a=a_prof, b=b[4], lty=2, col="red")
  panel.abline(a=a_wc, b=b[4], lty=3, col="blue")
},
key=list(space="top",
         text=list(c("Blue Collar", "Professional", "White Collar")),
         points=list(pch=pch.vec, col=col.vec),
         lines=list(lty=c(1,2,3), col=col.vec))
)

```


```

The ordering of the arguments `points`, `text` and `lines` matters in the `key` list. That is not to say that one is right and another wrong, rather the points, lines and text print in the order they are shown in the list. So,

```

key=list(space="top",
         text=list(c("Blue Collar", "Professional", "White Collar")),
         points=list(pch=pch.vec, col=col.vec),
         lines=list(lty=c(1,2,3), col=col.vec))

```

will produce a slightly different looking output than

```

key=list(space="top",
         points=list(pch=pch.vec, col=col.vec),
         text=list(c("Blue Collar", "Professional", "White Collar")),
         lines=list(lty=c(1,2,3), col=col.vec))

```

We can also exert some finer control over multi-panel displays. This is where things get a little weird, though. The `panel` function tells **R** what to do in each panel for the variables `x` and `y` by groups. So, let's go back to the argument above. It is actually a bit more difficult to control the plotting symbols in each panel, though by splitting the points into different panels, that makes it less useful or necessary to plot points in different symbols.

The one thing we might want to do is add back in the line specific to each group. This is difficult because you have to add a different line to each panel. What is useful to know is that you can access the panel number with the command `packet.number()`. So you can see how this works, I've had **R** print the packet number in each panel.

```
xyplot

```
prestige ~ income | type, data=Duncan, as.table=T,
  panel=function(x,y,subscripts){
    panel.text(50,50, paste("packet # = ", packet.number(), sep=""), cex=2)
  }
)
```


```

From this, you can see that blue collar jobs are the first packet `packet.number()`=1, professionals are the second packet and white collar jobs are the third. If at this point, you can see what we need to do, then you've really caught on quickly. You should definitely not feel bad if you're still not quite clear on what we would need to do to get different lines into each panel. To do this, we need to have a vector of three intercepts - one for each panel. The slope, if you remember is the same. Let's do this:

```
ints <- c(a_bc, a_prof, a_wc)
slope <- b[4]
```

I am just using the ones defined above. Now there is a vector called `ints` that has three elements and I'm defining the slope as `b[4]` (the fourth element of the coefficient vector) for transparency's sake. Then, we can do the following:

```
xyplot

```
prestige ~ income | type, data=Duncan, as.table=T,
  panel=function(x,y){
    panel.points(x,y, pch=1, col="black")
    panel.abline(a=ints[packet.number()], b=slope)
  }
)
```


```

Notice here that we've indexed the vector `ints` with the `packet.number()`. So, for blue collar jobs, it uses `ints[1]`, for professional jobs it uses `ints[2]` and for white collar jobs, it uses `ints[3]`.

One more thing you might need to know is how **R** identifies the observations in each panel and how you can access those values. **R** identifies the observations in each panel with the `subscripts` macro. You can have access to these by making `subscripts` one of the arguments to the panel function. This is not particularly interesting for the graphs we've made above, but imagine that we also want to add prestige vs. education to the plot, so we have prestige vs. education as one set of points and prestige vs income as another both in the same panel.

```
xyplot

```
prestige ~ income | type, data=Duncan, as.table=T,
  panel=function(x,y,subscripts){
    panel.points(x,y, pch=1, col="black")
    panel.points(Duncan$education[subscripts], y, col="red", pch=2)
  }
)
```


```

Now, you'll notice that the limits are set by prestige (y-axis) and income (x-axis). Education actually has a bit bigger observed range than income, so some of the education points are not being displayed (i.e., they are outside the bounds of the plot). We can change the bounds of the x-axis with the `xlim` argument:

```
xyplot

```
prestige ~ income | type, data=Duncan, as.table=T,
  xlim=c(0,100),
  panel=function(x,y,subscripts){
    panel.points(x,y, pch=1, col="black")
    panel.points(Duncan$education[subscripts], y, col="red", pch=2)
  }
)
```


```

Now, the only other thing we might want to do here is to add a legend identifying the income and education points. We can do this as we did above:

```
xyplot

```
prestige ~ income | type, data=Duncan, as.table=T,
  xlim=c(0,100),
  panel=function(x,y,subscripts){
    panel.points(x,y, pch=1, col="black")
    panel.points(Duncan$education[subscripts], y, col="red", pch=2)
  }
)
```

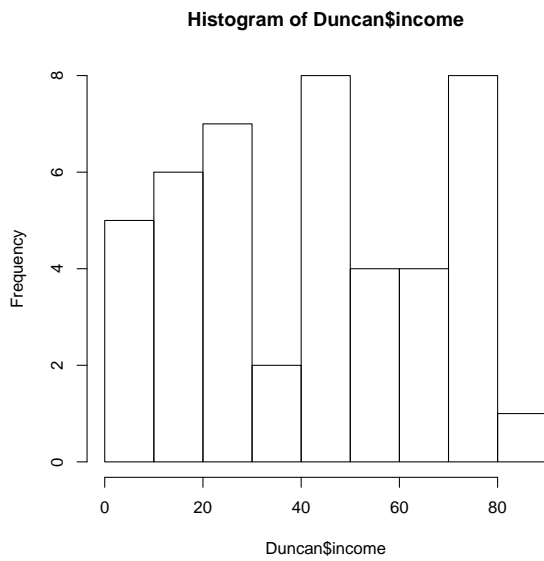

```

## 2.2 Other Lattice Graphs

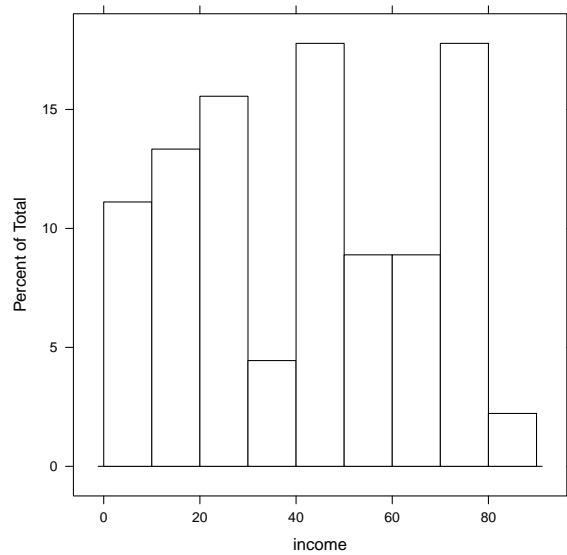
Lattice has some other options, too. These are generally wrappers to `xyplot`, but they have different default behavior for different situations.

### 2.2.1 Histograms

The `histogram` function makes histograms in the lattice package. The `hist()` command is the traditional graphics analog. In `hist`, the option `freq` can either be true (T) indicating you want counts or false (F) meaning you want density on the y-axis. In `histogram()`, the argument `type` can equal 'count', 'percent' or 'density'.



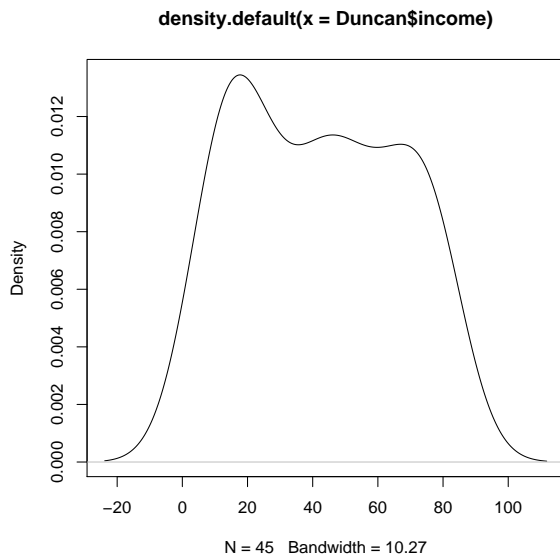
```
hist(Duncan$income)
```



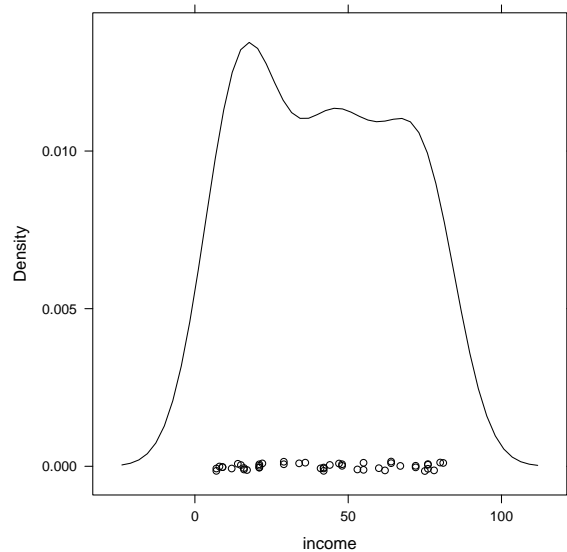
```
histogram(~income,
data=Duncan,
breaks=seq(0,90,10),
col="white")
```

## 2.2.2 Density Plots

We could also make density plots in a similar fashion using `plot` and the function `density` or using `densityplot` from `lattice`.

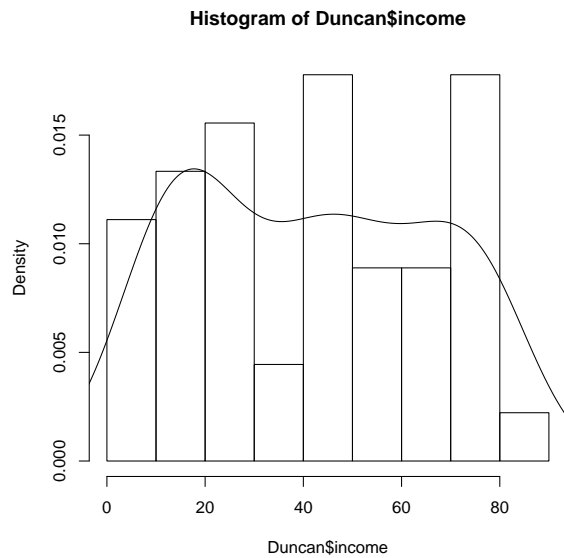


```
dens <- density(Duncan$income)
plot(dens, type="l")
```

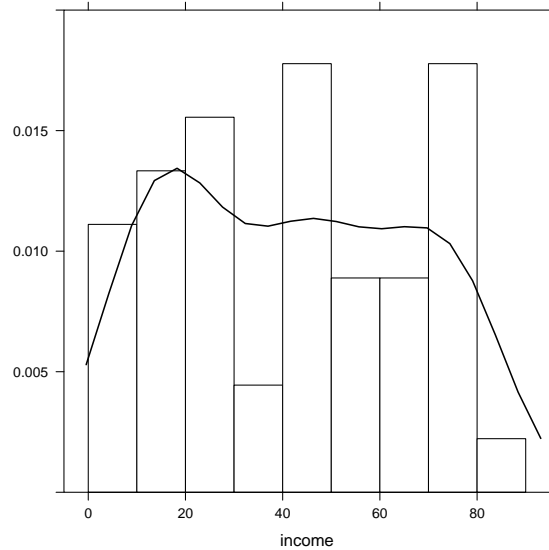


```
densityplot(~income,
data=Duncan,
col = "black")
```

### 2.2.3 Histograms with Density Overlay



```
hist(Duncan$income,  
freq=F)  
lines(dens, lwd=1.5)
```



```
xyplot(~income, data=Duncan,  
ylim=c(0,.02), xlim=c(-5,95),  
panel = function(x,y){  
  panel.histogram(x,  
    breaks=seq(0,90,10),  
    col="white")  
  panel.densityplot(x,  
    col="black",  
    lwd=1.5,  
    plot.points=F)  
})
```