The purpose of this section is to review some persistent questions or issues in the problem set code. This is particularly relevant for the current problem set. So if you don't pay attention then climate change will destroy your childrens' chances at a happy life. Maybe that was little extreme. But only a little. Many problem sets are upwards of 20 pages long, with multiple pages devoted to a single, descriptive graph. This need not be the case. We can combine multiple plots into one graph using the `par()` function to set graphical parameters. First, however, we should read in the data. There are many ways to import the data for Problem Set 5, but I thought I'd use the uploaded `.raw` file. The associated `.des` file contains the variable names.

```
raw.data <- read.table("card.raw")

col.names <- c("id", "nearc2", "nearc4", "educ", "age", "fatheduc",
               "motheduc", "weight", "momdad14", "sinmom14", "step14",
               "reg661", "reg662", "reg663", "reg664", "reg665", "reg666",
               "reg667", "reg668", "reg669", "south66", "black", "smsa",
               "south", "smsa66", "wage", "enroll", "kww", "iq", "married",
               "libcrd14", "exper", "lwage", "expersq")

names(raw.data) <- col.names
```

As always, there are many ways to clean the data, especially having been exported by Stata, but we will use this as an opportunity to practice the `lapply()` function.[1] Replace any elements within each column of the data frame matching the string `"."` with an `NA`. Then coerce the output array of lists into a data frame, employing the built-in `na.omit()` function to drop rows with any `NA` values:

```
list.data <- lapply(data, function(x) { replace(x, x == ".", NA) } )
df <- na.omit(as.data.frame(list.data))
```

Now to examine the data. Suppose we want to examine the distribution of a few variables. Let's choose `lwage`, `educ`, `exper` and `expersq` totally randomly, as if we haven't already looked at the problem set. We could produce a single, page-sized histogram for each. Or we could ... not ... do that. We define the graphical space to include two rows and two columns of graphs, and plot the histograms as we normally would:

```
par(mfrow=c(2,2))
graph.cols <- c("lwage", "educ", "exper", "expersq")

for(var in graph.cols) {
  hist(df[[var]], main = var, xlab="", breaks = 20,
       col = "grey", border = "white")
}
```

Another quick note: The code is much more readable if you separate your functions with a break between lines. That is, do not do this:

```
xFn <- function(x) {
  ...
}
yFn <- function(x) {
  ...
}
```

[1] The easiest way is to read in a CSV file. Ask your professor to use the `outsheet` command in Stata. `R` does a great job of handling missing values from a CSV file, but not so much from a `.raw` file.

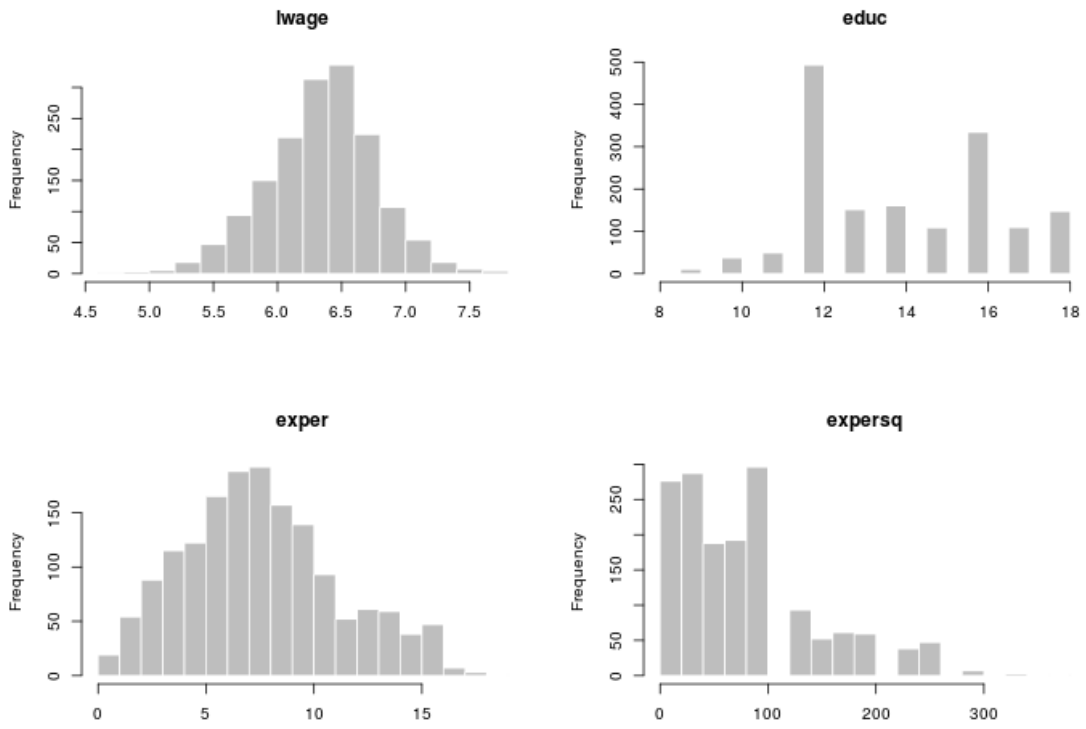

Figure 1: Descriptive histograms

The lack of break usually indicates that the functions are anonymous and part of a larger function. In ARE212, we tend to script our problem sets, rather than compose them in a functional sort of way. There are very few cases where you *wouldn't* use a line break, like so:

Now, another topic. There is a simple way to view and optimize a concave function, maybe a log-likelihood function. Consider the following function and associated plot for $x = 1, 2, \ldots 1000$ in Figure 2.

```
fn <- function(x) {
  return(-((400 - x)^2 / 1000) + 200)
}
```

It is easy enough to calculate and plot the maximum of the function. I know we've gone through this before, but it's relevant to show it again, especially in a clean way. For this, we can use the `optimize()` function. It's pretty simple, actually. There is also a European version:

```
(opt     <- optimize(fn, interval = c(1, 1000), maximum = TRUE))
euro.opt <- optimise(fn, interval = c(1, 1000), maximum = TRUE)
```

```
$maximum
[1] 400

$objective
[1] 200
```

The return object is a list with the value of the objective function at the maximum and the actual value of interest (in the domain). Another not altogether irrelevant thing worth doing, here, is plotting a series of random points along the function plot.

```
x <- sample(1:1000, 40)
```

The can then plot the result: the curve, the vertical line, the maximum point, and the array of random points.

```
curve(fn, from = 1, to = 1000, type = "l", col = "blue")
abline(v = opt$maximum, col = "red", lty = 3)
points(opt$maximum, opt$objective, col = "red", cex = 3)
points(x, fn(x), col = "black", pch = 16)
```

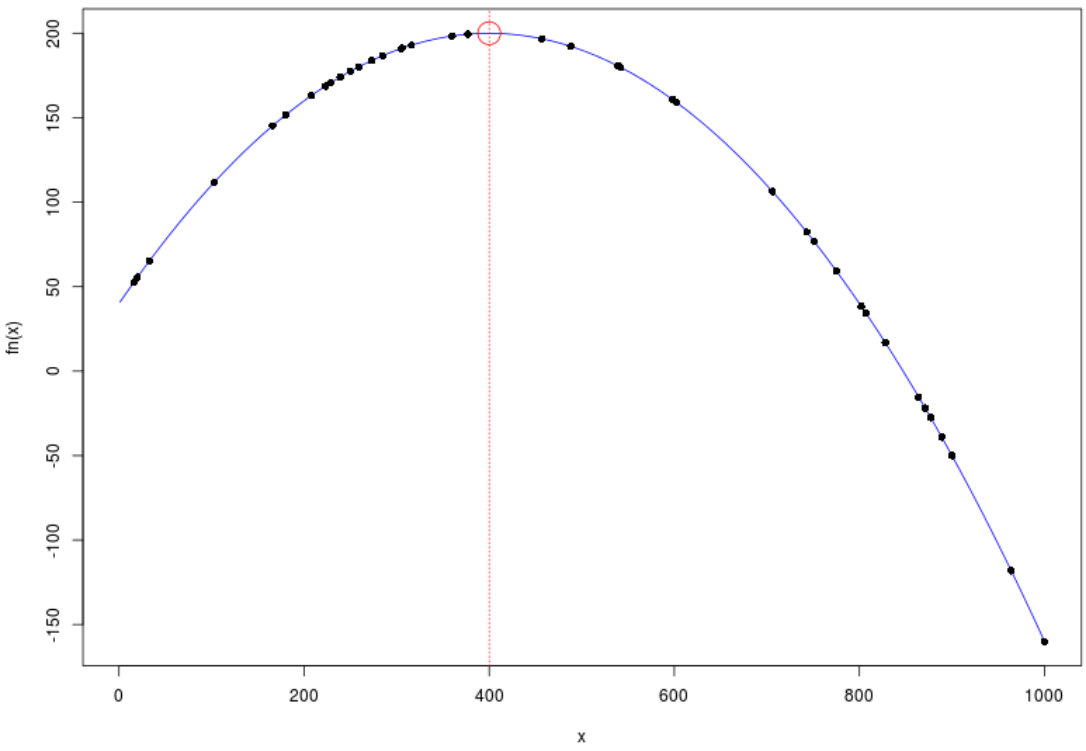

Figure 2: Function plot

Ok, another set of worthwhile functions or resources to use while in the `R` development environment:

1. `browseVignettes()`: List available vignettes in an HTML browser with links to PDF, LaTeX source, and (tangled) `R` code (if available).
2. StackOverflow questions tagged with `R`: `http://stackoverflow.com/questions/tagged/r`.
3. `data(package)`: List all data sets included in the supplied package.