An Introduction to R

Brenton Kenkel
on behalf of THE STAR LAB

February 17, 2014

License

Contents

0 Course Mechanics **4**

1 Basics of R **5**
- 1.1 Installing R . . . 6
- 1.2 The R Prompt . . . 7
- 1.3 Arithmetic in R . . . 8
- 1.4 Assigning Variables . . . 10
  - 1.4.1 Naming Variables . . . 10
  - 1.4.2 Removing Variables . . . 11
- 1.5 Vectors . . . 11
  - 1.5.1 Vector Operations . . . 12
  - 1.5.2 Creating Vectors . . . 13
  - 1.5.3 Indexing . . . 14
- 1.6 Scripting . . . 15
  - 1.6.1 Saving and Loading Data . . . 16
  - 1.6.2 The Working Directory . . . 16
  - 1.6.3 Text Editors for Scripting . . . 17
- 1.7 Help! . . . 18
- 1.8 Additional Resources . . . 21

2 Matrices **23**
- 2.1 Creating Matrices . . . 23
- 2.2 Matrix Attributes . . . 26
- 2.3 Mathematical Operations . . . 27

3 Data Frames **32**
- 3.1 Basics . . . 32
  - 3.1.1 Extracting Data . . . 33
  - 3.1.2 Summary Statistics . . . 34
  - 3.1.3 Manipulating Data . . . 37

3.2 Factors 38
3.3 Loading and Saving Data 40
3.3.1 Comma-Separated Values 40
3.3.2 Stata `.dta` 41

4 Plots **43**
4.1 Scatterplots 43
4.2 Other Plots: A Quick Cookbook 48
4.2.1 Histogram 48
4.2.2 Density Plot 49
4.2.3 Box Plot 49
4.2.4 Bar Plot 49
4.2.5 Line Plot 50
4.2.6 Text Plot 50
4.3 Exporting Graphics 51

5 Linear Regression **52**
5.1 Running a Regression 52
5.1.1 Formula Syntax 54
5.2 Analyzing the Output 55
5.3 Presenting Results 58
5.3.1 Tables 58
5.3.2 Coefficient Plots 60

6 Regression Tests and Diagnostics **62**
6.1 Robust Standard Errors 62
6.2 Wald Tests 64
6.3 Additional Tests 66
6.4 Diagnostic Plots 68
6.4.1 Partial Regression Plots 69
6.4.2 Component-Residual Plots 70

7 Generalized Least Squares **73**
7.1 Weighted Least Squares 73
7.2 Generalized Least Squares 76

Chapter 0

Course Mechanics

This course will

- introduce you to the R language and its capabilities
- familiarize you with basic R syntax and usage
- show you how to implement basic statistical techniques in R.

The purpose of this course is *not* to teach you everything you will ever need know about R. It's to give you the essential knowledge base you need to be able to go out and learn those things on your own.

We will meet every other Friday in THE STAR LAB from 2:00–4:00 pm. Each session, we will go through a chapter of these notes. Throughout the notes, you will see examples of R input and output like this:

```
> 2 + 2
[1] 4
```

When we reach these examples, you should run them yourself in an R session at your workstation. If the output you get doesn't match what's in the notes, let me know right away and we'll figure out what's going on.

As always, I encourage you to ask questions. If you don't understand something or want to know more, speak up! You can also email me your R questions anytime, and I will try to address them quickly.

Every week, I will hand out a problem set. These problem sets are optional, as in they won't be graded and your status in the program hinges in no way on whether you complete them. That said, I do encourage you to take the time to work through them, and I will send out answer keys so you can check your work.

Chapter 1

Basics of R

What is R?

- A programming language designed for statistical applications
- A statistical environment for data analysis
- Something you need to learn if you want to get by in PSC 405 and PSC 505

Why do we like R so much?

- It's free and open-source
- It's cross-platform — the same code will produce the same results on Windows, Mac, or Linux
- It's easy to use and to program with[1]
- It's regularly updated — there's a new major version roughly every year, with intermediate revisions every 4–5 months
- It's becoming the *lingua franca* of serious empirical work in political science
- It's already the *lingua franca* of applied statistics

[1]Some folks in the department may tell you that "Stata is better/easier for x," but unless x is heavy-duty time series econometrics, they're wrong.

- There are thousands of user-contributed packages — if a statistical technique has been published in a reputable journal, there's probably an R package for it[2]

What should you use R for?

- All of your basic data analysis: descriptive statistics, linear regressions, logits, probits, duration models, etc.
- Programming your own estimators and statistical models (the main goal of PSC 505)
- Numerical simulations — examining the properties of an estimator under a given probabilistic data-generating process when pen-and-paper analysis is infeasible (this is very useful and will also be covered in PSC 505)
- Any kind of plotting — R's graphical capabilities are widely recognized as superior to those of any other statistical environment[3]

What *shouldn't* you use R for?

- Most non-statistical programming or scripting (e.g., web scraping) — it's possible to use R for this, but Python is easier and better documented
- Extremely memory-intensive statistical applications that cause R to crash or run at a snail's pace — for this kind of thing you will want to learn a compiled language like C/C++[4]
- Looking cool in front of your high school friends — I've tried, it doesn't work

1.1 Installing R

If you plan to do all your work in THE STAR LAB, this section doesn't apply to you. But if you don't want to consign yourself to dozens of endless nights in

[2] *Caveat emptor*: although contributed packages — those that aren't included with the default R installation — are vetted to ensure they won't give your computer a virus, they are *not* tested to see if they give you the right answer. Later on in the course, I'll go over some basic heuristics for whether you should trust results from a contributed package.

[3] Seeing Stata plots in journal articles makes me gag a little. Seeing Excel plots makes me lose my lunch.

[4] The number of situations where this applies is getting smaller since memory is getting cheaper, as always. Also, many problems that would apparently fall into this category can actually be fixed with basic parallel computing, which you'll also be learning in PSC 505.

our cold, windowless computer lab, you probably want to know how to install R on your home computer.

1. Go to the Comprehensive R Archive Network (CRAN) website at `http://cran.r-project.org`
2. Click on the appropriate "Download R for [your operating system here]" link (descriptions below accurate as of 11 January 2012)

   Windows Go to the `base` subdirectory and click the big "Download R 2.X.X for Windows" at the top

   Mac Click the first download link under "Files," which will be labeled "R-2.X.X.pkg (latest version)"

   Linux Choose the subdirectory that corresponds to your Linux distribution, or download the source code from the homepage and compile it yourself[5]
3. Run the executable you downloaded

As I mentioned above, R is updated often. You should keep up to date, because the updates contain speed improvements, bug fixes, and new features. The easiest way to know when a new version is coming out is to subscribe to the R-announce listserv at `https://stat.ethz.ch/mailman/listinfo/r-announce`. Don't worry about this list flooding your inbox: it sends about one email every two months, either to announce a new version of R coming out in the future, or to list the features and improvements in a version that has just been released.

1.2 The R Prompt

When you first start R you'll first see a message listing the version of R you're using, information about the software license, and a few other basic commands. Below that is the *prompt*, which looks like this:

```
>
```

This means R is waiting for you to enter a command. What you type appears next to the prompt, and you run it by hitting `Enter`. Usually, after you enter a command, R displays the output and then brings back the prompt.

[5]If you're on Ubuntu or another Debian-like distribution, I recommend getting R from the repositories provided by CRAN rather than the standard ones, as they are updated much more frequently.

```
> 1 * 1
[1] 1
>
```

You can scroll through previous commands you've entered by using the up and down arrows on your keyboard.

Sometimes, instead of giving you the prompt again, R will instead display a +, known as a *continuation line*. This means you entered an unfinished command and that R is waiting on additional input.

```
> 2 *
+ 2
[1] 4
```

To get out of a continuation line and bring back the prompt, hit `Esc` or `Ctrl-C`.

On a final note, anything folllowing a # is a *comment* and is ignored by R. You will be using comments often when writing R scripts.

```
> 3 + 3
[1] 6
> 3 + 3  # + 20
[1] 6
```

1.3 Arithmetic in R

As you saw in the examples above, the simplest use of R is as a calculator. All of the basic mathematical operators are available: + and - for addition and subtraction, * and / for multiplication and division, ^ for exponents, %% for modulo, and %/% for integer division.

You should familiarize yourself with the basic rules of operator precedence; e.g., the fact that $3 \times 2 + 1 = (3 \times 2) + 1 \neq 3 \times (2 + 1)$. R follows all of the standard rules. If you don't feel like learning the rules, you may use parentheses to your heart's content.

```
> 3 * 2 + 1
[1] 7
> 3 * (2 + 1)
[1] 9
```

```
> (3 * 2) + 1
[1] 7
> 2^2 + 5
[1] 9
> 2^(2 + 5)
[1] 128
```

Many basic mathematical functions are available as well.

```
> log(10)
[1] 2.3026
> log(10, base = 10)
[1] 1
> exp(1)
[1] 2.7183
> sin(0)
[1] 0
> acos(-1)
[1] 3.1416
```

The `base` option in the `log` function is known as an *argument*. If `base` isn't specified, R assumes you want the natural logarithm. We'll talk more about function arguments and how to discover them in just a bit.

R follows the time-honored principle of "garbage in, garbage out," which we can see by asking it to take the logarithm of numbers that don't have well-defined logarithms.

```
> log(0)
[1] -Inf
> log(-1)
[1] NaN
```

`-Inf` stands for $-\infty$, of course. `NaN` means "not a number," which is R's polite way of telling you that you asked it to generate nonsense.

1.4 Assigning Variables

When you're using R for data analysis, you'll want to refer to the same computations repeatedly without having to type them out each time. You can do this by assigning those computations as *variables*, which you can then refer to by name.

An assignment statement in R looks like this:

```
> x <- exp(1)
```

The "arrow," `<-`, is the assignment command. On the left side of the arrow is the name of the variable we have created. On the right side is the value that we have assigned to it. We can now use `x` at the prompt like any other number.

```
> x
[1] 2.7183
> x - 2
[1] 0.71828
> log(x)
[1] 1
```

If you want to assign a new value to a variable, just use the arrow again:

```
> x <- exp(2)
> x
[1] 7.3891
```

1.4.1 Naming Variables

Variable names can be as long or short as you want. When you are writing code or performing analysis for your research, it is often useful to use descriptive names. This way, if you leave your code dormant for six months, you can pick it up again and know what's going on.

```
> n <- 10  # not descriptive
> numberOfStudents <- 10  # better
```

There are few restrictions on variable names in R. The main rules are that variable names must start with a letter and may not contain punctuation other than periods and underscores. So `number.of.students` and `number_of_students` are valid, while `number-of-students` is not.

1.4.2 Removing Variables

Use the ls command to see a list of the variables that have been assigned.

```
> ls()
[1] "n"                "numberOfStudents" "x"
```

To remove a variable from memory, use rm. This *cannot* be undone, so be careful!

```
> rm(numberOfStudents)
> ls()
[1] "n" "x"
```

If you want to get rid of everything, use rm(list = ls()). This is sure to ruin your day at least once when you're doing some analysis that takes forever and forgot to save the results.

```
> rm(list = ls())
> ls()
character(0)
```

character(0) is R's way of saying "This is a vector that is supposed to contain text, but there's nothing in it."

1.5 Vectors

You may have noticed that when you perform a simple calculation in R, a [1] appears next to the result.

```
> 2+2
[1] 4
```

This is because R thinks of all numbers as *vectors*. If you enter a scalar computation like 2+2, R considers it the addition of two vectors of length 1. The [1] is to tell you that the adjacent entry is the first entry of the output vector.

The simplest way to represent a vector in R is to use the c command, with each entry separated by commas. For example, to represent the vector $\mathbf{x} = (1,4,6,1)$, you would enter

```
> x <- c(1, 4, 6, 1)
> x
```

```
[1] 1 4 6 1
```

c stands for "concatenate," so you can also use it to string a bunch of different vectors together.

```
> y <- c(2, 3, 2, 1)
> c(y, y, y, x, x, x, y, y, x, x)
 [1] 2 3 2 1 2 3 2 1 2 3 2 1 1 4 6 1 1 4 6 1 1 4 6 1 2 3 2 1
[29] 2 3 2 1 1 4 6 1 1 4 6 1
```

1.5.1 Vector Operations

Almost all of R's standard mathematical functions (including the basic arithmetic operations) work with vectors.

```
> x + y
[1] 3 7 8 2
> x / y
[1] 0.5000 1.3333 3.0000 1.0000
> log(x)
[1] 0.0000 1.3863 1.7918 0.0000
```

There are plenty of additional mathematical and statistical commands that operate on vectors, usually with intuitive names.

```
> sum(x)
[1] 12
> prod(x)
[1] 24
> mean(x)
[1] 3
> median(x)
[1] 2.5
> sd(x)
[1] 2.4495
```

You can use `sort` to put the entries of a vector in order, `rev` to reverse it, and `length` to find out how many entries it has.

```
> sort(x)
[1] 1 1 4 6
> rev(sort(x))
[1] 6 4 1 1
> sort(x, decreasing = TRUE)
[1] 6 4 1 1
> length(x)
[1] 4
```

1.5.2 Creating Vectors

There are ways to create vectors according to patterns without having to type out all their entries individually. The easiest way is with the colon.

```
> 1:20
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
[19] 19 20
> 50:30
 [1] 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33
[19] 32 31 30
```

A more general version of the colon is `seq`.

```
> seq(0, 10)
 [1]  0  1  2  3  4  5  6  7  8  9 10
> seq(0, 10, by = 2)
[1]  0  2  4  6  8 10
> seq(0, 1, length.out = 11)
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

The `rep` command lets you repeat things.

```
> rep(10, times = 3)
[1] 10 10 10
> rep(x, times = 3)
 [1] 1 4 6 1 1 4 6 1 1 4 6 1
> rep(x, each = 3)
 [1] 1 1 1 4 4 4 6 6 6 1 1 1
```

1.5.3 Indexing

You'll often want to extract elements from a vector—either specific elements or via a pattern. In either case, you'll use square brackets.

```
> z <- seq(0, 20, by = 2)
> z[3]  # 3rd entry
[1] 4
> z[c(3, 10)]  # 3rd and 10th entries
[1]  4 18
> z[-c(3, 10)]  # all but the 3rd and 10th
[1]  0  2  6  8 10 12 14 16 20
```

To extract via a pattern, you'll need to know something about *logical comparisons*. For example, the elements of z greater than 10 are given by:

```
> z > 10
 [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
[10]  TRUE  TRUE
```

To extract these, put the logical statement within the brackets:

```
> z[z > 10]
[1] 12 14 16 18 20
```

The full list of logical operators is:

Operator	Meaning
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal
!=	not equal

You can string logical statements together by using & ("and") and | ("or").

```
> z > 3 & z < 10
 [1] FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
[10] FALSE FALSE
> z[z > 3 & z < 10]
```

```
[1] 4 6 8
> z[z < 3 | z > 10]
[1]  0  2 12 14 16 18 20
```

Make sure not to accidentally use = instead of == when testing for equality. A single equals sign is basically synonymous to <-, so this will result in your variable being overwritten.

```
> x <- c(0, 1, 0, 2)
> x == 0  # good
[1]  TRUE FALSE  TRUE FALSE
> x = 0   # bad
> x
[1] 0
```

1.6 Scripting

When you're using R for data analysis, it is important that you *save your results* and *be able to replicate them*. The best way to do this is to run your R code from scripts instead of at the interactive prompt.

To see how this works, copy the following script into Notepad and save to your `Z:` drive as `test1.r`.

```
## FILE: test1.r
## DESCRIPTION: First use of scripting in R
## DATE: 2012-01-20
## AUTHOR: You (you@rochester.edu)

## Create some data
mydata <- c(3, 7, 19, 23, 31, 42)

## Compute some descriptive statistics
mean.mydata <- mean(mydata)
sd.mydata <- sd(mydata)
```

You can now run the script by opening an R session and using the `source` command.

```
> source("z:/test1.r")
```

It will now be as if you had typed each of the commands included in the script at the R prompt.

```
> ls()
[1] "mean.mydata" "mydata"      "sd.mydata"
> mydata
[1]  3  7 19 23 31 42
> mean.mydata
[1] 20.833
```

1.6.1 Saving and Loading Data

You can also save the results of R output using the save command. This can be placed inside a script (preferable), or you can run it at the prompt after the script has finished (riskier, if you're forgetful). Just list the names of all the variables you want to save, followed by a file argument with the name of the file to save them in.

```
> save(mean.mydata, sd.mydata, file = "z:/results1.rda")
```

You can use the load command to retrieve the variables that you saved. Close out of R and open up a new session (tell it *not* to save your workspace if it asks). Now run the following commands.

```
> ls()  # to make sure nothing's there
character(0)
> load("z:/results1.rda")
> ls()
[1] "mean.mydata" "sd.mydata"
```

Your saved variables have been retrieved!

1.6.2 The Working Directory

When using source, save, and load, you may feel lazy and not want to type full file paths. You can find out R's "working directory" using the getwd command. On Windows, the output will look like:

```
> getwd()
```

```
[1] "C:/Windows/system32"
```

On a Mac or Linux, it will look like:

```
> getwd()
[1] "/home/bkenkel/Dropbox/teaching/r_course/latex"
```

If you run a command without specifying a path, such as in the following example, R will assume you mean to place the file in (or take it from) the working directory.

```
> save(x, file = "results.rda")
```

In most graphical interfaces for R, there is a menu option to change the working directory. If you want to change it from the prompt, use the `setwd` command.

```
> setwd("C:/Users/username/directory/goes/here/")  # Windows
> setwd("/Users/username/directory/goes/here/")  # Mac
> setwd("~/directory/goes/here/")  # shorter way for Mac
```

1.6.3 Text Editors for Scripting

You can use just about any plain-text editing program to write R scripts. It's best to look for one that has the following features:

- Syntax highlighting, which displays different features of your code in different fonts or colors, as in the following example:

  ```
  ## This is a comment
  x <- rnorm(n = 5, mean = 10, sd = 2)
  y <- sort(x, decreasing = TRUE)
  z <- c("a", "b", "d")
  f <- function(x) {   # another comment!
    ans <- (x + 7) / 3
    return(ans)
  }
  ```

- Side-by-side script editor and R session, which you can use to test commands in your script as you write it

- Parentheses matching, which highlights the corresponding "(" when you type a ")", so you can keep track of commands like

```
> round(exp(cos(sqrt(log(sd(x)^2)+1)/2))+sin(exp(pi)))
```

Here are some of the most popular text editors and my thoughts about them. There are dozens of other good text editors out there — don't be afraid to search, experiment, and find the one that works best for you.

Built-in The R GUIs for Windows and Mac have built-in script editors. The one for Mac is pretty good. The one for Windows ... not so much.

RStudio Cross-platform. Nice balance of functionality and user-friendliness. Useful for both scripting and interactive R sessions.

Emacs Cross-platform. The text editor of choice for programming nerds.[6] Also has excellent LaTeX support. Very steep learning curve. I spent about two weeks of the summer between my first and second year learning Emacs, and the investment has paid off nicely for my productivity.

RWinEdt Windows only. Hard to set up and otherwise basically awful, unless you're the kind of person who just can't get enough of WinEdt.

TextMate Mac only, costs money. A very good text editor, also with LaTeX support.

TextWrangler Mac only. The best free all-around text editor on the Mac if you don't want to spend money, but probably not as good for R code as the R GUI's built-in editor.

1.7 Help!

Before we dive any further into R, let's go over how to get help about it and find documentation. I am not exaggerating when I say that **this is the most important topic in the entire short course**. If you have a problem with R or can't figure out how to do something, chances are (1) someone else has had the same problem before and (2) the solution is available in R's documentation, in an R package, or somewhere on the World Wide Web. The best way to look for help depends on what kind of issue you're having.

[6]Some nerds prefer Vi, but its support for R and LaTeX is objectively inferior to that of Emacs.

1. You want to know more about a particular function. Remember how we were able to compute a base 10 logarithm by calling `log` with the argument `base = 10`? If we didn't know that `log` had an argument called `base`, how could we have found out? To pull up a help page for the `log` function, enter

```
> ?log
```

at the prompt. There is a documentation page available for just about every function included in R or any of its packages.

A function help page has the following sections:

Description What the function is supposed to do

Usage What a typical invocation of the function looks like

Arguments A list of function arguments and what each of them does

Details Additional information about the function and its arguments

Value What to expect from the output of a function (e.g., a single number, a list of numbers, a matrix, something completely different)

See Also Other functions that may come in handy

Examples Illustrations of how to use the function

In theory, the documentation page tells you everything you need to know about a function. This is often the case. However, keep in mind that these help pages are usually written by programmers, who are typically great at talking to computers but not so hot with humans. Sometimes, the documentation for functions that should be very simple to explain (e.g., `sort`, which orders a vector of numbers from least to greatest) is the hardest to read. In these cases, I recommend running the code from the "Examples" section and inspecting the output carefully. The easiest way to do this is with the command

```
> example(function_name)
```

A final note: Sometimes the `?` syntax doesn't work. For example, if you want help about addition and enter

```
> ?+
```

you end up with a continuation line and no help page. To get what you're looking for, put what you're looking for in quotation marks inside the `help` function:

```
> help("+")
```

2. You don't know what function to use. Perhaps you have a task in mind but are unsure which R function, if any, will do it for you. If you have no idea what the appropriate function would even be called, use `help.search` to search the documentation of base R and every package you have loaded.

```
> help.search("ridge regression")
Help files with alias or concept or title matching
'ridge regression' using fuzzy matching:

MASS::lm.ridge     Ridge Regression
mgcv::bam          Generalized additive models for
                   very large datasets
mgcv::bam.update   Update a strictly additive bam
                   model for new data.
mgcv::gam          Generalized additive models with
                   integrated smoothness estimation
survival::ridge    Ridge regression
```

Syntax like `MASS::lm.ridge` means the function is called `lm.ridge` and is contained in the `MASS` package. To use it, you would first have to run

```
> library(MASS)
```

to load the `MASS` package. We'll talk more about packages later on in the course.

In other cases, you may have a rough idea of what the relevant function would be named. For example, suppose you know that `dnorm` can be used to calculate the density function for a normal distribution. What other functions deal with the normal distribution? To find out, use the `apropos` function.[7]

```
> apropos("norm")
 [1] "dlnorm"          "dnorm"          "norm"
 [4] "normalizePath"   "plnorm"         "pnorm"
 [7] "qlnorm"          "qnorm"          "qqnorm"
[10] "qqnorm.default"  "rlnorm"         "rnorm"
```

This gives you a list of R functions that have "norm" in their names. You can then find out what each one does by using `?`.

[7]One subtle difference between `help.search` and `apropos` is that `help.search` searches base R and all packages that you have installed on your computer, while `apropos` only searches base R and the packages you have loaded in the current session.

3. The first two don't apply or didn't work. Maybe `help.search` drew a blank, or the documentation for the function you want to use is goobledygook. Now it's time to Google, which I assume you already know how to do. You might think Googling "your problem here R" would just return a bunch of useless pages from people whose middle initial is R or who have cutely tried to trademark their website name using "(R)", but I've found it usually works fine. Other queries that will filter out more irrelevant material are "your problem here R CRAN" or "your problem here R package".

There are also lists of R-specific search engines available at `http://www.r-project.org/search.html` and `http://search.r-project.org`. The mailing list archives can be especially helpful if you are trying to decipher an error message. I don't recommend posting to the mailing list itself unless you have searched exhaustively for previous answers to your problem, have read the list's posting rules carefully, and are prepared to receive impolite or even nasty responses.

4. Even Google isn't helpful. *Only* after you have tried Googling your problem is it appropriate to bother whichever more-advanced student is sitting near you in THE STAR LAB. (Exception: STAR LAB fellows, myself included, are paid to be helpful and may be bothered without compunction at any point in the process.)

1.8 Additional Resources

These are some books (either printed or as PDFs) that might be useful as you learn R. This list isn't meant to be comprehensive — it's just a set of the things I've run into that I've found useful at some time.

Verzani, *simpleR* (`http://cran.r-project.org/doc/contrib/Verzani-SimpleR.pdf`) This is an introduction to R by John Verzani, meant to accompany an undergraduate-level introductory statistics course. Verzani later turned this into a printed book, *Using R for Introductory Statistics* (`http://www.amazon.com/dp/1584884509`), which I believe THE STAR LAB has a copy of.

Faraway, *Practical Regression and Anova Using R* (`http://cran.r-project.org/doc/contrib/Faraway-PRA.pdf`) Kevin uses this book when he teaches PSC 405. It contains examples of how to implement most basic regression techniques in R. A similar text that covers more advanced econometric techniques is Farnsworth's

Econometrics in R (http://cran.r-project.org/doc/contrib/Farnsworth-EconometricsInR.pdf).

Braun and Murdoch, *A First Course in Statistical Programming with R* (http://www.amazon.com/dp/0521694248) This is my favorite R book. Unlike most other books about R, it focuses on programming in R, not just using canned R features for data analysis. Depending on your previous programming experience, this means the book may not be useful until you have more of a handle on R itself. In any case, I recommend reading this book and working through its exercises before you take PSC 505 in the fall.

Burns, *The R Inferno* (http://www.burns-stat.com/pages/Tutor/R_inferno.pdf) Patrick Burns takes you through the nine circles of R hell. This is usually where I look first when R doesn't do what I expect or when I run into an error message that makes no sense. The utility of this guide will also become more apparent once you've begun programming in R.

Venables and Ripley, *Modern Applied Statistics with S* (http://www.stats.ox.ac.uk/pub/MASS4/) This is *the* statistician's guide to R,[8] written by two of the major contributors to the language.

Many, many more There's a whole list of additional free resources available at http://cran.r-project.org/other-docs.html.

[8]Why is it called "S" in the title? Technically, R is just one dialect of the S language; for example, there is a commercial dialect called S-PLUS that you'll still see mentioned sometimes. These days, as far as I know, R is the only S dialect that is actively developed and widely used.

Chapter 2

Matrices

Matrix algebra is at the heart of basically every statistical technique for multivariate data analysis. So if you want to do serious statistics in R, you need to learn about its matrix algebra features.

2.1 Creating Matrices

The most straightforward way to create a matrix is with the `matrix` command, which takes a vector and shapes it into a matrix. You provide the command with a vector and use the arguments `nrow` or `ncol` to specify the dimensions.

```
> x <- 1:6
> matrix(x, nrow = 2)
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> matrix(x, ncol = 2)
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

Notice that by default the matrix is filled column-wise. Use the `byrow` argument to fill by rows.

```
> matrix(x, ncol = 2, byrow = TRUE)
```

```
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

You can also create matrices by "binding" vectors together. `cbind` treats the vectors as columns, while `rbind` treats them as rows.

```
> y <- 4:9
> cbind(x, y)
     x y
[1,] 1 4
[2,] 2 5
[3,] 3 6
[4,] 4 7
[5,] 5 8
[6,] 6 9
> rbind(x, y)
  [,1] [,2] [,3] [,4] [,5] [,6]
x    1    2    3    4    5    6
y    4    5    6    7    8    9
```

The `diag` command (as in "diagonal") has a few different uses.

```
> diag(4)  # 4 x 4 identity matrix
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1
> diag(x)  # square matrix with x along the diagonal
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    0    0    0    0    0
[2,]    0    2    0    0    0    0
[3,]    0    0    3    0    0    0
[4,]    0    0    0    4    0    0
[5,]    0    0    0    0    5    0
[6,]    0    0    0    0    0    6
> Z <- matrix(1:9, nrow = 3)
> Z
```

```
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> diag(Z)  # extract the diagonal vector from a matrix
[1] 1 5 9
```

Sometimes you want to turn a matrix back into a vector. The easiest way to do this is to place the matrix within `c`, which makes a vector by going down the columns of the matrix.

```
> c(Z)
[1] 1 2 3 4 5 6 7 8 9
```

A sometimes useful tool is the `drop` command, which turns one-row or one-column matrices into vectors but preserves the structure of others.

```
> X1 <- matrix(1:4, nrow = 2)
> X2 <- matrix(1:4, nrow = 1)
> X1
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> drop(X1)  # no change
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> X2
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
> drop(X2)  # back to vector
[1] 1 2 3 4
```

If you're ever in doubt about whether something is a matrix, check its `class`.

```
> class(X2)
[1] "matrix"
> class(drop(X2))
[1] "integer"
```

If the class is `"integer"` or `"numeric"`, you have a vector, not a matrix.

2.2 Matrix Attributes

Recall that we used the square brackets to extract elements from a vector.

```
> x <- 12:1
> x[3]
[1] 10
```

The syntax to extract elements from a matrix is similar. If `X` is a matrix, `X[i, j]` gives you the jth element of the ith row.

```
> X <- matrix(12:1, nrow = 3)
> X
     [,1] [,2] [,3] [,4]
[1,]   12    9    6    3
[2,]   11    8    5    2
[3,]   10    7    4    1
> X[2, 4]
[1] 2
```

As with vectors, you can use this to change the value of a single entry in a matrix.

```
> X[3, 2] <- 8
> X
     [,1] [,2] [,3] [,4]
[1,]   12    9    6    3
[2,]   11    8    5    2
[3,]   10    8    4    1
```

You can extract an entire row by leaving the column entry blank, or vice versa.

```
> X[1, ]  # first row
[1] 12  9  6  3
> X[, 3]  # third column
[1] 6 5 4
```

Notice that when you extract a single row or column, the output is a vector. To preserve the matrix structure, use the `drop = FALSE` argument.

```
> X[, 3, drop = FALSE]
     [,1]
[1,]    6
[2,]    5
[3,]    4
```

All of the special indexing rules that we learned about last time also work for matrices.

```
> X[, 2] == 8  # which rows have 8 in the second column?
[1] FALSE  TRUE  TRUE
> X[X[, 2] == 8, ]
     [,1] [,2] [,3] [,4]
[1,]   11    8    5    2
[2,]   10    8    4    1
```

If you want to find out the dimension of a matrix without printing all of it to the R console, use dim.

```
> dim(X)
[1] 3 4
```

The nrow and ncol commands do what you'd expect.

```
> nrow(X)
[1] 3
> ncol(X)
[1] 4
```

2.3 Mathematical Operations

All of the matrix operations you'd expect to see are available in R. First, some basic matrix arithmetic.

```
> X <- matrix(1:4, nrow = 2)
> Y <- diag(2)  # identity matrix
> X + Y
```

```
     [,1] [,2]
[1,]    2    3
[2,]    2    5
> X - Y
     [,1] [,2]
[1,]    0    3
[2,]    2    3
```

Be warned: the standard multiplication command, `*`, performs *elementwise* multiplication. Use `%*%` for actual matrix multiplication.

```
> X * Y
     [,1] [,2]
[1,]    1    0
[2,]    0    4
> X %*% Y
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

If you try to multiply a matrix by a vector, R automatically treats the vector as a row or column vector depending on context.

```
> c(1, 1) %*% X
     [,1] [,2]
[1,]    3    7
> X %*% c(1, 1)
     [,1]
[1,]    4
[2,]    6
```

The transpose is provided by `t`.

```
> t(X)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
```

You can invert matrices — the crucial step in linear regression and many other statistical procedures — via the `solve` command.

```
> solve(X)
     [,1] [,2]
[1,]   -2  1.5
[2,]    1 -0.5
> solve(X) %*% X
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

It is important to note that numerical matrix inversion is *not* an exact science. The results you obtain from solve are only accurate up to a certain decimal place, normally no greater than the 16th.[1] To illustrate this, let's invert a matrix of random numbers. (Your results will differ slightly due to randomness.)

```
> Z <- matrix(rnorm(16), nrow = 4)
> solve(Z) %*% Z
            [,1]        [,2]        [,3]        [,4]
[1,]  1.0000e+00 -5.5511e-16 -1.6653e-16 -1.1102e-16
[2,] -2.2204e-16  1.0000e+00  1.1102e-16  3.3307e-16
[3,]  0.0000e+00 -1.1102e-16  1.0000e+00  0.0000e+00
[4,]  0.0000e+00  0.0000e+00  1.1102e-16  1.0000e+00
```

Notice that what we get back isn't *exactly* an identity matrix but is really close.

```
> round(solve(Z) %*% Z, digits = 12)
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1
```

Luckily, this level of precision is more than accurate enough for most normal data analysis applications in political science.

Let's go through a few more useful matrix operations. det and chol give you the determinant and the Cholesky decomposition respectively.

```
> Y <- matrix(c(1, .5, .5, 1), nrow = 2)
> det(Y)
```

[1]This is because R, like almost every computer program, uses a floating-point representation of fractional numbers. See the entry "Why doesn't R think these numbers are equal?" in the official R FAQ, http://cran.r-project.org/doc/FAQ/R-FAQ.html.

```
[1] 0.75
> chol(Y)
     [,1]    [,2]
[1,]    1 0.50000
[2,]    0 0.86603
> t(chol(Y)) %*% chol(Y)
     [,1] [,2]
[1,]  1.0  0.5
[2,]  0.5  1.0
```

There's no "trace" command included, but it's easy to compute by hand by taking the sum of the diagonal.

```
> sum(diag(Y))
[1] 2
```

`eigen` gives you eigenvalues and eigenvectors.

```
> eigen(Y)
$values
[1] 1.5 0.5

$vectors
        [,1]     [,2]
[1,] 0.70711 -0.70711
[2,] 0.70711  0.70711
```

The output of this command requires a brief detour into some R syntax. This is the first example you'll see of a *list* in R. A list is just a collection of R objects, possibly all of different types. You can see above that `values` is a vector while `vectors` is a matrix. To extract one component of a list, use `$`:

```
> eigen(Y)$values
[1] 1.5 0.5
```

We'll be seeing a lot of lists throughout this course — the output of most data analysis procedures in R is a list. One more useful command before we move on is `names`, which tells you the named elements of a list.

```
> names(eigen(Y))
```

```
[1] "values"  "vectors"
```

The last matrix operation we'll look at is the rank. Again, there's no "rank" command, but you can compute it via the QR decomposition.

```
> x1 <- rnorm(5)
> x3 <- rnorm(5)
> x2 <- x1 - x3  # linearly dependent
> X <- cbind(x1, x2, x3)
> qr(X)
$qr
           x1       x2          x3
[1,] -1.62622 -1.47354 -1.5268e-01
[2,]  0.68996 -1.80292  1.8029e+00
[3,] -0.48249  0.66549 -2.2204e-16
[4,] -0.26177  0.10631  0.0000e+00
[5,]  0.24166 -0.39548  0.0000e+00

$rank
[1] 2

$qraux
[1] 1.4053 1.6240 2.0000

$pivot
[1] 1 2 3

attr(,"class")
[1] "qr"
```

Chapter 3

Data Frames

Datasets in R are usually stored as `data.frame` objects. These are similar to matrices, but more flexible (and play more nicely with R's data analysis functions). This chapter will be an introduction to how to work with data frames and load datasets into R.

3.1 Basics

R comes with numerous datasets already loaded. We'll use one of these — `esoph`, which documents risk factors for esophageal cancer — to illustrate the basic features of data frames. To load the `esoph` data frame into your workspace, use the `data` command:[1]

```
> data(esoph)
```

To see the full list of built-in datasets:

```
> data()
```

The esoph data frame looks like this:

```
> esoph
   agegp     alcgp    tobgp ncases ncontrols
1  25-34 0-39g/day 0-9g/day      0        40
2  25-34 0-39g/day    10-19      0        10
3  25-34 0-39g/day    20-29      0         6
```

[1]Fun fact: users can overwrite functions that are built into R by creating variables with the same name. So don't be foolish and name your dataset `data` (or your matrix `matrix`, for that matter) when you're writing a script.

```
...
87    75+       120+ 0-9g/day       2          2
88    75+       120+    10-19       1          1
```

There are five variables. Notice that the first three are stored as text (actually, as we will see soon, as factors) while the last two are numbers. For practical purposes, this is the main difference between data frames and matrices in R. A matrix can only store one type of data, while a data frame can manage multiple types.

3.1.1 Extracting Data

Some of the basic commands we used for matrices also work for data frames.

```
> dim(esoph)  # dimensions
[1] 88  5
> nrow(esoph)  # number of rows
[1] 88
> esoph[1:4, ]  # indexing with brackets
  agegp     alcgp    tobgp ncases ncontrols
1 25-34 0-39g/day 0-9g/day      0        40
2 25-34 0-39g/day    10-19      0        10
3 25-34 0-39g/day    20-29      0         6
4 25-34 0-39g/day      30+      0         5
> esoph[30:33, 2]
[1] 120+      0-39g/day 0-39g/day 0-39g/day
Levels: 0-39g/day < 40-79 < 80-119 < 120+
> esoph[30:33, "alcgp"]  # can also use variable names
[1] 120+      0-39g/day 0-39g/day 0-39g/day
Levels: 0-39g/day < 40-79 < 80-119 < 120+
```

Use names to get the list of variable names without printing the whole dataset.

```
> names(esoph)
[1] "agegp"     "alcgp"     "tobgp"     "ncases"
[5] "ncontrols"
```

Remember how we used $ to extract elements from a list when we looked at the eigen and qr commands last chapter? A data frame is just a special kind of list, so we can use $ here to extract variables.

```
> esoph$ncases
 [1]  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  1  0
[19]  0  0  3  1  0  0  0  0  0  2  0  2  1  0  0  0  6  4
[37]  5  5  3  6  1  2  4  3  2  4  2  3  3  4  9  6  4  3
[55]  9  8  3  4  5  6  2  5  5  4  2  0 17  3  5  6  4  2
[73]  1  3  1  1  1  1  2  1  2  1  0  1  1  1  2  1
> sum(esoph$ncases + esoph$ncontrols)
[1] 1175
```

If you need to extract multiple variables at once and don't feel like typing `dataname$varname` a bunch of times, use the `with` command.

```
> with(esoph, sum(ncases + ncontrols))
[1] 1175
```

If you speak to students in the program who entered in 2009 or earlier, they may tell you to use the `attach` function for this kind of operation. *Don't do it!* I am not going to go through the `attach` function here, because its use often leads to code that isn't reproducible, errors that are hard to trace, and other obnoxious problems.

3.1.2 Summary Statistics

If you want to see what a dataset looks like without printing the whole thing to the R console, the commands `head`, `tail`, and `str` (as in "structure") come in handy.

```
> head(esoph)
  agegp     alcgp    tobgp ncases ncontrols
1 25-34 0-39g/day 0-9g/day      0        40
2 25-34 0-39g/day    10-19      0        10
3 25-34 0-39g/day    20-29      0         6
4 25-34 0-39g/day      30+      0         5
5 25-34     40-79 0-9g/day      0        27
6 25-34     40-79    10-19      0         7
> tail(esoph)
```

```
   agegp  alcgp    tobgp ncases ncontrols
83   75+  40-79    20-29      0         3
84   75+  40-79      30+      1         1
85   75+ 80-119 0-9g/day      1         1
86   75+ 80-119    10-19      1         1
87   75+   120+ 0-9g/day      2         2
88   75+   120+    10-19      1         1
> str(esoph)
'data.frame':        88 obs. of  5 variables:
$ agegp : Ord.factor w/ 6 levels "25-34"<"35-44"<..: 1 1 1
   1 1 1 1 1 1 1 ...
$ alcgp : Ord.factor w/ 4 levels "0-39g/day"<"40-79"<..: 1
   1 1 1 2 2 2 2 3 3 ...
$ tobgp : Ord.factor w/ 4 levels "0-9g/day"<"10-19"<..: 1 2
   3 4 1 2 3 4 1 2 ...
$ ncases : num 0 0 0 0 0 0 0 0 0 0 ...
$ ncontrols: num 40 10 6 5 27 7 4 7 2 1 ...
```

The summary command gives you basic summary statistics about the variables in a data frame.

```
> summary(esoph)
   agegp          alcgp          tobgp         ncases
 25-34:15   0-39g/day:23   0-9g/day:24   Min.   : 0.00
 35-44:15   40-79    :23   10-19   :24   1st Qu.: 0.00
 45-54:16   80-119   :21   20-29   :20   Median : 1.00
 55-64:16   120+     :21   30+     :20   Mean   : 2.27
 65-74:15                                3rd Qu.: 4.00
 75+  :11                                Max.   :17.00
   ncontrols
 Min.   : 1.0
 1st Qu.: 3.0
 Median : 6.0
 Mean   :11.1
 3rd Qu.:14.0
 Max.   :60.0
```

The Hmisc package provides a describe command that is a slightly more full-featured version of summary. To load the package:

```
> library(Hmisc)
```

(Remember that R is case-sensitive, so `library(hmisc)` won't work.) If this results in an error, you probably need to install the package:

```
> install.packages("Hmisc")
```

Now you can use the `describe` command:

```
> describe(esoph)
esoph

 5  Variables      88  Observations
------------------------------------------------------------
agegp
      n missing  unique
     88       0       6

          25-34 35-44 45-54 55-64 65-74 75+
Frequency    15    15    16    16    15  11
%            17    17    18    18    17  12
------------------------------------------------------------
alcgp
      n missing  unique
     88       0       4

0-39g/day (23, 26%), 40-79 (23, 26%)
80-119 (21, 24%), 120+ (21, 24%)
------------------------------------------------------------
tobgp
      n missing  unique
     88       0       4

0-9g/day (24, 27%), 10-19 (24, 27%)
20-29 (20, 23%), 30+ (20, 23%)
------------------------------------------------------------
ncases
      n missing  unique    Mean     .05     .10     .25
     88       0      10   2.273     0.0     0.0     0.0
    .50     .75     .90     .95
    1.0     4.0     5.3     6.0
```

```
           0  1  2  3 4 5 6 8 9 17
Frequency 29 16 11  9 8 6 5 1 2  1
%         33 18 12 10 9 7 6 1 2  1
------------------------------------------------------------
ncontrols
      n missing  unique    Mean     .05     .10     .25
     88       0      30   11.08     1.0     1.0     3.0
    .50     .75     .90     .95
    6.0    14.0    29.1    40.0

lowest :  1  2  3  4  5, highest: 40 46 48 49 60
------------------------------------------------------------
```

3.1.3 Manipulating Data

If you want to change a bunch of stuff at once within a data frame, the `transform` command is your friend. Let's create a simple data frame:

```
> ourData <- data.frame(var1 = 1:5, var2 = 2:6, var3 = 10:6)
> ourData
  var1 var2 var3
1    1    2   10
2    2    3    9
3    3    4    8
4    4    5    7
5    5    6    6
```

To rescale the first two variables by 1/10:

```
> ourData <- transform(ourData,
+                      var1 = var1 / 10,
+                      var2 = var2 / 10)
> ourData
  var1 var2 var3
1  0.1  0.2   10
2  0.2  0.3    9
3  0.3  0.4    8
4  0.4  0.5    7
5  0.5  0.6    6
```

To create three new variables out of combinations of the first three:

```
> ourData <- transform(ourData,
+                      var4 = var1 + var2,
+                      var5 = var1 + var3,
+                      var6 = var2 + var3)
> ourData
  var1 var2 var3 var4 var5 var6
1  0.1  0.2   10  0.3 10.1 10.2
2  0.2  0.3    9  0.5  9.2  9.3
3  0.3  0.4    8  0.7  8.3  8.4
4  0.4  0.5    7  0.9  7.4  7.5
5  0.5  0.6    6  1.1  6.5  6.6
```

And so on.

3.2 Factors

The first three columns of esoph contain a type of data we haven't dealt with yet: *factor* variables. This is how R stores categorical data. Since there is virtually no plausible regression model in political science[2] that doesn't include some kind of categorical control (e.g., race of a survey respondent, regime type of a country), you should know how to use factors.

When you print the contents of a factor variable to the R console, you see text followed by a list of "levels":

```
> head(esoph$alcgp)
[1] 0-39g/day 0-39g/day 0-39g/day 0-39g/day 40-79
[6] 40-79
Levels: 0-39g/day < 40-79 < 80-119 < 120+
```

This may lead you to believe that factors are just character variables. This is false! To find out whether R thinks a variable containing text is a factor, check its class:

```
> class(esoph$alcgp)  # is a factor
[1] "ordered" "factor"
```

[2]To be more precise, no regression model that could plausibly get three reviewers for a political science journal to sign off on it.

```
> x <- c("a", "a", "b", "c", "a")
> class(x)  # not a factor
[1] "character"
```

Luckily, you can easily convert a character variable to a factor with `as.factor`.

```
> xfactor <- as.factor(x)
> xfactor
[1] a a b c a
Levels: a b c
> class(xfactor)
[1] "factor"
```

If the difference between factors and characters seems unimportant to you now, don't worry. You will thank me later — possibly much later — when you're using some kind of non-standard regression model with categorical data and it throws up an indecipherable error message about how it doesn't recognize character variables.

We'll spend more time with factors in our next session when we start running regressions in R. Until then, here are some basic things you can do with factors. Use `levels` and `nlevels` to get the names of the levels and how many there are.

```
> levels(xfactor)
[1] "a" "b" "c"
> nlevels(xfactor)
[1] 3
```

`table` lets you see how many instances there are within each level.

```
> table(xfactor)
xfactor
a b c
3 1 1
```

The `table` command actually also works for other kinds of variables too.

```
> ## random draws from Binomial(6, 0.5)
> y <- rbinom(1000, size = 6, prob = 0.5)
> table(y)
```

```
y
  0   1   2   3   4   5   6 
 14 110 234 313 228  87  14 
> table(y) / 1000  # estimated probabilities
y
    0     1     2     3     4     5     6 
0.014 0.110 0.234 0.313 0.228 0.087 0.014 
```

3.3 Loading and Saving Data

From last session's material, you already know how to `load` and `save` R objects. So if there is a data frame stored as R data (typically in a `.rda` or `.RData` file), you know what to do. Sadly, approximately 0% of all political science datasets are distributed in this format. So let's see what to do with two popular formats: comma-separated values (CSV) and Stata `.dta` files.

3.3.1 Comma-Separated Values

This is generally the format you should distribute data in, since it can be read by any spreadsheet or statistical program. A CSV file is simply a text file where each row is a row of data, where fields are separated by commas. A typical CSV file looks something like this:

```
"var1","var2","var3","var4"
1.25,2.75,3.45,"D"
4.1,2.886,3.99,"A"
8.23,1.789,2.3,"B"
...
2.212,1.2,3.917,"A"
```

To load a dataset in CSV format, use the `read.csv` command. For example, download the file `data1.csv` from the course website (`http://www.bkenkel.com/teaching.html`) to your `Z:` drive,[3] then run:

```
> data1 <- read.csv("z:/data1.csv")
> data1
```

[3]Or, if you're a laptop-using scofflaw, to your working directory: `getwd()`.

```
            x       y z
1  -0.262011 0.20830 a
2  -0.168523 0.01383 b
3  -0.226056 1.74640 c
4   0.651193 0.31098 d
5   0.637790 1.12686 e
6   0.083914 0.39382 e
7  -0.136230 0.51429 d
8  -0.450585 0.96430 c
9   0.341704 1.08622 b
10 -2.120221 1.22373 a
```

Notice that, unlike the `load` command we used last time, you must assign the data frame to a variable. Otherwise, if you run `read.csv` without making an assignment, it just prints the data frame to the R console — typically not what you want.

To save a data frame in CSV format, use `write.csv`. Unless you've created row names via the `rownames` command, you typically want to set `row.names = FALSE`.

```
> write.csv(data1, file = "z:/mydata1.csv", row.names = FALSE)
```

3.3.2 Stata `.dta`

To read in Stata files, you need to load up the `foreign` package and use its `read.dta` command. For this example, download the file `data2.dta` from the course website.

```
> library(foreign)
> data2 <- read.dta("z:/data2.dta")
> data2
  var1      var2  var3     var4
1    0 -0.239443  bear  0.83674
2    1  1.211068 moose -0.38043
3    1 -9.263507 pizza  0.64275
4    0  0.098341 pizza -0.19735
5    1  0.436587 moose  1.19144
6    0  0.207671  bear  2.10379
7    0 -2.340137 moose  0.28661
8    0 -2.240873  bear -0.55143
```

```
9     1  0.217004  bear -0.16988
10    1  0.264562 pizza -0.83495
```

For the sake of completeness, I'll mention that you can save data frames in `.dta` format using `write.dta`. However, I don't know *why* you would want to do this: CSV files can be used by anyone anywhere.

If a `.dta` file comes from a new edition of Stata, you may receive an error causing `read.dta` to fail. Here are the steps I recommend taking if this happens:

1. Check that you are running the latest version of R
2. Run the command

   ```
   > old.packages()
   ```

   to see which of your packages needs to be updated. If `foreign` appears in the list, run

   ```
   > update.packages(oldPkgs = "foreign")
   ```

   to update it.
3. Try `read.dta` again
4. See if the `?read.dta` help page has any insight about the Stata data file version you're trying to load from
5. If all this has failed, go to THE STAR LAB, open the data file in Stata, and export it to CSV format

Chapter 4

Plots

We now will start working with R's excellent graphics capabilities. I should start with a disclaimer that applies to all of the segments of this course, but especially this one: we're only going to be scratching the surface of this topic. The base graphics engine in R has enough capabilities that an entire course could be devoted to it alone, and add-on graphics packages — notably ggplot2, which has quickly won the affection of data visualization nerds everywhere — can do even more.

The best way to learn about plotting in R is to read the help pages and experiment with the various options available. There are also websites with sample graphs and the code to produce them, such as http://addictedtor.free.fr/graphiques/allgraph.php.

4.1 Scatterplots

For today's examples, we'll use the Duncan data on occupational prestige from the car package.

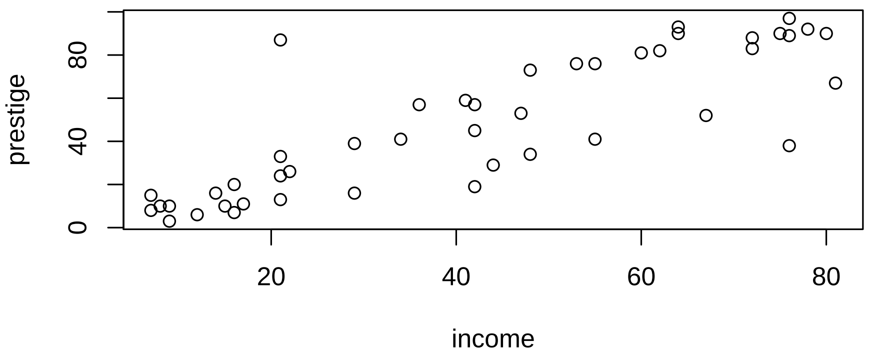
```
> library(car)  # scofflaws may need to install
> data(Duncan)
> str(Duncan)
'data.frame':        45 obs. of  4 variables:
$ type : Factor w/ 3 levels "bc","prof","wc": 2 2 2 2 2 2 2
   2 3 2 ...
$ income : int 62 72 75 55 64 21 64 80 67 72 ...
$ education: int 86 76 92 90 86 84 93 100 87 86 ...
$ prestige : int 82 83 90 76 90 87 93 90 52 88 ...
```

As we learned last time, we can access the variables within `Duncan` using syntax like `Duncan$variable` or `with(Duncan, variable)`. But let's just make things easy on ourselves for the rest of this session and make global assignments of the key variables.

```
> income <- Duncan$income
> prestige <- Duncan$prestige
> type <- Duncan$type
```

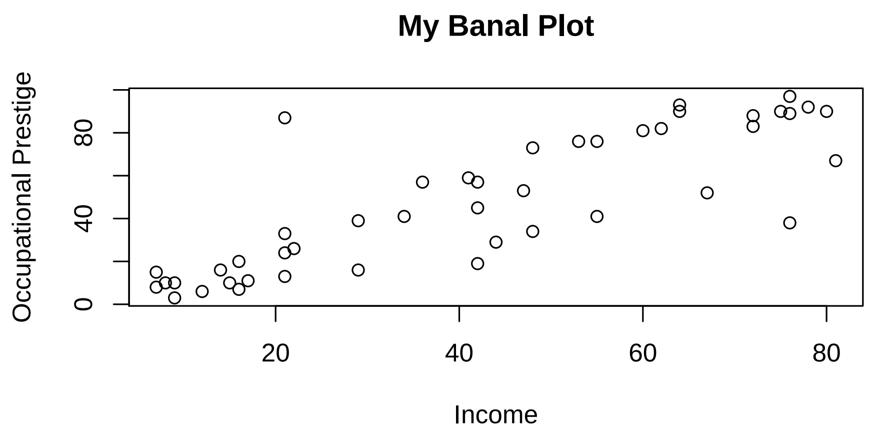
To make a scatterplot with income on the x-axis and prestige on the y-axis, use the `plot` command:

```
> plot(income, prestige)
```



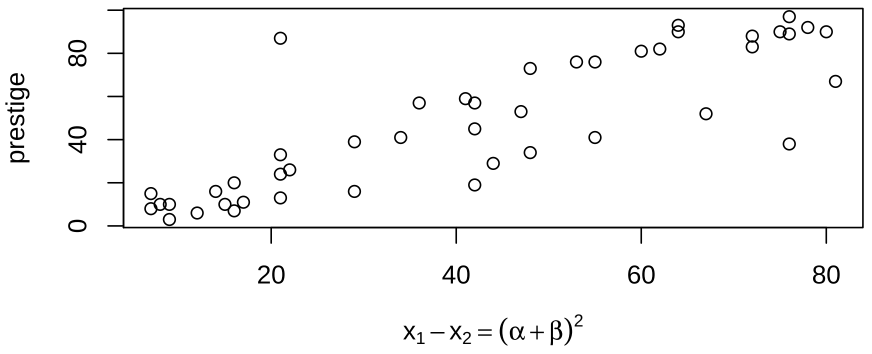
You can set the axis labels and plot title using the `xlab`, `ylab`, and `main` arguments.

```
> plot(income, prestige,
+      xlab = "Income", ylab = "Occupational Prestige",
+      main = "My Banal Plot")
```



If you want to use Greek letters, sub/superscripts, or other mathematical features, there's a LaTeX-like syntax described in ?plotmath. To wit:

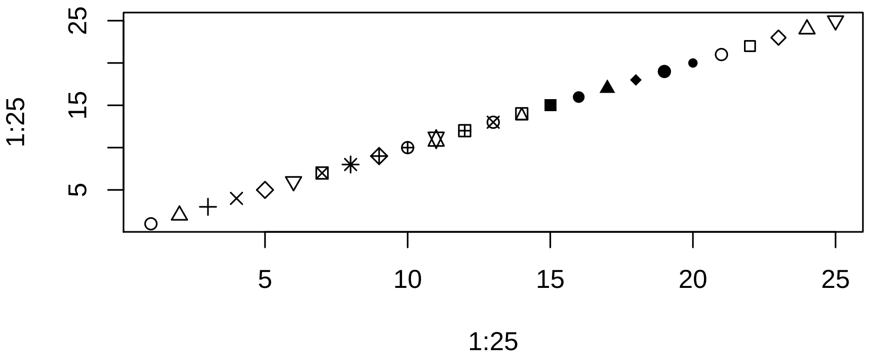
```
> plot(income, prestige,
+      xlab = expression(x[1] - x[2] == (alpha + beta)^2))
```



There are many other arguments to change basic features of the plot like the axis limits. Most of these arguments are described in ?plot.default.

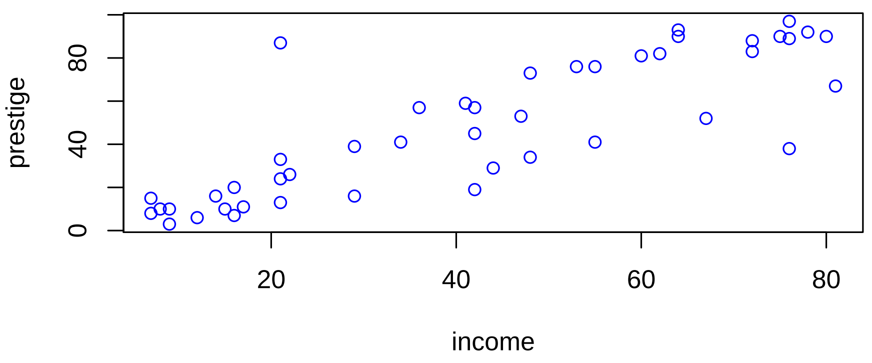
You can use the pch argument to change the type of points used in a scatterplot.

```
> plot(1:25, 1:25, pch = 1:25)
```



The col option can be used to change the color of the points.

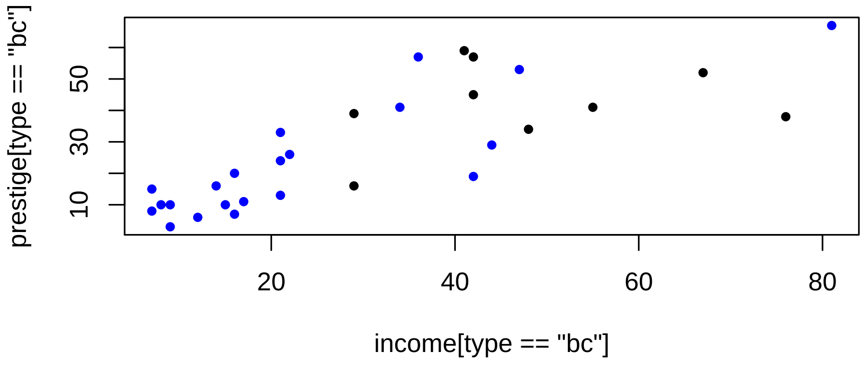
```
> plot(income, prestige, col = "blue")
```



There are, of course, many more graphical options. These can be found in the ?par help page.

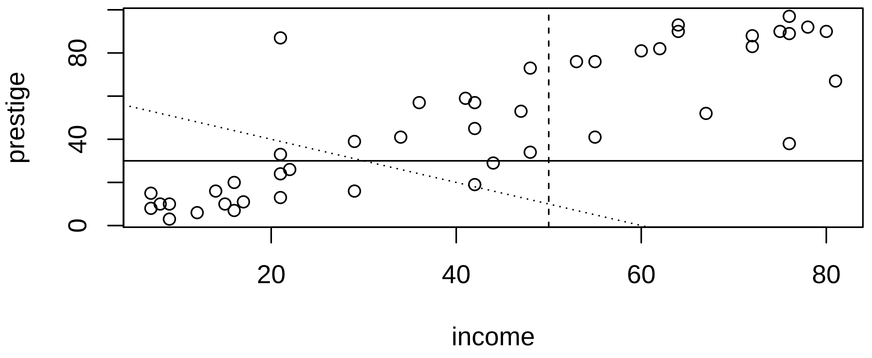
You can add points to an existing plot using the points command. Let's use this to show blue collar workers in blue and others in black.

```
> plot(income[type == "bc"], prestige[type == "bc"],
+      pch = 20, col = "blue")
> points(income[type != "bc"], prestige[type != "bc"],
+        pch = 20, col = "black")
```



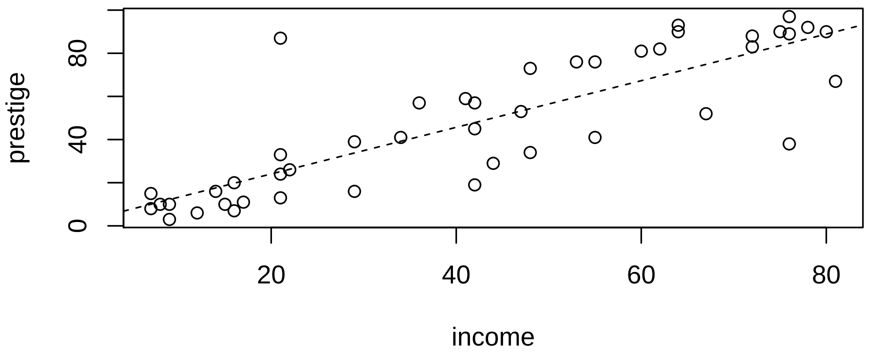
You can superimpose lines using the abline command. The lty option controls whether the line is solid, dotted, etc.

```
> plot(income, prestige)
> abline(h = 30, lty = 1)  # horizontal
> abline(v = 50, lty = 2)  # vertical
> abline(a = 60, b = -1, lty = 3)  # slope-intercept
```



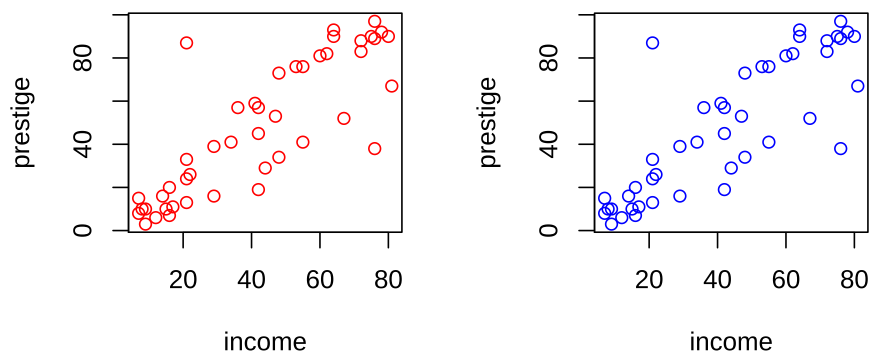
To superimpose a bivariate regression line, use `abline(lm(y  x))`. (We'll be talking more about the `lm` command, which runs linear regressions, very soon.)

```
> plot(income, prestige)
> abline(lm(prestige ~ income), lty = 2)
```



One more thing you might want to do is put multiple plots together in a single graphic. To do this, run `par(mfrow = c(nrow, ncol))` and then make your plots.

```
> par(mfrow = c(1, 2))
> plot(income, prestige, col = "red")
> plot(income, prestige, col = "blue")
```



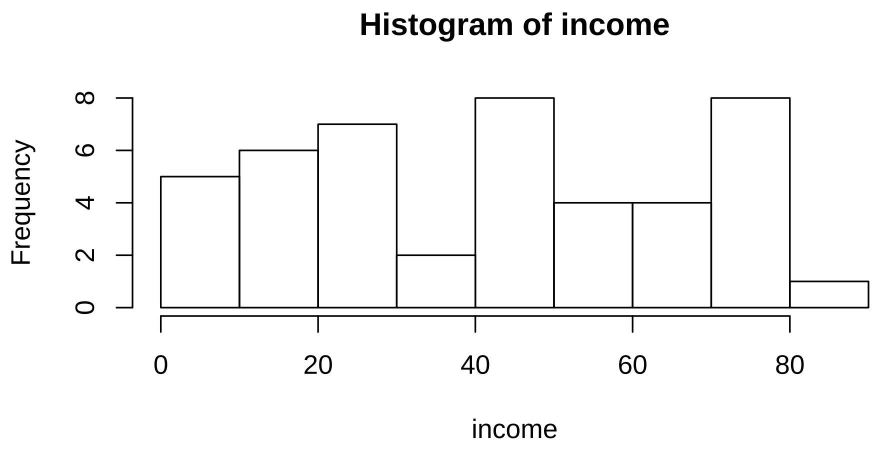
You can change other low-level graphical elements, such as the plot margins, using `par`.

4.2 Other Plots: A Quick Cookbook

Scatterplots aren't all you can make in R. Below I'm going to give just a few basic examples of other types of plots, mainly so you know what the commands are called and can look at their help pages. Most of the graphical options we met above, like `pch` and `lty`, carry over to these plots too.

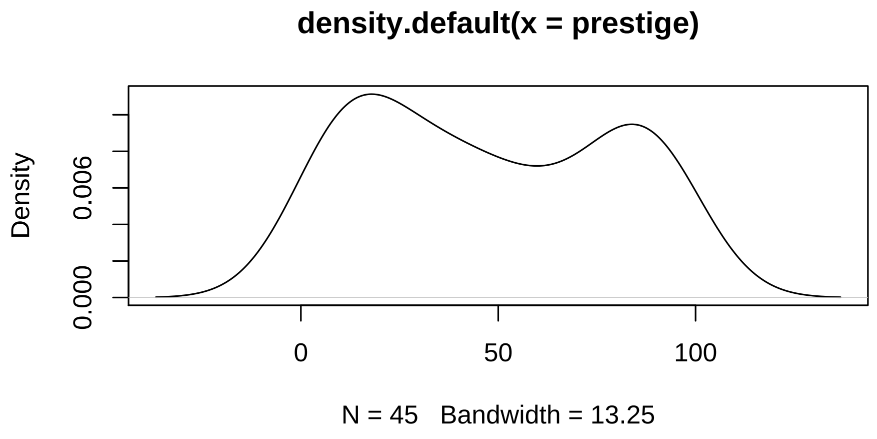
4.2.1 Histogram

```
> hist(income)
```



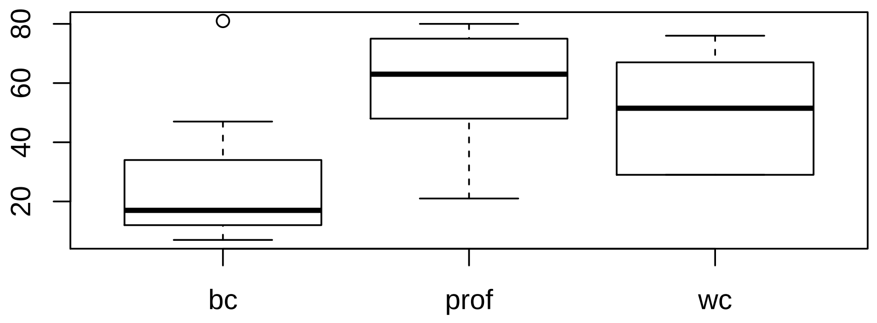
4.2.2 Density Plot

```
> plot(density(prestige))
```



4.2.3 Box Plot

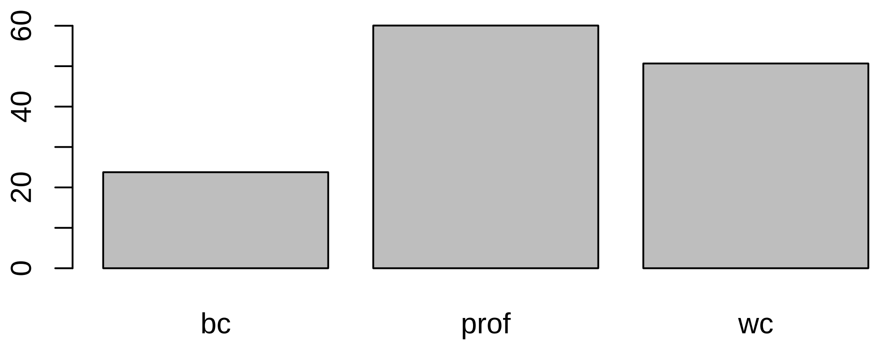
```
> plot(type, income)
```



4.2.4 Bar Plot

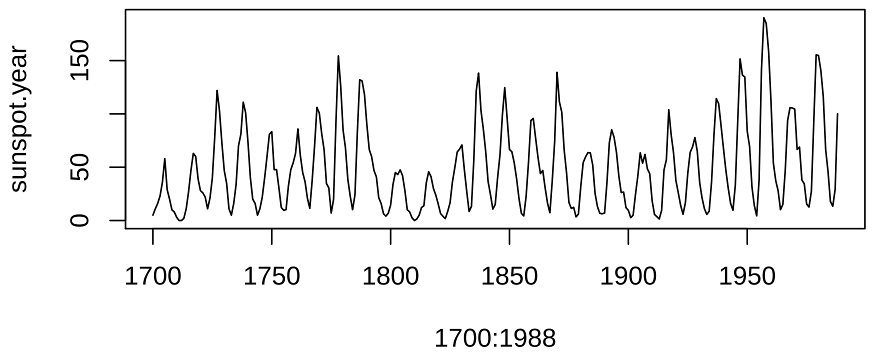
(aka the box plot's uglier, less informative cousin)

```
> avg.bc <- mean(income[type == "bc"])
> avg.prof <- mean(income[type == "prof"])
> avg.wc <- mean(income[type == "wc"])
> avg <- c(avg.bc, avg.prof, avg.wc)
> barplot(avg, names = levels(type))
```



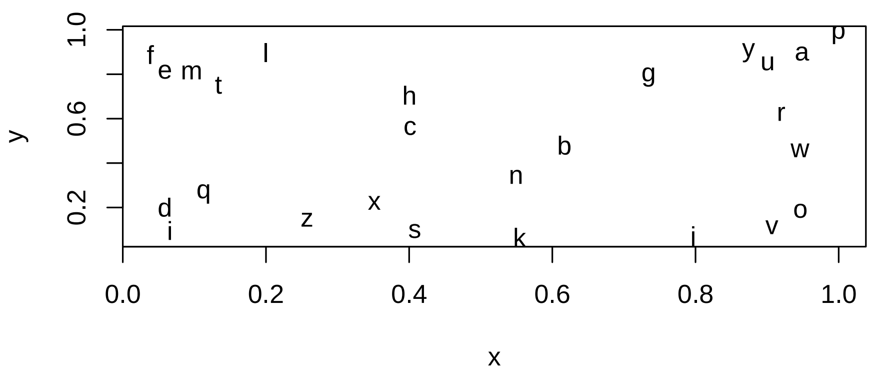
4.2.5 Line Plot

```
> data(sunspot.year)
> plot(1700:1988, sunspot.year, type = "l")
```



4.2.6 Text Plot

```
> x <- runif(26)
> y <- runif(26)
> plot(x, y, type = "n")  # set up axes without placing points
> text(x, y, labels = letters)
```



4.3 Exporting Graphics

To place a graphic in a research paper, you need to export it to a format that LaTeX can read, typically PDF. The best way to make PDFs of R graphics is the `pdf` command:

```
> pdf(file = "mygraph1.pdf")
> plot(income, prestige)
> abline(lm(prestige ~ income))
> dev.off()
```

Here's roughly how the sequence of commands works:

1. The `pdf` command opens a connection to the PDF graphics device.
2. The graphics device executes your plotting commands (just like the standard interactive graphics device does when you run plotting commands without running `pdf()` first).
3. `dev.off()` tells the graphics device to shut down and write its contents to the file you originally specified.

You can change the width or height of the output PDF using the `width` and `height` arguments.

```
> pdf(file = "mygraph2.pdf", width = 10, height = 3)
> plot(income, prestige)
> dev.off()
```

Many file types other than PDF are supported; see `?Devices` for a list. If you create a plot with tens of thousands of points, it is often wise to use a device other than PDF to ensure that the resulting file isn't unduly large.

Chapter 5

Linear Regression

We now arrive at what you'll primarily be using `R` for: data analysis, particularly multivariate regression modeling. In this chapter we'll look at the basic features of `lm`, the linear modeling command. Future chapters will cover more advanced regression techniques.

5.1 Running a Regression

To run a regression in `R`, use the `lm` ("linear model") command. This command has a number of arguments, the most important of which are:

- `formula` The model formula, generally of the form `y ~ x1 + x2 + ...`, where `y` is the response variable and the x's are regressors. We'll go through the intricacies of formula syntax in just a bit.
- `data` The data frame to take the variables from. If none is specified, `lm` looks for the variables in the global environment (i.e., the objects you see when you run `ls()`).
- `subset` A logical statement indicating a subset of the data to include in the regression.
- `weights` A variable containing weights for each observation.
- `na.action` How to deal with `NA`s (missing data); the default is to delete any observation with missingness.

We'll use the `Highway1` dataset from the `car` package to illustrate the `lm` function.

```
> library(car)
> data(Highway1)
> str(Highway1)
'data.frame':        39 obs. of  12 variables:
$ rate : num 4.58 2.86 3.02 2.29 1.61 6.87 3.85 6.12 3.29
   5.88 ...
$ len : num 4.99 16.11 9.75 10.65 20.01 ...
$ ADT : int 69 73 49 61 28 30 46 25 43 23 ...
$ trks : int 8 8 10 13 12 6 8 9 12 7 ...
$ sigs1: num 0.2004 0.0621 0.1026 0.0939 0.05 ...
$ slim : int 55 60 60 65 70 55 55 55 50 50 ...
$ shld : int 10 10 10 10 10 10 8 10 4 5 ...
$ lane : int 8 4 4 6 4 4 4 4 4 4 ...
$ acpt : num 4.6 4.4 4.7 3.8 2.2 24.8 11 18.5 7.5 8.2 ...
$ itg : num 1.2 1.43 1.54 0.94 0.65 0.34 0.47 0.38 0.95
   0.12 ...
$ lwid : int 12 12 12 12 12 12 12 12 12 12 ...
$ hwy : Factor w/ 4 levels "FAI","MA","MC",..: 1 1 1 1 1 4
   4 4 4 4 ...
```

The response variable is `rate`, the number of accidents per million vehicle miles in 1973. We'll look at the following independent variables:

`ADT`	Average daily traffic count in thousands
`trks`	Truck volume as a percent of the total volume
`slim`	Speed limit in 1973
`shld`	Width in feet of the outer shoulder

To regress the accident rate on these variables, we run:

```
> modelMain <- lm(rate ~ ADT + trks + slim + shld,
+                  data = Highway1)
> modelMain
Call:
lm(formula = rate ~ ADT + trks + slim + shld, data = Highway1)

Coefficients:
(Intercept)          ADT         trks         slim
   17.49702      0.00994     -0.27612     -0.20183
       shld
   -0.01172
```

Notice that we have assigned the regression output to a variable, `modelMain`, so that we can further analyze the results later without having to run the regression again. If we only cared about the relationship among 4-lane highways, we would use the `subset` argument:

```
> model4lane <- lm(rate ~ ADT + trks + slim + shld,
+                  data = Highway1, subset = lane == 4)
> model4lane
Call:
lm(formula = rate ~ ADT + trks + slim + shld, data = Highway1,
    subset = lane == 4)

Coefficients:
(Intercept)          ADT         trks         slim
  18.081417     0.000643    -0.394214    -0.188329
       shld
  -0.027159
```

We could have done this even more easily with `update`.

```
> update(modelMain, subset = lane == 4)
Call:
lm(formula = rate ~ ADT + trks + slim + shld, data = Highway1,
    subset = lane == 4)

Coefficients:
(Intercept)          ADT         trks         slim
  18.081417     0.000643    -0.394214    -0.188329
       shld
  -0.027159
```

5.1.1 Formula Syntax

If you want to use interaction terms, transform some variables, or drop the intercept, you need to specify the formula slightly differently.

To drop the intercept:

```
> y ~ x1 + x2 - 1
```

To include x_2^2:

```
> y ~ x1 + x2 + I(x2^2)
```

To force x_1 and x_2 to have the same coefficient:

```
> y ~ I(x1 + x2)
```

To log-transform x_2:

```
> y ~ x1 + log(x2)
```

To include x_1, x_2, and $x_1 \times x_2$:

```
> y ~ x1*x2
```

To include only $x_1 \times x_2$:

```
> y ~ x1:x2
```

5.2 Analyzing the Output

The output of `lm` is a list that contains many objects.

```
> names(modelMain)
 [1] "coefficients"  "residuals"     "effects"
 [4] "rank"          "fitted.values" "assign"
 [7] "qr"            "df.residual"   "xlevels"
[10] "call"          "terms"         "model"
```

You can see a description of each of these in the “Value” section of `?lm`. There are also a number of functions that extract key features of `lm` output and other fitted regression models (e.g., logistic regressions) in R.

```
> coef(modelMain)  # coefficients
(Intercept)         ADT        trks        slim        shld
 17.4970178   0.0099353  -0.2761234  -0.2018336  -0.0117242
> vcov(modelMain)  # variance-covariance matrix
            (Intercept)         ADT        trks        slim
(Intercept)   6.1291692 -3.5606e-03  0.00818296 -1.2870e-01
ADT          -0.0035606  1.8240e-04  0.00010338  5.5246e-05
trks          0.0081830  1.0338e-04  0.01062959 -2.3340e-03
```

```
slim         -0.1287008  5.5246e-05 -0.00233396  3.2779e-03
shld          0.1441658 -5.8508e-04  0.00275733 -4.4942e-03
                   shld
(Intercept)  0.14416576
ADT         -0.00058508
trks         0.00275733
slim        -0.00449421
shld         0.01291617
> residuals(modelMain)  # residuals
        1         2         3         4         5         6
-0.175471 -0.926044  0.024649  1.012964  1.393873  1.949757
        7         8         9        10        11        12
-0.700409  2.077804 -1.182174  0.237638  0.358472 -0.485482
       13        14        15        16        17        18
 1.175351  0.580589 -1.924314 -1.013222 -0.963417 -0.167050
       19        20        21        22        23        24
-1.476058 -0.906117 -0.971798 -0.800252 -2.046108  0.597245
       25        26        27        28        29        30
 1.258006  1.736527  4.047067 -0.733941  0.914309 -0.615360
       31        32        33        34        35        36
-0.440085 -0.772885  0.170593 -1.198569  0.160369 -2.419257
       37        38        39
 0.948152  0.764346  0.510304
> fitted(modelMain)  # fitted values
      1       2       3       4       5       6       7
4.75547 3.78604 2.99535 1.27704 0.21613 4.92024 4.55041
      8       9      10      11      12      13      14
4.04220 4.47217 5.64236 3.84153 5.09548 3.62465 3.26941
     15      16      17      18      19      20      21
4.61431 3.00322 2.97342 4.38705 4.23606 3.45612 2.86180
     22      23      24      25      26      27      28
3.14025 4.87611 1.21275 7.97199 6.86347 4.16293 3.66394
     29      30      31      32      33      34      35
6.56569 3.18536 6.21009 3.67289 2.79941 3.03857 3.61963
     36      37      38      39
5.17926 3.32185 2.28565 3.60970
```

For better or worse, you will usually want to see the typical "regression table" of the coefficients with their standard errors and *p*-values, along with the R^2 and some other summary statistics. You can get this using the summary command:

```
> summary(modelMain)
Call:
lm(formula = rate ~ ADT + trks + slim + shld, data = Highway1)

Residuals:
   Min     1Q Median     3Q    Max 
-2.419 -0.916 -0.167  0.839  4.047 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 17.49702    2.47572    7.07  3.7e-08
ADT          0.00994    0.01351    0.74   0.4670
trks        -0.27612    0.10310   -2.68   0.0113
slim        -0.20183    0.05725   -3.53   0.0012
shld        -0.01172    0.11365   -0.10   0.9184

Residual standard error: 1.37 on 34 degrees of freedom
Multiple R-squared:  0.577,        Adjusted R-squared:  0.527 
F-statistic: 11.6 on 4 and 34 DF,  p-value: 4.83e-06
```

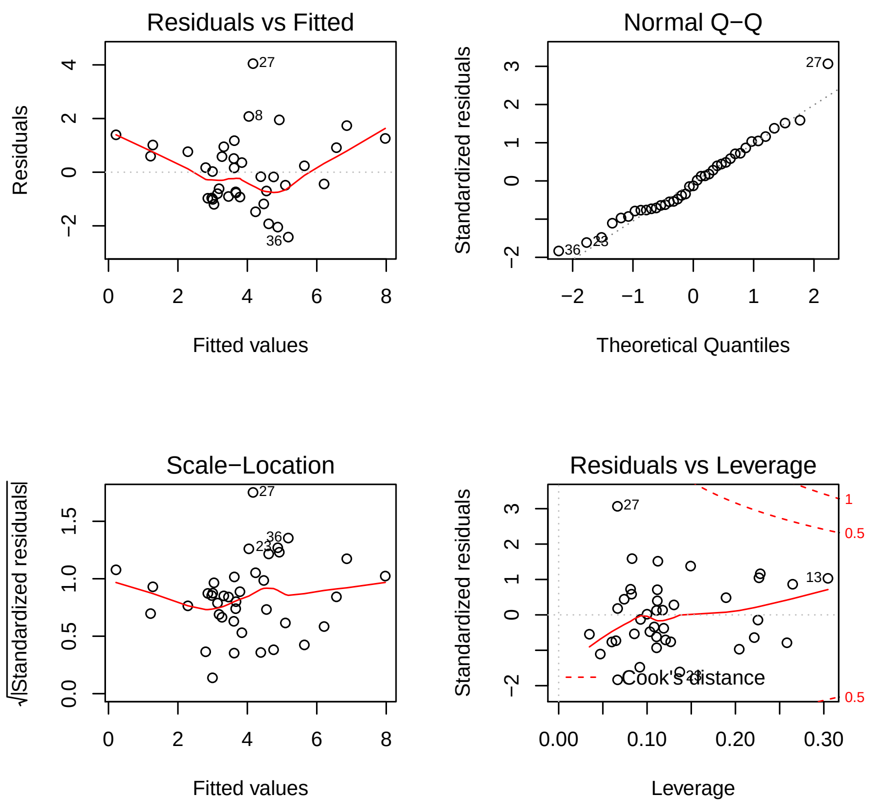
If you want, you can see "significance stars" by setting `options(show.signif.stars = TRUE)`. (Or, sadly, you probably don't have to do anything, since this was the default option in R the last time I checked.) Note that the output of summary is itself a list:

```
> names(summary(modelMain))
 [1] "call"          "terms"         "residuals"    
 [4] "coefficients"  "aliased"       "sigma"        
 [7] "df"            "r.squared"     "adj.r.squared"
[10] "fstatistic"    "cov.unscaled" 
```

You can use this to extract $\hat{\sigma}$ or the R^2 values.

As you may or may not find out in PSC 405, there are some regression diagnostics (e.g., checking for heteroscedasticity or outlier influence) that are best accomplished graphically. R provides a plot method for lm objects so that you can run these tests easily.

```
> par(mfrow = c(2, 2))
> plot(modelMain)
```



See ?plot.lm for descriptions of these plots and appropriate references.

5.3 Presenting Results

You're eventually going to want to get your regression out of R and into a LaTeX paper or a presentation. Here are some ways to make that easier for yourself.

5.3.1 Tables

You can automatically generate LaTeX code for the archetypical regression table using the xtable package.

```
> library(xtable)  # scofflaws may need to install
> xtable(modelMain)
% latex table generated in R 3.0.2 by xtable 1.7-1 package
% Mon Feb 17 11:29:20 2014
\begin{table}[ht]
```

```
\centering
\begin{tabular}{rrrrr}
  \hline
 & Estimate & Std. Error & t value & Pr($>$$|$t$|$) \\
  \hline
(Intercept) & 17.4970 & 2.4757 & 7.07 & 0.0000 \\
  ADT & 0.0099 & 0.0135 & 0.74 & 0.4670 \\
  trks & -0.2761 & 0.1031 & -2.68 & 0.0113 \\
  slim & -0.2018 & 0.0573 & -3.53 & 0.0012 \\
  shld & -0.0117 & 0.1136 & -0.10 & 0.9184 \\
   \hline
\end{tabular}
\end{table}
```

There are many options you can use to control the look and feel of the output; see `?xtable` and `?print.xtable`.

```
> xt <- xtable(modelMain, digits = 2, label = "tab:xt",
+              caption = "\\texttt{xtable} output")
> print(xt,
+       math.style.negative = TRUE,
+       table.placement = "t")
% latex table generated in R 3.0.2 by xtable 1.7-1 package
% Mon Feb 17 11:29:20 2014
\begin{table}[t]
\centering
\begin{tabular}{rrrrr}
  \hline
 & Estimate & Std. Error & t value & Pr($>$$|$t$|$) \\
  \hline
(Intercept) & 17.50 & 2.48 & 7.07 & 0.00 \\
  ADT & 0.01 & 0.01 & 0.74 & 0.47 \\
  trks & $-$0.28 & 0.10 & $-$2.68 & 0.01 \\
  slim & $-$0.20 & 0.06 & $-$3.53 & 0.00 \\
  shld & $-$0.01 & 0.11 & $-$0.10 & 0.92 \\
   \hline
\end{tabular}
\caption{\texttt{xtable} output}
\label{tab:xt}
\end{table}
```

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	17.50	2.48	7.07	0.00
ADT	0.01	0.01	0.74	0.47
trks	−0.28	0.10	−2.68	0.01
slim	−0.20	0.06	−3.53	0.00
shld	−0.01	0.11	−0.10	0.92

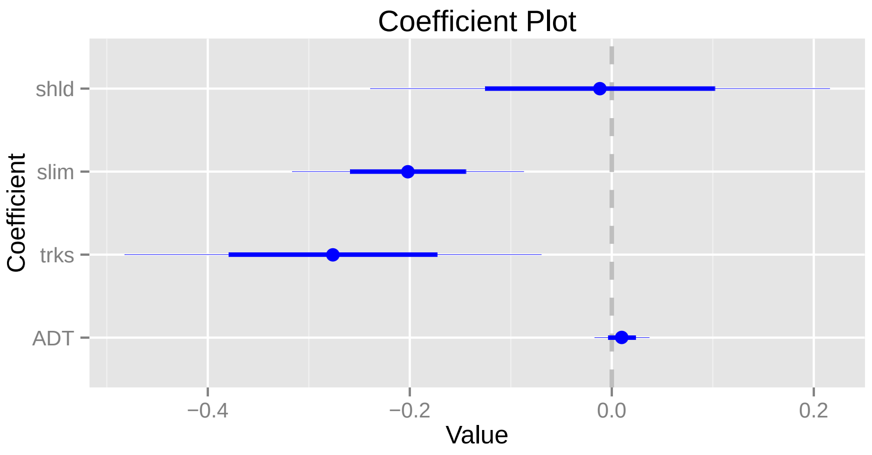
Table 5.1: `xtable` output

See Figure 5.1 for how the output looks. If you want to be real fancy, you can use the `file` option of `print.xtable` to write the output directly to a file, then use `\input` in LaTeX to read it in. This way, if you change your R script and your results change, your table will automatically change as well.

5.3.2 Coefficient Plots

One of the new signaling devices among job candidates is to present regression results graphically (instead of in a table) in their job talk. You may as well learn how to do this too. It's not too hard to hand-roll these plots, but a canned version is available in the `coefplot` package.[1]

```
> library(coefplot)  # may need to be installed
> coefplot(modelMain, intercept = FALSE)
```



[1]This plot looks different than the ones we made earlier because it uses the `ggplot2` package as a backend instead of base R graphics. You could teach a whole course on `ggplot2` itself, so it's beyond our scope here. However, there are many easy-to-follow tutorials about `ggplot2` online. I use `ggplot2` in all of my own work, and I encourage you to learn it on your own too.

The dots represent point estimates, and the lines represent two standard errors (roughly the 95% confidence interval) around each. So we can immediately see statistical signifiance at the ordinary[2] level by looking for coefficients whose bands don't cross 0. As always, there are tons of options to change the substance and look of the plot, which you can read about in `?coefplot` and `?coefplot.lm`.

[2] i.e., arbitrary

Chapter 6

Regression Tests and Diagnostics

Last time we saw how to fit regression models and perform some basic post-estimation analysis in R. Now we're going to see how to implement some more sophisticated tests and diagnostics.

All of the functions that we're looking at today come from the `car` and `lmtest` packages. As always, we only have enough time to scratch the surface. To see the full set of functions in each package, run the following:

```
> help(package = "car")
> help(package = "lmtest")
```

The CRAN Task View pages on Econometrics (`http://cran.r-project.org/web/views/Econometrics.html`) and Social Science (`http://cran.r-project.org/web/views/SocialSciences.html`) contain information about and links to many more packages that extend the basic regression functionality in R.

6.1 Robust Standard Errors

And by that I mean "standard errors that are estimated consistently even if some ordinary regression assumptions [usually homoscedasticity] are violated". In PSC 405, you've already seen the Huber–White variance–covariance matrix,

$$\hat{\Sigma} = (X'X)^{-1}X'\Omega X(X'X)^{-1}.$$

where Ω is a diagonal matrix with $\Omega_{ii} = e_i^2$. This is easy enough to compute yourself if you want. Let's illustrate with Chirot's data on the 1907 Romanian peasant rebellion, available as the `Chirot` data frame in the `car` package.

```
> library(car)
> data(Chirot)
> modelChirot <- lm(intensity ~ commerce + tradition + inequality,
+                   data = Chirot)
> X <- model.matrix(modelChirot)
> XtXi <- solve(t(X) %*% X)
> e <- residuals(modelChirot)
> vhat <- XtXi %*% t(X) %*% diag(e^2) %*% X %*% XtXi
> vhat
            (Intercept)    commerce  tradition inequality
(Intercept)  19.5939241  0.00709452 -0.2337984  0.3512013
commerce      0.0070945  0.00032496 -0.0001322 -0.0044422
tradition    -0.2337984 -0.00013220  0.0028836 -0.0155024
inequality    0.3512013 -0.00444224 -0.0155024  1.7936734
> vcov(modelChirot)
            (Intercept)    commerce   tradition inequality
(Intercept)   24.318483  0.02413357 -0.27973172 -2.1289424
commerce       0.024134  0.00039597 -0.00031876 -0.0122675
tradition     -0.279732 -0.00031876  0.00344230 -0.0031432
inequality    -2.128942 -0.01226753 -0.00314317  4.3335791
```

Though simple, it's tedious (and potentially error-prone) to compute this yourself. You can instead use the hccm function from the car package.

```
> vhat1 <- hccm(modelChirot, type = "hc0")
> vhat1
            (Intercept)    commerce  tradition inequality
(Intercept)  19.5939241  0.00709452 -0.2337984  0.3512013
commerce      0.0070945  0.00032496 -0.0001322 -0.0044422
tradition    -0.2337984 -0.00013220  0.0028836 -0.0155024
inequality    0.3512013 -0.00444224 -0.0155024  1.7936734
```

The type argument specifies how you want Ω to be computed. "hc0" stands for White's original formulation with e_i^2's along the diagonal; other options, such as the default "hc3", provide formulations that are also consistent against heteroscedasticity[1] but have arguably better finite-sample properties. See the references section in ?hccm, or in the almost identical ?sandwich::vcovHC,[2] for details on these alternatives.

[1]"hc" = "heteroscedasticity consistent"

[2]Remember that pkg::fn indicates the function fn in the package pkg.

Remember from last time that you can display the standard regression table by running `summary` on the output of `lm`. But what if you want the table to be shown with the robust standard errors? Use the function `coeftest` from the `lmtest` package:

```
> library(lmtest)
> coeftest(modelChirot, vcov = vhat1)
t test of coefficients:

            Estimate Std. Error t value Pr(>|t|)
(Intercept) -13.3474     4.4265   -3.02   0.0054
commerce      0.0910     0.0180    5.05  2.4e-05
tradition     0.1188     0.0537    2.21   0.0352
inequality    1.4981     1.3393    1.12   0.2728
```

6.2 Wald Tests

You have also seen in PSC 405 that the Wald test can be used to evaluate any linear hypothesis of the form

$$R\beta = r,$$

where R is $m \times k$. To jog your memory, the Wald statistic is

$$W = (R\hat{\beta} - r)'(R\hat{\Sigma}R')^{-1}(R\hat{\beta} - r),$$

which is asymptotically distributed χ^2_m. Once again, we have something that is possible to compute by hand but is better done automatically. Let's first simulate some data and run a regression.

```
> n <- 50
> x1 <- rnorm(n)
> x2 <- rnorm(n)
> x3 <- rnorm(n)
> y <- 1 + 2*x1 + 2*x2 + rnorm(n)
> modelSim <- lm(y ~ x1 + x2 + x3)
> modelSim
Call:
lm(formula = y ~ x1 + x2 + x3)

Coefficients:
```

```
(Intercept)          x1          x2          x3
      1.017       2.183       1.995      -0.316
```

Suppose we want to test the hypothesis that $\beta_1 = \beta_2$, which can be represented in matrix form with $R = \begin{bmatrix} 0 & 1 & -1 & 0 \end{bmatrix}$ and $r = [0]$.

```
> R <- matrix(c(0, 1, -1, 0), ncol = 4)
> b <- coef(modelSim)
> V <- vcov(modelSim)
> W <- t(R %*% b) %*% solve(R %*% V %*% t(R)) %*% (R %*% b)
> W
       [,1]
[1,] 1.1118
> 1 - pchisq(W, df = 1)
        [,1]
[1,] 0.29169
```

We can accomplish this much more easily with the linearHypothesis function from the car package.

```
> linearHypothesis(modelSim, "x1 = x2", test = "Chisq")
Linear hypothesis test

Hypothesis:
x1 - x2 = 0

Model 1: restricted model
Model 2: y ~ x1 + x2 + x3

  Res.Df  RSS Df Sum of Sq Chisq Pr(>Chisq)
1     47 35.2
2     46 34.4  1     0.831  1.11       0.29
```

We get the same test statistic and p-value without having to remember all that matrix algebra. If you want an F test instead of a χ^2 test, you can omit the test argument or set test = "F". You can also set white.adjust = "hc0" to use the robust variance–covariance matrix to compute the test statistic, or supply your own personal favorite $\hat{\Sigma}$ via the vcov argument.

The syntax of linearHypothesis is fairly flexible, and its help page contains numerous illustrations. Some examples of valid specifications:

```
> linearHypothesis(modelSim, "x1 - x2 = 0")
> linearHypothesis(modelSim, "x1 + x2 = 4")
> linearHypothesis(modelSim, c("x1 = x2", "x1 = x3"))
> linearHypothesis(modelSim, c("x1 = 0", "x2 = 2*x3"))
```

6.3 Additional Tests

Here are R implementations of some other diagnostic tests and statistics that you've covered in PSC 405. We'll set up with some simulated data that suffers from heteroscedastic, autocorrelated, non-normal disturbances.

```
> N <- 1000
> x1 <- rnorm(N)
> x2 <- rnorm(N)
> phi <- 0.5
> e <- arima.sim(list(ar = phi), n = 1000, rand.gen = rlnorm)
> e <- abs(x1) * e
> y <- 1 + x1 - x2 + e
> modelOLS <- lm(y ~ x1 + x2)
> modelOLS
Call:
lm(formula = y ~ x1 + x2)

Coefficients:
(Intercept)           x1           x2
      3.614        0.854       -1.092
```

AIC and BIC. These are included in base R and can be run on virtually any statistical model.

```
> AIC(modelOLS)
[1] 5058.9
> BIC(modelOLS)
[1] 5078.6
```

Breusch-Pagan and White tests. The Breusch-Pagan test is included via the function `bptest` in the `lmtest` package.

```
> library(lmtest)
> bptest(modelOLS)
        studentized Breusch-Pagan test

data:  modelOLS
BP = 0.6278, df = 2, p-value = 0.7306
```

The White test is a special case of the Breusch-Pagan test and can be implemented with an additional argument to the `bptest` function. Unfortunately, when you have a large number of variables the manual implementation can be somewhat tedious, but there does not appear to exist a completely canned version in a reputable package.

```
> bptest(modelOLS, ~ x1*x2 + I(x1^2) + I(x2^2))
        studentized Breusch-Pagan test

data:  modelOLS
BP = 41.014, df = 5, p-value = 9.321e-08
```

Kolmogorov-Smirnov test. The Kolmogorov-Smirnov test is included in base R via the `ks.test` function. To test whether the residuals follow a certain distribution, supply a function giving the CDF of that distribution.

```
> e.standard <- residuals(modelOLS) / sd(residuals(modelOLS))
> ks.test(e.standard, pnorm)
        One-sample Kolmogorov-Smirnov test

data:  e.standard
D = 0.189, p-value < 2.2e-16
alternative hypothesis: two-sided
```

Shapiro-Wilk test. The Shapiro-Wilk test for normality is included in base R via the `shapiro.test` function.

```
> shapiro.test(residuals(modelOLS))
```

```
        Shapiro-Wilk normality test

data:  residuals(modelOLS)
W = 0.7135, p-value < 2.2e-16
> shapiro.test(rnorm(10, sd = 2))
        Shapiro-Wilk normality test

data:  rnorm(10, sd = 2)
W = 0.9419, p-value = 0.5746
```

Jarque-Bera test. The Jarque-Bera test for normality is included via the `jarque.bera.test` function in the `tseries` package.

```
> library(tseries)
> jarque.bera.test(residuals(modelOLS))
        Jarque Bera Test

data:  residuals(modelOLS)
X-squared = 19039, df = 2, p-value < 2.2e-16
```

Durbin-Watson test. The Durbin-Watson test for autocorrelation is included via the `dwtest` function in the `lmtest` package.

```
> library(lmtest)
> dwtest(modelOLS)
        Durbin-Watson test

data:  modelOLS
DW = 1.7189, p-value = 4.277e-06
alternative hypothesis: true autocorrelation is greater than 0
```

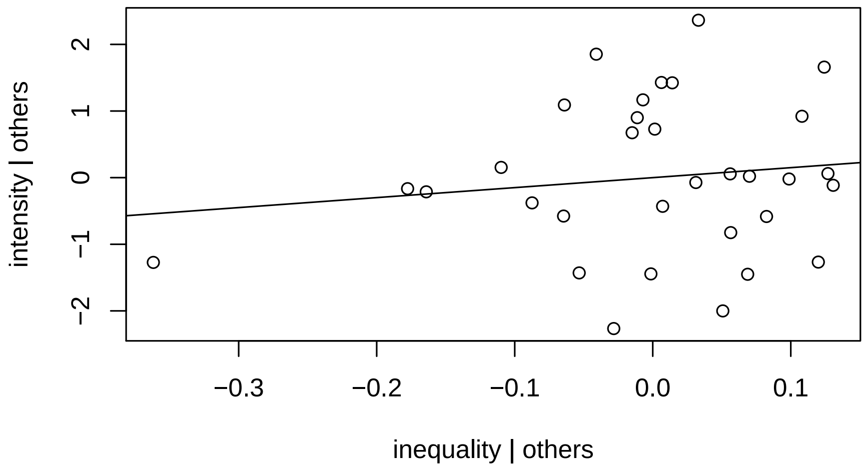
6.4 Diagnostic Plots

The `car` package also provides functionality for two kinds of residual-based plots that can help illustrate regression results and diagnose violations of the standard assumptions.

6.4.1 Partial Regression Plots

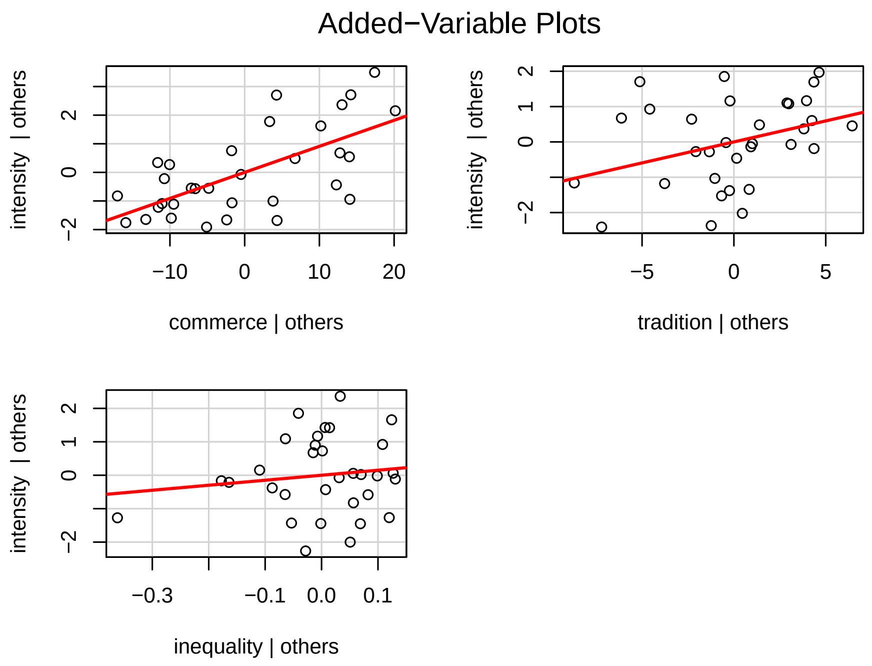
Also known as *added-variable plots*, these illustrate the conditional relationship between Y and X_j after "integrating out" the other regressors, X_{-j}. To form a partial regression plot, we plot the residuals of a regression of Y on X_{-j} against those of a regression of X_j on X_{-j}. Once again, we'll first do this by hand for `inequality` in the Chirot data.

```
> ey <- residuals(lm(intensity ~ commerce + tradition,
+                    data = Chirot))
> ex <- residuals(lm(inequality ~ commerce + tradition,
+                    data = Chirot))
> plot(ex, ey, xlab = "inequality | others",
+      ylab = "intensity | others")
> abline(lm(ey ~ ex))
```



We can get an easier, better-looking, more comprehensive version by running `avPlots` (from the `car` package) on our original fitted model.

```
> avPlots(modelChirot)
```

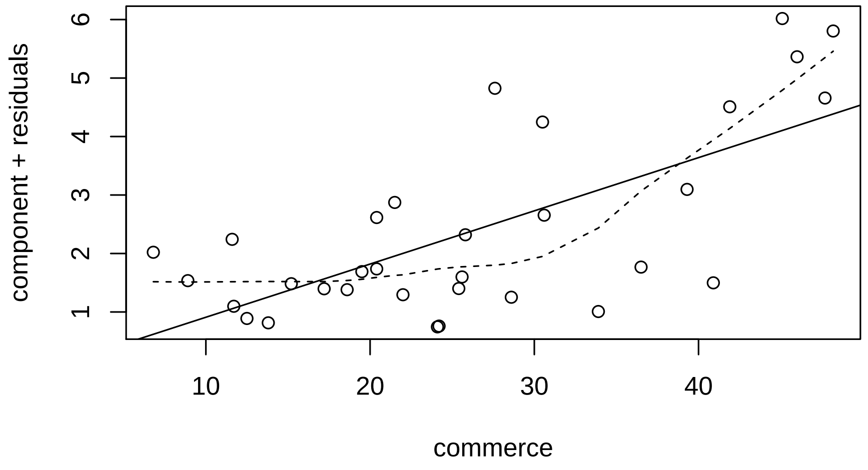


The main use of partial regression plots is to determine whether there are any outliers that have a lot of influence on the regression coefficient. You may be tempted to use naive plots of Y against each X_j for this task, but partial regression plots are better. Unlike in the naive plots, the slope of the bivariate regression line in a partial regression plot is equal to the corresponding coefficient in the full regression.

6.4.2 Component-Residual Plots

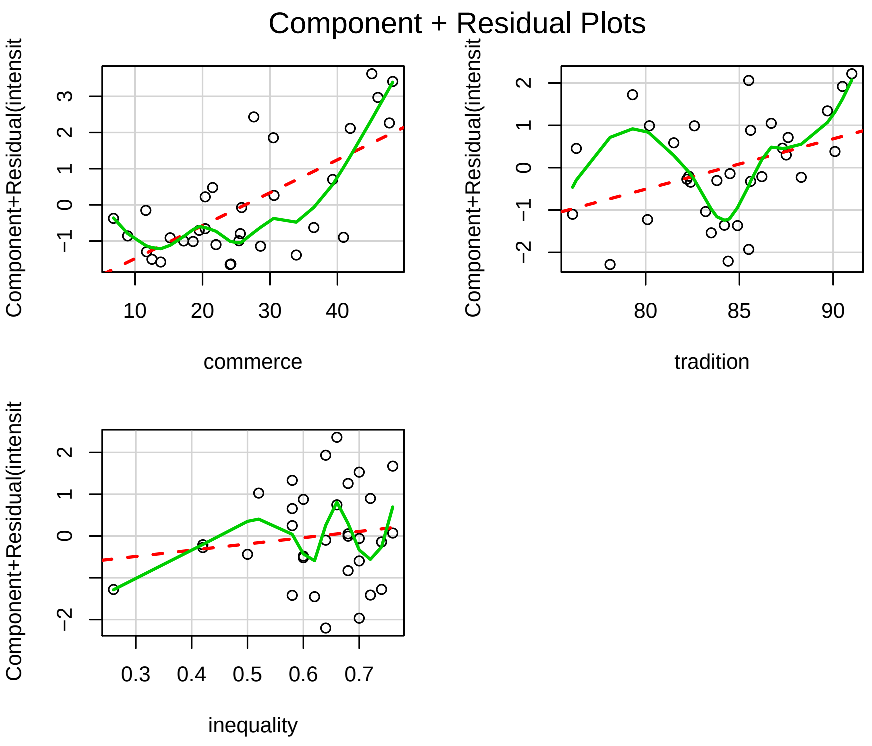
Another common problem in regression is that the partial relationship between Y and X_j is not actually linear, and that X_j may need to be transformed in order for the regression to fit well. The standard diagnostic tool in this case is a component-residual plot, where $X_j'\beta_j + e$ is plotted against X_j. We'll try it first by hand.

```
> b1 <- coef(modelChirot)[2]
> x <- Chirot$commerce
> cr <- b1 * x + residuals(modelChirot)
> plot(x, cr, xlab = "commerce", ylab = "component + residuals")
> abline(lm(cr ~ x))
> lines(lowess(cr ~ x), lty = 2)
```



As was the case in partial residual plots, the coefficient of the regression of the component+residual on X_j is the same as its coefficient in the full regression. Notice also that we've used `lowess` to superimpose a smoothed estimate of the bivariate relationship in order to detect potential nonlinearity. We can make these plots even more easily with the `crPlots` function from the `car` package.

```
> crPlots(modelChirot)
```


Component + Residual Plots
Component+Residual(intensit
commerce
tradition
inequality
10
20
30
40
80
85
90
0.3
0.4
0.5
0.6
0.7
−2
−1
0
1
2
3

Chapter 7

Generalized Least Squares

In this chapter, we'll cover weighted least squares (WLS) and generalized least squares (GLS) estimators. These are actually pretty easy to implement in R once you know your way around the `lm` command, so I'm going to use this topic as an excuse to introduce you to some more intermediate simulation techniques in R.

7.1 Weighted Least Squares

Consider the heteroscedastic regression model

$$\mathbf{y} = \mathbf{X}\beta + \mathbf{u},$$

where $E[\mathbf{u}\mathbf{u}'] \equiv \Sigma$ with $\Sigma_{ij} = 0$ for all i, j but potentially $\Sigma_{ii} \neq \Sigma_{jj}$. As you've already seen in PSC 405, it is best to estimate this model by weighting each observation i by $1/\Sigma_{ii}$. The OLS estimate of β is unbiased but inefficient, and the estimated standard errors from OLS will be wrong.

Let's examine these properties by simulating our own data. Suppose we have

$$y_i = 1 + 2x_i + \epsilon_i,$$

with $N = 5$, $x_i \sim N(0, 1)$ and

$$\Sigma = \begin{bmatrix} 0.25 & 0 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 4 \end{bmatrix}.$$

By now you all should be able to simulate a single dataset according to this model and run an OLS regression on it.

```
> sigma <- c(0.25, 0.5, 1, 2, 4)
> x <- rnorm(5)
> e <- rnorm(5, sd = sqrt(sigma))
> y <- 1 + 2*x + e
> fitOLS <- lm(y ~ x)
> fitOLS
Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
      0.702        1.688
```

To run weighted least squares instead, just use the `weights` argument.

```
> fitWLS <- lm(y ~ x, weights = 1 / sigma)
> fitWLS
Call:
lm(formula = y ~ x, weights = 1/sigma)

Coefficients:
(Intercept)            x
       1.12         2.51
```

What if you didn't believe my claim that WLS is more efficient than OLS? Well, that would be silly, since it's a standard result in any econometrics textbook and you probably proved it in PSC 405. But if you really wanted to reassure yourself, you could simulate the sampling distributions of $\hat{\beta}_{OLS}$ and $\hat{\beta}_{WLS}$. To be more specific, you could simulate the dataset many times, run each estimator in each iteration, and save the results. The easiest way to do this in R is with the `replicate` command.

Before getting into that relatively involved simulation, let's illustrate `replicate` with a couple of simple examples.

```
> replicate(10, runif(1))
 [1] 0.290290 0.302115 0.044684 0.391355 0.172544 0.131722
 [7] 0.329778 0.076276 0.195160 0.758266
```

This performs the command `runif(1)` ten times and stores the results in a vector. So it's basically a less computationally efficient version of `runif(10)` — but useful for illustration.

```
> replicate(5, rnorm(2))
         [,1]     [,2]     [,3]   [,4]    [,5]
[1,]  1.02339 0.219380 -0.91542 1.0267 0.54225
[2,] -0.62516 0.038781  0.30201 1.4150 0.30424
```

This performs the command `rnorm(2)` five times and stores the results in a 2×5 matrix. Again, this could be accomplished more easily by running `rnorm(10)` and then reshaping the results into a matrix. But what if you wanted to perform a simulation with intermediate steps? For example, suppose you wanted to repeat the following 10 times and save the results

```
> a <- rexp(10)
> b <- runif(10, -a, a)
> z <- rnorm(10, sd = sqrt(a + b^2))
> max(abs(z))
[1] 4.2062
```

You could copy and paste that block of code 10 times in your script, but that would be unwieldy and error-prone, and it would probably result in your PSC 505 TA giving you an F. To do this with `replicate`, you can use your trusty friend the curly brackets:

```
> replicate(10, {
+     a <- rexp(10)
+     b <- runif(10, -a, a)
+     z <- rnorm(10, sd = sqrt(a + b^2))
+     max(abs(z))
+ })
 [1] 1.2583 1.6177 1.5375 1.4022 2.2173 2.8567 1.3088 3.0639
 [9] 2.5132 1.9013
```

With a large enough sample, you could even get pretty good estimates of the mean and variance of this random variable.[1]

By now you probably see where I'm going with this. Let's go back to our regression example. Suppose you wanted to generate 100 samples of data and store the coefficients from OLS and GLS on each.

[1]Though this is much less fun than using moment-generating functions, am I right?

```
> ourSim <- replicate(100, {
+     x <- rnorm(5)
+     e <- rnorm(5, sd = sqrt(sigma))
+     y <- 1 + 2*x + e
+     fitOLS <- lm(y ~ x)
+     fitWLS <- lm(y ~ x, weights = 1/sigma)
+     c(coef(fitOLS), coef(fitWLS))
+ })
```

This gives us a 4×100 matrix whose first two rows are OLS coefficients and whose last two rows are WLS coefficients. We can use the `rowMeans` command to (roughly) confirm that both estimators are unbiased.

```
> rowMeans(ourSim)
(Intercept)           x (Intercept)           x
    0.93887     1.87953     0.96326     1.91940
```

These are very close to the true values, as we would expect since the estimators are unbiased. Unfortunately `R` doesn't have a `rowVars` command, so we'll have to dig a bit more into the weeds to compute the variances.[2] You want to take the `var` function and apply it to each row of the `ourSim` matrix. To do that, use the apply function:

```
> apply(ourSim, 1, var)
(Intercept)           x (Intercept)           x
    0.42261     0.53798     0.28185     0.47124
```

If you'd wanted to take the variance of each column instead, you would have run `apply(ourSim, 2, var)` — the second argument tells `apply` which dimension of the matrix to travel along. Getting back to the point, notice that the variances of the WLS estimates are much lower than those of the OLS estimates, as you'd expect.

7.2 Generalized Least Squares

Suppose now that the off-diagonal elements of Σ may be non-zero. In the vanishingly unlikely event that you know the complete form of Σ, you may

[2]Yes, you could run `var(ourSim[1, ])`, `var(ourSim[2, ])`, etc, all separately, or if you're quite clever you could even do `diag(var(t(ourSim)))`, but my real point here is to familiarize you with `apply`.

estimate β via the MASS package's function lm.gls. First let's construct such a weight matrix and simulate some data. In particular we'll use an AR(1) autocorrelation structure with $\phi = 0.5$.

```
> phi <- 0.5
> N <- 10
> Sigma <- diag(N)
> Sigma <- phi^abs(row(Sigma)-col(Sigma))
> Sigma
           [,1]      [,2]      [,3]     [,4]    [,5]
 [1,] 1.0000000 0.5000000 0.2500000 0.125000 0.06250
 [2,] 0.5000000 1.0000000 0.5000000 0.250000 0.12500
 [3,] 0.2500000 0.5000000 1.0000000 0.500000 0.25000
 [4,] 0.1250000 0.2500000 0.5000000 1.000000 0.50000
 [5,] 0.0625000 0.1250000 0.2500000 0.500000 1.00000
 [6,] 0.0312500 0.0625000 0.1250000 0.250000 0.50000
 [7,] 0.0156250 0.0312500 0.0625000 0.125000 0.25000
 [8,] 0.0078125 0.0156250 0.0312500 0.062500 0.12500
 [9,] 0.0039062 0.0078125 0.0156250 0.031250 0.06250
[10,] 0.0019531 0.0039062 0.0078125 0.015625 0.03125
         [,6]     [,7]      [,8]      [,9]     [,10]
 [1,] 0.03125 0.015625 0.0078125 0.0039062 0.0019531
 [2,] 0.06250 0.031250 0.0156250 0.0078125 0.0039062
 [3,] 0.12500 0.062500 0.0312500 0.0156250 0.0078125
 [4,] 0.25000 0.125000 0.0625000 0.0312500 0.0156250
 [5,] 0.50000 0.250000 0.1250000 0.0625000 0.0312500
 [6,] 1.00000 0.500000 0.2500000 0.1250000 0.0625000
 [7,] 0.50000 1.000000 0.5000000 0.2500000 0.1250000
 [8,] 0.25000 0.500000 1.0000000 0.5000000 0.2500000
 [9,] 0.12500 0.250000 0.5000000 1.0000000 0.5000000
[10,] 0.06250 0.125000 0.2500000 0.5000000 1.0000000
```

We'll simulate our data according to the same regression equation as before. Note that we need to use mvrnorm from the MASS package to draw errors from a multivariate normal distribution.

```
> library(MASS)
> x <- rnorm(N)
> e <- mvrnorm(1, mu = rep(0, N), Sigma = Sigma)
> y <- 1 + 2*x + drop(e)
> fitOLS <- lm(y ~ x)
> fitOLS
Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
       1.53         1.77
> fitGLS <- lm.gls(y ~ x, W = solve(Sigma))
> fitGLS$coefficients
(Intercept)           x
     1.5064      1.6880
```

This is all well and good, but it's not a situation you're ever really going to be in. A more realistic scenario would be to assume that the general structure of the weights matrix is known (e.g., that it is an AR(1) autocorrelated process) but that its free parameters (e.g., ϕ) are unknown and to be estimated. For this we want the `gls` function from the `nlme` package, which despite its name actually performs *feasible* GLS estimation.

```
> library(nlme)
> fitFGLS <- gls(y ~ x, correlation = corAR1())
> fitFGLS
Generalized least squares fit by REML
  Model: y ~ x
  Data: NULL
  Log-restricted-likelihood: -10.542

Coefficients:
(Intercept)           x
     1.5030      1.8722

Correlation Structure: AR(1)
 Formula: ~1
 Parameter estimate(s):
     Phi
-0.39752
Degrees of freedom: 10 total; 8 residual
Residual standard error: 0.70905
```

Note that the FGLS estimate of ϕ is consistent but not unbiased. I leave it as an exercise to you to convince yourself of this using `replicate`. The one hint I'll give you — since this took me a little while to figure out — is that you can extract $\hat{\phi}$ from a fitted GLS object using

```
> coef(fitFGLS$modelStruct, unconstrained = FALSE)
corStruct.Phi
     -0.39752
```