

STATISTICAL GRAPHICS FOR VISUALIZING DATA AN INTRODUCTION TO LATTICE GRAPHICS IN R

William G. Jacoby
ICPSR and Michigan State University

ICPSR Summer Program
August 10-11, 2009

<http://polisci.msu.edu/jacoby/icpsr/graphics>

I. Some Basic R Concepts

A. Terminology— “R” and “S”

1. The S language was developed at Bell Laboratories, beginning in the 1970’s
2. Since the 1990’s, a commercial version of S, called “S-Plus,” has been available.
3. R is an independent and collaborative software project which is very similar (but not completely identical) to S.
4. R was originally developed by Robert Gentleman and Ross Ihaka from the University of Auckland (rumor has it that the initial letter of their first names is the source of the name, “R.”)
5. Since 1997, an international group of statisticians has continued to develop and oversee the R environment. Additional contributions have been provided by many other analysts.

B. R is a computing environment that creates *objects* by applying *functions* to other objects.

C. Naming R objects (i.e., datasets, variables, functions, etc.)

1. R names can be arbitrarily long (but, short names are preferable for practical reasons).
2. R names can be composed of letters, numbers, and periods.
3. The first character of a name must be a letter, and periods are usually used as word separators within long names.

D. R is composed of a base system, which is supplemented by user-supplied “packages.”

II. Obtaining and Installing R

A. R is open-source software— in other words, it’s free!

B. The R base system

1. The “central location” for information about R is the web site for “The R Project for Statistical Computing,” located at <http://www.r-project.org>.

2. Download the executable file required to create your R installation from a mirror site of the “Comprehensive R Archive Network” (CRAN).
3. After downloading R, run the executable to install.

C. How to install R packages

1. Start R in the usual manner (either double-click on the appropriate desktop icon, or select the R item from the Windows Start menu).
2. Select the “Packages” item from the menu bar.
3. Select “Install package(s) . . .” from the drop-down list.
4. Select a CRAN mirror site from the list that R provides, and then select the package that you want to install.
5. Immediately after installation (i.e., when the R command prompt appears in the Console), the package is ready for use.

D. How to load R packages

1. Within an R session, packages are typically loaded with the `library()` function.
2. For example, the presentation in this course will employ the “RWinEdt” package (to coordinate the WinEdt text processor with R) and the “lattice” package to create the various graphs. These are loaded with the following R statements:

```
library(RWinEdt)
library(lattice)
```

III. Communicating with R

A. The “R Console”

1. This is the window through which the user communicates with R
2. Commands (usually R functions and/or assignments) are entered on the command line within the Console
3. Note that R is case-sensitive! Therefore, it is very important to develop (and adhere to) your own personal rules for using upper- and lower-case letters in R commands.

B. Using a text processor with R

1. Although commands can be typed directly into R Console, it is usually more convenient to use a text processor to create the commands, and then paste them into the R Console.
2. R contains a sparse, but fully functional text processor. Just select the “File” item on the menu bar, and then the “New Script” item from the drop-down list. This will open a separate window within which you can type, edit, and submit R commands.
3. Any word processor (e.g., MS Word) can be used to create R commands. However, hidden codes may cause problems. For example, R does not interpret the hyphen from MS Word as a subtraction or negation operator.
4. It is generally better to use a text processor, some of which (e.g., WinEdt, Tinn-R, and ESS) contain special features that facilitate interaction with R.

C. R commands

1. The structure of a typical R statement is as follows:

```
new.object <- function(arguments)
```

Where: `new.object` is the name of a new object which is created by the command; the “<-” is the R assignment operator; `function()` is the name of a previously-defined (or built-in) function; and, `arguments` are the arguments to that function.

2. Typing in the name of an object will cause R to print out the contents of that object.
3. Defining an object without giving it a name is equivalent to typing in the name of that object (this may seem like an obtuse point now, but it has great practical importance for lattice graphics).

- D. A graphical user interface (that is, a “GUI”) for R can be obtained by installing and loading the “Rcmdr” package.

IV. Getting Data into R

A. Input from the keyboard

1. A vector of data values (e.g., the data values for a single observation, or for a single variable) can be input using the `c()` function.
2. Combine vectors into a rectangular matrix, using the `rbind()` or `cbind()` functions.

B. Reading from an ASCII text file

1. The `read.table()` function.
2. Use the `file.choose()` function to browse to the file.
3. Typical format for input data file has one line per observation, with whitespace between adjacent values.

C. The header record

1. The data file can contain a header record which provides variable names for the dataset. If so, the `header=T` must be included in the call to `read.table()`.
2. If the header record contains one fewer names than there are columns in the data, R will assume that the first column contains row names, rather than a variable.

D. Using data files created by other statistical software

1. The “foreign” package can be used to translate data files into R data frames.
2. For example, the `read.dta()` function reads data files created by STATA software.

V. Some Basic R Data Structures

- A. Data frame— corresponds to structure of a typical multivariate dataset (but don’t confuse with a matrix, which is a different kind of data structure in R!).
- B. Vector— a unidimensional array of data values (can be either numeric or character), often created with the `c()` function.

- C. Factor— an array of data values that R regards as non-numeric categories (which R calls “levels”). Note that the levels of a factor can have a specified ordering (the default is alphabetical order for factors with character values as levels).
- D. List— an array of varying elements, often used to convey several pieces of information in one succinct combination.

VI. Variable Names in R

- A. To R, variables do not really have an existence of their own; instead, they are merely named columns contained within a data frame.
 - 1. The column names are taken from the header record, if one was defined when the data were read into R.
 - 2. If there is no header record, R assigns each column a name of the form “V x ”, where x is the column number.
 - 3. The `colname()` function can be used to set (or to reassign) variable names as follows:

```
colnames(dataset) <- c("x", "y", "z")
```

Where: `dataset` is a data frame with three columns of data; x , y , and z are the names that will be assigned to the respective columns.

- B. The R Workspace
 - 1. This is a region in memory which contains all currently-defined R objects.
 - 2. The Workspace can contain many objects, including data frames, vectors, factors, and functions.
 - 3. R is very different from most statistical software packages which can only interact with a single dataset at a time.
- C. When a variable name is included in a function, R needs to know where to find that variable within the Workspace. There are several ways to provide this information.
 - 1. Use the fully-qualified version of the variable name: `data.frame.name$variable.name`
 - 2. When referring to a variable name in an R function, use the `data=` parameter, if it is available (the `data=` parameter is available in all lattice general display functions).
 - 3. The `with()` function instructs R to carry out a particular function on data from a specified data frame. For example, consider the statement:

```
with(sample.data, mean(x))
```

The preceding command instructs R to calculate the mean of the variable, x , which is found within the data frame, `sample.data`.

- 4. Include the data frame in the R search path, using the `attach()` function. NOTE: I recommend that you do *not* do this!

VII. Some Very Basic Data Manipulation Concepts

- A. Printing out the contents of a data frame (or some other type of data object)
 - 1. Can always print entire data frame by simply typing its name (unwieldy with large datasets!).
 - 2. The `str()` function summarizes the structure of a data frame in a relatively comprehensible form, without too much printed output.
- B. The `fix()` function opens a new window containing a data editor, which can be useful for changing specific data values for specific observations.
- C. Arithmetic operations are very similar to those in other software packages.
 - 1. For example, `mydata$new.x <- mydata$x + 5` will create a new column, named `new.x`, in the data frame, `mydata`. This new column is obtained by adding 5 to every entry in the column of `mydata` named `x`.
 - 2. Common arithmetic operators include `+` for addition, `-` for subtraction, `*` for multiplication, `/` for division, and `^` for raising to a power.
 - 3. Arithmetic operators are “vectorized,” meaning that they are applied to each element of conforming vectors (or matrices) and that scalars are expanded to vectors of appropriate size for the operation.
- D. Subscript or index operators can be used to refer to specific subsets of a data object.
 - 1. For example, `mydata[3,5]` would refer to the value that occurs in the cell at the intersection of the third row and fifth column of data object `mydata`.
 - 2. Logical subscripts can be used to select subsets of a data object that fit specified conditions. For example, `mydata$x[mydata$x > 5]` selects only those observations on variable `x` (contained within data object `mydata`) that have `x` values greater than 5.
 - 3. Logical subscripts can be used to perform conditional data manipulations. For example, consider the following statement:

```
mydata$x[logical.condition] <- new.value
```

The preceding statement specifies that, for those observations where `logical.condition` is true, variable `x` should be assigned the value, `new.value`.
 - 4. Logical subscripts are one of the most powerful data manipulation tools in R

VIII. Getting Help in R

- A. Each function in R has an associated help file (this is also the case for some packages).
 - 1. To view the help file for a function called `this.function`, type `help(this.function)`
 - 2. The help files can also be accessed by typing a question mark, followed immediately (i.e., no spaces) by the function name. So, `?this.function` would produce the help file for `this.function`.
 - 3. If a function is contained in a package, then the help file for that function is not available until the package is loaded.

- B. If you know part of a function's name, but want to determine the entire name, you can use the `apropos()` function.
 - 1. For example, `apropos("xy")` would return a list of all objects with the character string "xy" appearing in their names.
 - 2. Note that the argument to `apropos()` must be enclosed in quotation marks. Otherwise, R looks for an object named "xy".
- C. Help files and R manuals are also available in html format.
 - 1. Select the "Help" item on the menu bar, then select "Html help" from the drop-down list.
 - 2. I find the html help particularly useful for packages and functions they contain.
- D. Typing the name of a function, without the parentheses, will print out the contents of that function (a function is an object, like any other object in R, so it obeys the same rules); sometimes this information can provide useful insights.

IX. Introductory Thoughts about Lattice Graphics

- A. The "lattice" package
 - 1. Created by Deepayan Sarkar, from the University of Wisconsin
 - 2. Very similar to the trellis graphics system in S-Plus
 - 3. The lattice system can be used to generate "trellis displays," which are multipanel graphs suitable for illustrating multivariate data.
 - 4. The defaults in the lattice system operationalize the principles of graphical display laid out by William S. Cleveland.
 - 5. The combination of well-conceived defaults and virtually unlimited flexibility makes the lattice system an excellent tool for many scientific graphing tasks.
- B. There are *always* alternative ways to create any graph in R.
 - 1. The R base graphics system can also be used to construct graphical displays which are very similar to those produced by the lattice package.
 - 2. The R grid graphics system (which "contains" the lattice package) provides a powerful set of functions and resources for creating new graphical tools and integrating them with the rest of the R environment.
 - 3. The `ggplot2` package operationalizes the "grammar of graphics" approach developed by Leland Wilkinson.
- C. Getting help for lattice graphics
 - 1. Many people find that the references for the lattice system are not very helpful; some of the references materials for the S-Plus Trellis graphics system are also applicable to lattice in R.
 - 2. Use the help files for lattice functions frequently!
 - 3. The help file for the `xypplot()` function is particularly useful because it contains information about many parameters used in the lattice general display functions.
 - 4. The help file for the panel function associated with a particular type of graph (for example, `panel.xypplot()`) can also be very useful (panel functions are explained below).
 - 5. Within reason, trial and error is the best teacher!

X. Basic Lattice Functions

A. Basic form for lattice function call: `function.name(formula)`

1. The general arrangement of a formula in a lattice function is:

`vertical.axis.variable ~ horizontal.axis.variable`

2. Note that the tilde operator (i.e., `~`) must be used in a lattice function call, even if the graph only uses a single variable.
3. For example, `histogram(~data$x)` or `xyplot(data$y ~ data$x)`
4. Multipanel displays require slightly more complicated formulas, such as `histogram(~data$x | data$z)`

B. Some important lattice display functions

1. `histogram()`, `densityplot()`, `bwplot()`, and `stripplot()` for displaying univariate data.
2. `qqmath()` and `qq()` for various quantile plots.
3. `barchart()` and `dotplot()` for displaying labeled data values.
4. `xyplot()` for creating scatterplots. Note that many other lattice graphs can be conceptualized as special cases of `xyplot()`.
5. `cloud()` and `wireframe()` for three-dimensional scatterplots and surface plots, respectively.

XI. Modifying the Details of a Graph

A. Some optional (but useful) arguments for general display functions

1. The `data=` argument specifies the data frame, making fully-qualified variable names unnecessary
2. the `xlab=` and `ylab=` arguments provide axis labels (variable names are default)
3. The `aspect=` argument sets the aspect ratio for each panel of the graphical display
4. The `xlim=` and `ylim=` arguments set the bounds (or limits) of the axes.
5. The `main=` argument provides a title for the graph.
6. The `cex=` argument changes the size of an element (e.g., text, or the plotting symbol) in a graph.

B. The `scales=` argument uses a “mini-language” to modify the characteristics of the display axes, ticks, and tick labels.

1. The input to `scales=` is a list, which may be composed of separate lists for each axis
2. Arguments in these lists can include `tick.number=`, `at=`, `labels=`, as well as many others.

C. The order in which the arguments are supplied to the general display function is arbitrary. But, any graphical output produced by an argument will write over any output generated by earlier arguments.

XII. Panel Functions

- A. Creating a trellis graph is a two-step process (although the distinction between the two steps is often invisible to the user)
1. The function call (e.g., issuing the `histogram()` command) activates the “general display function” for the graph, which sets up the external components of the display (including the scale rectangle, axis tick marks, tick and axis labels, etc.).
 2. The general display function calls the “panel function” which creates everything that is placed into the plotting region of the graph (including plotting symbols, histogram bars, etc.)
- B. Every general display function in the lattice package has a default panel function. The name of the default panel function is usually the name of the general display function, with a `panel.` prefix.
1. For example, the default panel function for the general display function, `histogram()`, is named `panel.histogram`.
 2. The panel function, itself, is called from the general display function by the `panel=` argument.
 3. The function call, `xyplot(y~x, data=dataset)` is actually equivalent to the following:

```
xyplot(y~x, data=dataset,  
       panel = panel.xyplot  
       )
```

- C. Any arguments to the general display function that affect anything within the plotting region of the graph are actually passed on to the panel function.
1. The `pch=` argument determines the plotting symbol, the `col=` argument determines the color of a plotted element, and so on.
 2. Plotting symbols in a scatterplot could be modified in the general display function, as follows:
- ```
xyplot(y ~ x, data = dataset,
 col = "black")
```
3. The `col = "black"` argument is *not* evaluated by the general display function, `xyplot`; instead, it is passed “invisibly” to `panel.xyplot`.
  4. For all but the very simplest modifications to the function defaults, it is probably better to invoke the panel function and modify its elements directly.

- D. Invoking and modifying the panel function explicitly.

1. Actually, create a new panel function that modifies the default panel function.
2. The `function` function is used, and it has the following general form:

```
new.function <- function (argument list) {
 body of function
}
```

3. In the preceding general form:
  - a. The argument list is contained in parentheses, and it provides one or more pieces of information that are required to evaluate the function.
  - b. The contents of `function` are contained within curly braces.
  - c. The body of the function consists of one or more statements which are usually, themselves, calls to additional functions.

4. Applying the preceding general form to the panel function, we could change the color of the plotting symbols in the scatterplot as follows:

```
xyplot(y ~ x, data = dataset,
 panel = function (x, y) {
 panel.xyplot(x, y, col = "black")
 }
)
```

5. There are several details in the previous example worth noting:
- The new panel function requires two arguments ( $x$  and  $y$ ); here, these are the horizontal axis variable and the vertical axis variable from the general display function.
  - The body of the new function is simply the original panel function, `panel.xyplot`.
  - Here, `panel.xyplot` picks up the two variable names that are passed as arguments from the general display function ( $x$  and  $y$ ; note that they are given in reverse order.).
  - One additional argument is supplied to `panel.xyplot`, that is `col = "black"`.
  - There are many other arguments that could be given to `panel.xyplot`; however, since they are not mentioned in the function call, they are left at their default values.
  - Note that the equal sign (`=`) is used rather than the usual assignment operator (`<-`) because we are assigning a value to an argument within a function (even though that “value” is, itself, a new function).

### XIII. Using Several Panel Functions in a Single Graph

- A. Multiple panel functions are often used simultaneously; each subsequent panel function “writes over” any material that was drawn by an earlier panel function.
- B. Panel functions can be used to insert additional elements within the plotting region of a graphical display.
- `panel.abline` inserts a line at a specified position within the plotting region
  - `panel.lmline` inserts a regression line into a scatterplot
  - `panel.loess` fits a loess curve to a scatterplot
  - `panel.text` inserts text (e.g., for point labels and so on)
- C. Panel functions can be used to perform selective modifications on plotting elements. For example:

```
xyplot(y ~ x, data = dataset,
 col = "black",
 panel = function (x, y) {
 panel.xyplot(x[x <= mean(x)], y[x <= mean(x)], col = "red")
 panel.xyplot(x[x > mean(x)], y[x > mean(x)], col = "blue")
 }
)
```

#### XIV. Displays for Subsets and Grouped Data

- A. The `subset=` argument can be used in the general display function to specify a subset of the data to be plotted, using logical conditions.
- B. A grouping variable can be used to differentiate subgroups within a single display, using the `groups=` argument.
  - 1. Specifying `groups=x` in the lattice function will produce a graph in which different plotting colors, symbols, or line types (if appropriate) are used for each distinct value of the variable, `x`.
  - 2. The user can control the plotting symbols used across the values of the grouping variable by providing vectors of values to arguments like `pch=`, `col=`, and `lty=`.
- C. As mentioned earlier, subsets can be plotted separately by using several panel functions under a single general display function.
- D. If a graph displays several distinct subsets of objects (i.e., using different plotting symbols, separate curves, etc.), the `key=` argument can be included in the general display function.
  - 1. Parameters are passed to `key=` in a list. Elements of this list usually include `text=` and either `points=` or `lines=`, depending on the specific type of information.
  - 2. Each of the preceding list elements is a vector, with each entry in the vector corresponding to one of the subsets being plotted in the lattice display.
  - 3. Further parameters to `key=` determine placement of the key (`space=`) and whether a border will be drawn around the key (`border=`).

#### XV. Three-Dimensional Displays

- A. The `cloud()` function produces three-dimensional scatterplots
  - 1. The formula given to the `cloud()` function is a generalization of that used in `xyplot()` to generate a two-dimensional scatterplot. For example, `cloud(z ~ x * y, data = mydata)` would produce a three-dimensional scatterplot with variable `z` shown on the vertical axis, and variables `x` and `y` on the two horizontal axes. Note that the slightly different formula, `cloud(z ~ x y, data = mydata)+`, produces identical results.
  - 2. Many of the arguments to `cloud()` are either identical to those used with `xyplot()`, such as `pch=`, `col=`, and so on, or obvious generalizations (e.g., `zlab=` to produce a label for the vertical axis).
  - 3. The `scales=` works a bit differently from `xyplot()`. In `cloud()`, it is usually used to remove the arrows that are drawn by default along the axes (and replace them with tick marks and scale values for the three variables).
  - 4. The `screen=` argument can be used to provide a list of values that control the amount of rotation for each axis (NOTE: Adjusting these values can be tricky and often takes a great deal of trial and error!).
  - 5. The `distance=` argument controls the degree of perspective shown in the display.

- B. The `wireframe()` function can be used to generate plots of smooth surfaces in three dimensions.
  - 1. The main argument to `wireframe()` is a three-variable formula, similar to the type used in `cloud()`.
  - 2. In order to produce an effective surface plot, the observations used in `wireframe()` should comprise a regular “grid” of evenly- and closely- spaced values covering the ranges of both independent variables.
  - 3. The `expand.grid()` provides an easy tool for creating the grid of independent variable values. After the grid is created, just apply the appropriate function to the generated values, in order to create the values of the `z` variable (and, hence, the surface).
  - 4. The `shade=` logical argument can generate “color draping” across the wireframe surface. The `alpha=` argument controls the saturation of the color (smaller values, between zero and one, correspond to more transparency).
  
- C. The `levelplot()` function uses a color gradient to show variation in one variable, across the ranges of two other variables. The arguments to this function are, basically, identical to those used for `wireframe()`.

## XVI. Multipanel Displays

- A. The philosophy underlying trellis (and lattice) graphics usually discourages the use of three-dimensional scatterplots.
  - 1. Problematic for accurate visual perception of quantitative information.
  - 2. Virtually impossible to generalize beyond displays of trivariate data.
  
- B. Instead, employ a strategy called *multipanel conditioning*.
  - 1. Essential idea is to examine the structure of data, while holding potential confounding variables constant.
  - 2. For example, show scatterplots of  $Y$  versus  $X$  for subsets of observations, defined by the values (or intervals of values) on some third variable,  $Z$ .
  - 3. Each subset is plotted within a separate panel in a common display; the panels are arranged in a rectangular pattern (hence, the terms “trellis” and “lattice”).
  - 4. The “details” of the scatterplots remain constant across the panels of the display; the only thing that changes is the data.
  - 5. Cleveland and his colleagues argue (and demonstrate with experimental evidence) that trellis displays can be used to depict fairly complex multivariate structure.
  
- C. Multipanel displays can be generated very easily, by specifying the appropriate formula in the call to the lattice function
  - 1. The formula,  $y \sim x \mid z$ , would produce separate plots of  $y$  versus  $x$ , conditioned on the values of  $z$ .
  - 2. It is easy to incorporate multiple conditioning variables, by simply extending the formula. For example,  $y \sim x \mid w * z$  would condition the graph of  $y$  versus  $x$  on combinations of values of variables  $w$  and  $z$ .

- D. If the conditioning variable is a factor, then the lattice display will include a separate panel for each level of the factor.
- E. If the conditioning variable is numeric, then the default is to produce a separate panel for each distinct value of the conditioning variable. Of course, this is problematic when the conditioning variable is relatively continuous.
  - 1. One solution would be to recode the continuous conditioning variable into a smaller number of discrete categories. This is easy to do with the `cut()` function.
  - 2. Another solution— particularly useful when the number of data points is relatively small— is to create a “data shingle” and use that for conditioning. Basically, a shingle is a continuous variable that has been divided into a set of overlapping categories; in other words, the categories are not mutually exclusive of each other. The `equal.count()` function is usually used to create a data shingle.
- F. The layout of panels in a trellis display is particularly important for accurate perception, retrieval, and comprehension of graphical information.
  - 1. The `layout=` argument specifies how panels are to be arranged. The value for this argument is a two-element vector giving the number of columns and rows to be used in the display. Optionally, a third element can be added to the vector, giving the number of pages.
  - 2. By default, the lattice system lays out panels starting in the lower-left corner, working to the right and upward. The logical argument, `as.table=`, can be used to change this, so that lattice begins in the upper-left and works to the right and downward.
- G. Any general display function in the lattice system can be used in a multipanel display.

## XVII. Positioning Several Graphs on a Single Page

- A. Create and save an object for each graph. For example, the statements:

```
graph1 <- xyplot(y ~ x)
graph2 <- xyplot(y ~ z)
```

Would create two objects, “graph1” and “graph2,” each containing a scatterplot. Note that these graphs would *not* be displayed.

- B. Then use R’s `print()` function to print out the contents of each of the two objects.
  - 1. The `position=c(left, bottom, right, top)` argument is used to locate the printed version of each object on the page. Position specifications are given as proportions of the total page size.
  - 2. Use the `more=` logical argument to tell R whether there are more graphs to be printed on the same page (the default is `more=FALSE` so this argument can be omitted from the last `print()` call).
- C. When placing several graphs on a single page, it is often necessary to use the `cex=` argument to make some components of the graphs smaller (e.g., plotting symbols, axis labels, etc.)

## XVIII. Trellis Settings and Graphical Parameters

- A. All elements of a lattice graph are controlled by trellis “settings.”
  - 1. The names of the settings are relatively self-explanatory (e.g., the `plot.symbol` setting controls the plotting symbols used in scatterplots and other graphical displays).
  - 2. Each setting is an object that contains several parameters (e.g., `plot.symbol` contains a `col` parameter to control the color of the plotting symbols, a `cex` parameter to control the size of the symbols, and so on).
- B. Users seldom need to deal with the trellis settings directly, since most of the parameters can be accessed and modified directly through the general display function or the panel function.
- C. However, certain parameters (e.g., the “whiskers” in a box plot) can only be modified through the trellis settings.

### D. Accessing trellis settings

- 1. The `trellis.par.get()` function accesses settings. For example:

```
new.symbol <- trellis.par.get("plot.symbol")
```

Retrieves the `plot.symbol` setting and assigns it to the object, `new.symbol`.

- 2. The parameters associated with any particular setting can be seen by printing out the contents of the setting. The `str()` function is useful for doing this.
- 3. Specific parameters in a setting are accessed by name. Thus:

```
new.symbol$col <- "black"
```

Sets the `col` parameter in the `new.symbol` object to "black".

### E. Which settings are available?

- 1. Use the `show.settings()` function to obtain a graphical display showing the settings that are currently in effect.
- 2. Use the following function to obtain a complete list of trellis settings:

```
names(trellis.par.get())
```

### F. Modifying trellis settings for all graphs in an R session

- 1. The `trellis.par.set()` function can be used to modify trellis settings for the remainder of the R session.
- 2. For example:

```
trellis.par.set(plot.symbol = new.symbol)
```

Changes the `plot.symbol` setting to the parameter values associated with the object, `new.symbol`.

- 3. All subsequent lattice graphs will have black (rather than the default cyan) plotting symbols, because the `plot.symbol` setting now contains the parameters from `new.symbol`.

G. Modifying trellis settings for a single graph

1. The `par.settings` argument can be used in a general display function to supply a list of parameters for the details of that particular graph.
2. For example, the following function produces a scatterplot with black plotting symbols:

```
xyplot(y ~ x,
 par.settings = list(plot.symbol =
 list(col = "black"))
)
```

XIX. Saving and Printing Graphs

- A. Graphs can be printed directly from R window
- B. Often more convenient to right-click on graph, and either save to a file or paste into a word-processing document.
- C. R graphics devices can be used to save graphs in other formats (e.g., Postscript, jpeg, and PDF)

XX. Lattice Graphs and Statistical Models

- A. Objects created by statistical analysis can be passed directly to `lattice` functions, often without creating a new data frame.
- B. Statistical models created in R usually have associated “methods” (e.g., calculating residuals, predicted values, and so on) that can be used as input to the formula in a `lattice` function call.
- C. The “`car`” package includes several useful graphical displays (e.g., scatterplots with marginal box plots).