Advanced Econometric Methods EMET3011/8014

Lecture 2

John Stachurski

Semester 1, 2011

Announcements

Missed first lecture?

- See `www.johnstachurski.net/emet`
- Weekly download of course notes

First computer labs today

- Comp Lab 1: 12–1 PM, COP G21 (Bld. 24)
- Comp Lab 2: 1–2 PM, COP G21 (Bld. 24)

Students should attend Comp Lab 1 or 2, but **not both**

COP G21 has 20 seats/computers

Today's Lecture

Introduction to R

- Variables, vectors
- Graphics
- Data
- Tips on working with the interpreter

Starting R produces welcome message and then prompt

```
R version 2.9.2 (2009-08-24)

[Some other messages]

>
```

The prompt > is where we type commands

```
> print("foobar")  # This is a comment
[1] "foobar"
```

R as a calculator: To add 12 and 23,

```
> 12 + 23  # Whitespace
[1] 35
```

To multiply we type

```
> 12 * 23  # Whitespace
```

Exponents, division

```
> 12^23  # No whitespace
> 12 / 23  # Whitespace
```

Parentheses:

```
> 12 * 2 + 10
[1] 34
> 12 * (2 + 10)
[1] 144
```

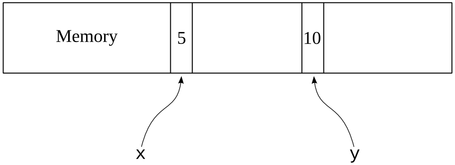
Variables

Variable: a name bound to (associated with) a value/object

Bind name `x` to value 5, and `y` to 10:

```
> x <- 5  # Equivalent to x = 5, note whitespace
> y <- 10
```

Values stored in memory, names associated to that memory block



When you use symbol `x`, R retrieves values and uses it in place of `x`:

```
> x <- 5
> y <- 10
> x
[1] 5
> x + 3
[1] 8
> x + y
[1] 15
> 2 * x
[1] 10
> x   # Value of x has not changed
[1] 5
> y   # Value of y has not changed
[1] 10
```

More examples:

```
> x <- 5
> x <- 2 * x
> x
[1] 10
```

What happens here?

1. expression on r.h.s. is evaluated
2. name `x` (re)bound to this value

Hence, value of `x` is now 10

Can use `ls()` to see what variables exist in memory

```
> ls()
[1] "x" "y"
> rm(x)
> ls()
[1] "y"
```

More complex names can help us remember what our variables stand for:

```
> number_of_observations <- 200
```

or

```
> number.of.observations <- 200
```

Some rules:

- `x1` a legitimate name, but `1x` is not (can't start with number)
- R is case sensitive (`a` and `A` are distinct)

Vectors

Now let's create a **vector**

```
> a <- c(2, 5, 7.3, 0, -1)
> a
[1]  2.0  5.0  7.3  0.0 -1.0
```

Here `c()` concatenates the numbers 2, 5, 7.3, 0, -1 into a vector

Resulting vector is of length 5:

```
> length(a)
[1] 5
```

In R, scalar variables are vectors of length 1

The c() function can also concatenate vectors:

```
> a <- c(2, 4)
> b <- c(6, 8)
> a_and_b <- c(a, b)
> a_and_b
[1] 2 4 6 8
```

We can also create vectors of regular sequences:

```
> b <- 1:5   # Same as b <- c(1, 2, 3, 4, 5)
> b
[1] 1 2 3 4 5
> b <- 5:1
> b
[1] 5 4 3 2 1
```

For more flexibility, use the function `seq()`:

```
> b <- seq(-1, 1, length=5)
> b
[1] -1.0 -0.5  0.0  0.5  1.0
```

Try `help(seq)` or `?seq` to learn more

To generate a constant vector, try `rep`:

```
> z <- rep(0, 5)
> z
[1] 0 0 0 0 0
```

Vectors of random variables:

```
> x <- rnorm(3)       # 3 draws from N(0,1)
> y <- rlnorm(30)     # 30 lognormal rvs
> z <- runif(300)     # 300 uniform rvs on [0,1]
```

We'll learn more about these later

Referencing the k-th element:

```
> a <- c(2, 5, 7.3, 0, -1)
> a[3]
[1] 7.3
> a[3] <- 100
> a
[1]   2   5 100   0  -1
```

We cannot reference element of undeclared vector

```
> ls()
[1] "a"
> b[1] <- 42
Error in b[1] <- 42 : object 'b' not found
> b <- rep(0, 5)
> b[1] <- 42
> b
[1] 42  0  0  0  0
```

Extracting several elements at once:

```
> a <- c(2, 5, 7.3, 0, -1)
> a[1:4]
[1] 2.0 5.0 7.3 0.0
```

Negative index returns all but the indicated value:

```
> a[-1]
[1]  5.0  7.3  0.0 -1.0
```

Vector Operations

```
> x <- 1:5
```

Can obtain sum of all elements via

```
> sum(x)
[1] 15
```

Minimal and maximal element, average and median:

```
> min(x)
[1] 1
> max(x)
[1] 5
> mean(x)
[1] 3
> median(x)
[1] 3
```

Vectorized Functions

Many functions act element by element on vectors

- Common to most math scripting languages

For example, `log()` returns log of each element:

```
> x <- 1:4
> log(x)
[1] 0.0000000 0.6931472 1.0986123 1.3862944
```

We say that `log()` is a **vectorized** function

Some more examples

```
> exp(x)
[1]   2.718282   7.389056  20.085537  54.598150
> sin(x)
[1]  0.8414710  0.9092974  0.1411200 -0.7568025
> abs(cos(x))
[1] 0.5403023 0.4161468 0.9899925 0.6536436
> round(sqrt(x), 1)
[1] 1.0 1.4 1.7 2.0
```

Arithmetic Operations on Vectors

Performed elementwise. For example:

```
> a <- 1:4
> b <- 5:8
> a
[1] 1 2 3 4
> b
[1] 5 6 7 8
> a + b    # Add a to b, element by element
[1]  6  8 10 12
> a * b    # Multiply, element by element
[1]  5 12 21 32
```

Suppose we want to multiply each element of vector `a` by 2

One way:

```
> a <- 1:4
> a * rep(2, 4)    # 4 because a has 4 elements
[1] 2 4 6 8
```

More generally:

```
> a * rep(2, length(a))
[1] 2 4 6 8
```

A nicer way:

```
> a * 2  # Standard method of scalar multiplication
[1] 2 4 6 8
```

Same principle works for addition, division and so on

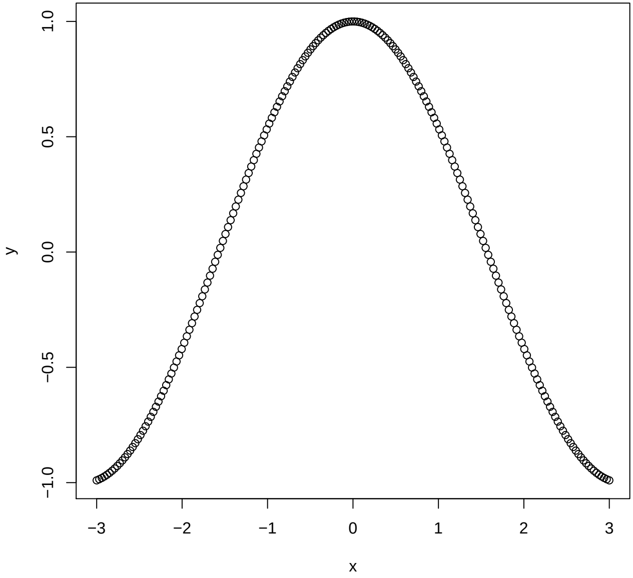
Graphics

Many different ways to produce figures

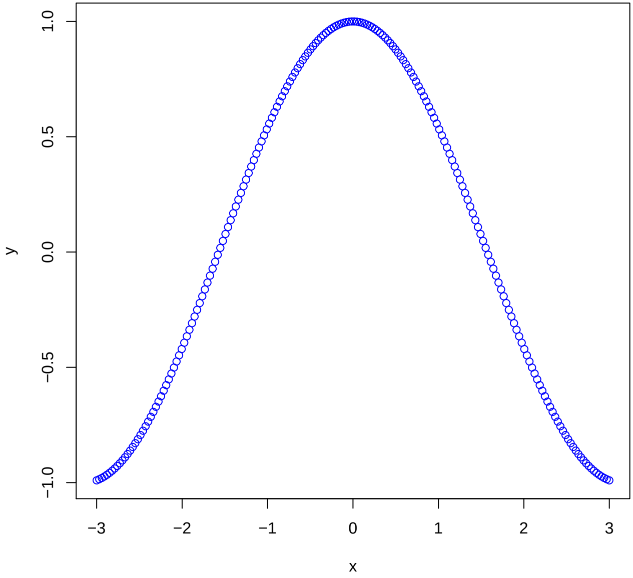
A common one is the `plot()` command

```
> x <- seq(-3, 3, length=200)
> y <- cos(x)
> plot(x, y)
```

Produces a **scatter plot** of x and y, as in next figure


y
x
1.0
0.5
0.0
−0.5
−1.0
−3
−2
−1
0
1
2
3

For blue we use the command `plot(x, y, col="blue")`


1.0
0.5
0.0
−0.5
−1.0
y
−3
−2
−1
0
1
2
3
x

To produce a blue line, try `plot(x, y, col="blue", type="l")`

(That's `l` as in London, not the number `1`)

To produce a red line: `plot(x, y, col="red", type="l")`

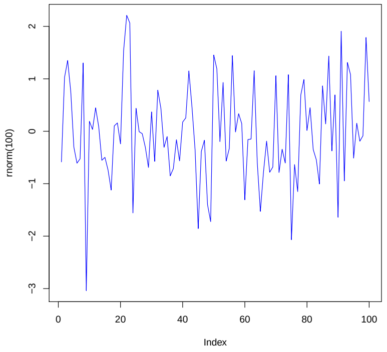
To produce green/step: `plot(x, y, col="green", type="s")`

Try also:

```
plot(x, y, col="green", type="l", xlab="foo",
     ylab="bar", main="foobar")
```

Note: `plot()` with one vector argument plots a time series

```
> plot(rnorm(100), type="l", col="blue")
```



High-Level Graphical Commands

Graphical commands either “high-level” or “low-level”

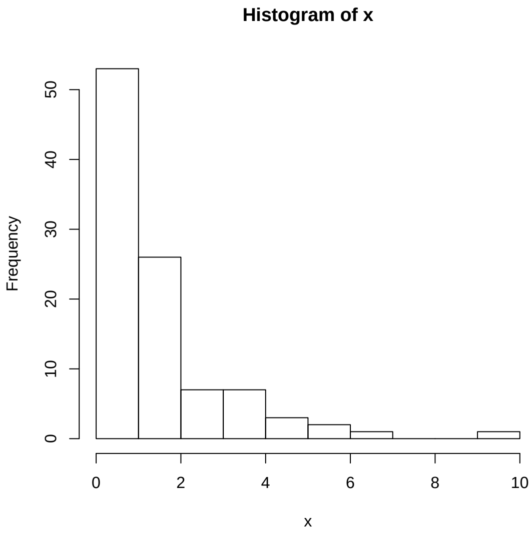
High-level commands implement initial plot/figure

Low-level commands add points, lines and text

`plot()` is an example of a high-level command

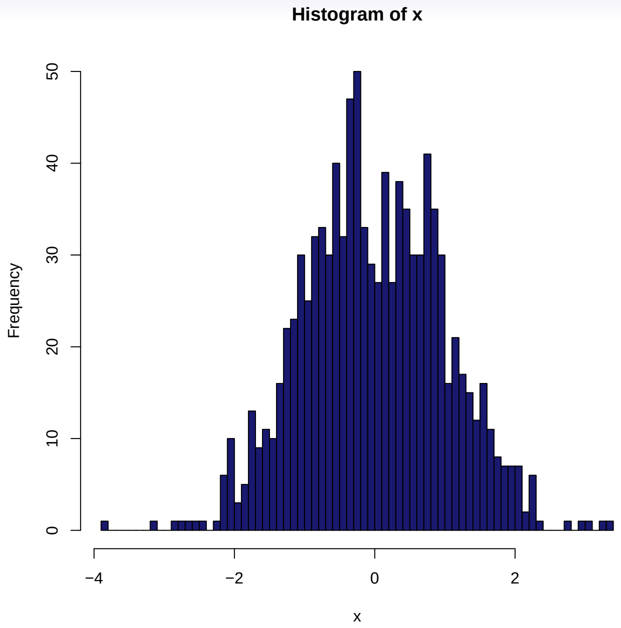
Another is `hist()`:

```
> x <- rlnorm(100)   # lognormal density
> hist(x)
```


Histogram of x
Frequency
0
10
20
30
40
50
0
2
4
6
8
10
x

Let's make something fancier

```
> x <- rnorm(1000)
> hist(x, breaks=100, col="midnightblue")
```

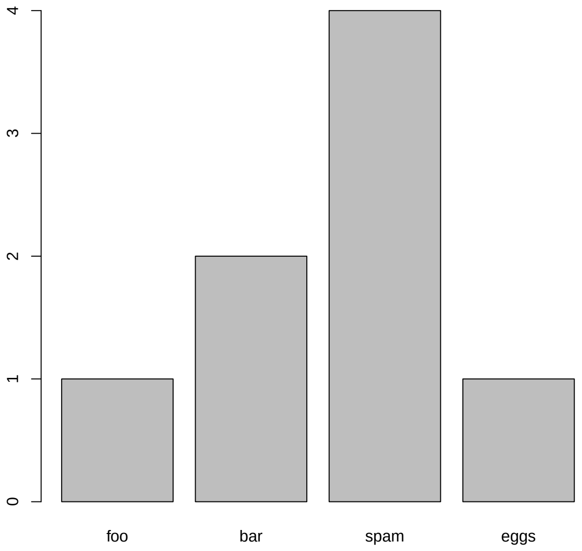

Histogram of x
Frequency
0
10
20
30
40
50
−4
−2
0
2
x

Other high-level plotting functions for presenting data:

- `barplot()` produces bar plots,
- `boxplot()` produces box plots,
- `pie()` produces pie charts
- `contour()` produces contour maps of 3D data
- `persp()` produces 3D graphs

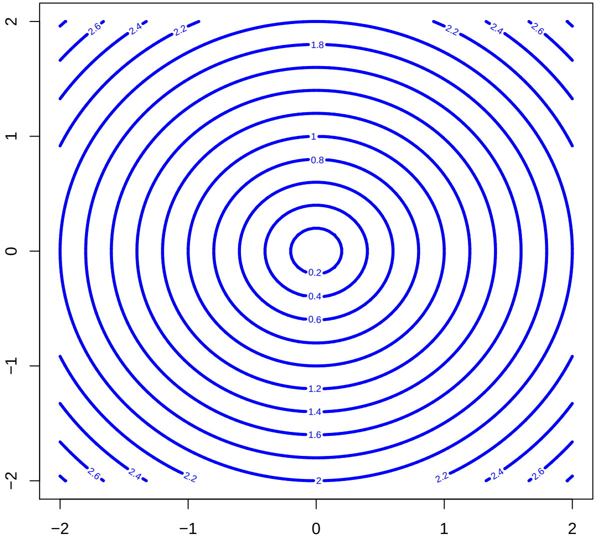
Here's an example of a bar plot:

```
> x <- c(1, 2, 4, 1)
> names(x) <- c("foo", "bar", "spam", "eggs")
> x
 foo  bar spam eggs
   1    2    4    1
> barplot(x)
```

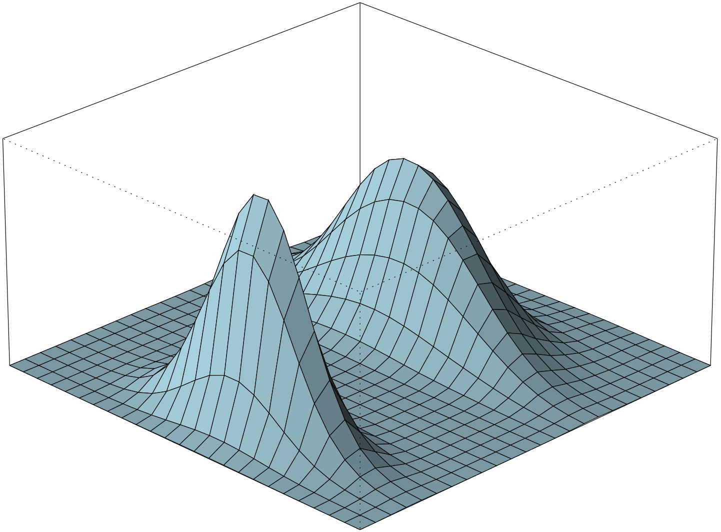

0
1
2
3
4
foo
bar
spam
eggs

Here's an example of a contour plot:

```
> x <- seq(-2, 2, length=200)
> y <- x
> z <- outer(x, y, function (x,y) sqrt(x^2 + y^2))
> contour(x, y, z, col = "blue", lwd = 3)
```


0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
2.2
2.4
2.6
−2
−1
0
1
2
−2
−1
0
1
2

Finally, a 3D graph produced using `persp()` is shown in next figure



Low-Level Graphical Commands

Low-level commands used to add components to existing plots

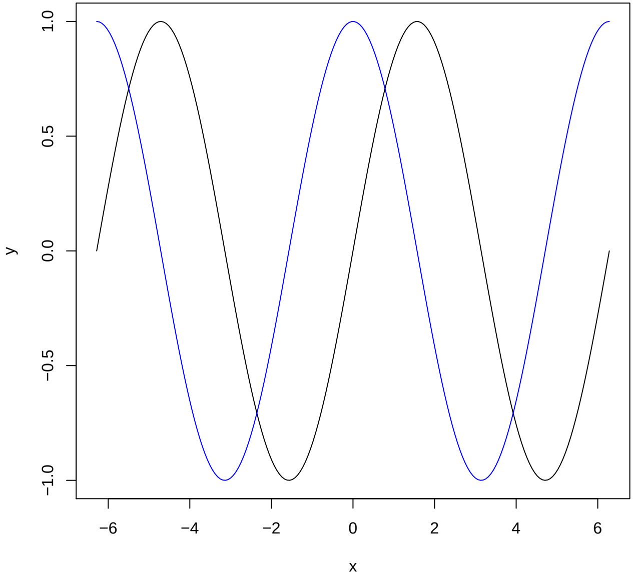
- `points()`, which adds points,
- `lines()`, which adds lines,
- `text()`, which adds text,
- `abline()`, which adds straight lines,
- `polygon()`, which adds polygons
- `arrows()`, which adds arrows, and
- `legend()`, which adds a legend

General procedure for producing figures

- One and only one high level graphics command
- Followed by low level commands to add details

Example combining high and low level commands:

```
> x <- seq(-2 * pi, 2 * pi, length=200)
> y <- sin(x)
> z <- cos(x)
> plot(x, y, type="l")
> lines(x, z, col="blue")
```


1.0
0.5
0.0
−0.5
−1.0
y
−6
−4
−2
0
2
4
6
x

Getting Hardcopy

R provides range of graphics drivers to produce hardcopy

For example, to produce histogram as a PDF, try

```
> x <- rlnorm(100)
> pdf("foo.pdf")    # Target file called foo.pdf
> hist(x)           # Issue graphics commands
> dev.off()         # Write graphics output to file
```

Data Types

R needs to work with numbers, text, Boolean values, etc.

These variables have different "types"

A common type is **numeric**, for **floating point numbers**

```
> b <- c(3, 4)
> class(b)
[1] "numeric"
> mode(b)
[1] "numeric"
```

Output of `mode()` and `class()` may differ

- `mode()` refers to primitive data type
- `class()` is more specific—see below

Common data type 2: **strings**, for pieces of text

In R, strings have mode “character”

```
> x <- "foobar"  # Bind name x to string "foobar"
> mode(x)
[1] "character"
```

We can concatenate strings using the `paste()` function:

```
> paste("foo", "bar")
[1] "foo bar"
> paste("foo", "bar", sep="")  # Sep by empty string
[1] "foobar"
```

A more useful example of `paste()`:

```
> paste("Today is ", date())
[1] "Today is  Mon Feb 28 11:22:11 2011"
```

Strings written between quotes: `x <- "foobar"` not `x <- foobar`

Otherwise R interprets as name of a variable

Example: we looked at the command

```
> hist(x, breaks=100, col="midnightblue")
```

Here `"midnightblue"` is a string, passed to `hist()`

Why not

```
> hist(x, breaks=100, col=midnightblue)  # Wrong!
```

Because R would think that `midnightblue` is a variable

Common data type 3: **Booleans**, for `TRUE` and `FALSE`

```
> x <- TRUE
> mode(x)
[1] "logical"
```

Like other primitive types, can be stored in vectors

```
> x <- c(TRUE, TRUE, FALSE)
> mode(x)
[1] "logical"
```

Can use abbreviations `T` and `F`:

```
> x <- c(T, T, F)
> x
[1]  TRUE  TRUE FALSE
```

To test and change type, R provides `is.` and `as.` functions

Example:

```
> x <- c("100", "200")
> is.numeric(x)
[1] FALSE
> sum(x)
Error in sum(x) : invalid type
> x <- as.numeric(x)  # Convert x to numeric
> is.numeric(x)
[1] TRUE
> sum(x)
[1] 300
```

In any one vector, all data must be of only one type

Example:

```
> x <- c(1.1, "foo")
> mode(x)
[1] "character"
> x
[1] "1.1" "foo"
```

Here 1.1 has been converted to a character string

(Ensures all elements have the same type)

To store different data types as single object, can use **lists**

```
> employee1 <- list(surname="Smith", salary=50000)
```

Elements of this list have names `surname` and `salary`

Elements can be accessed via names as follows:

```
> employee1$surname
[1] "Smith"
> employee1$salary
[1] 50000
```

Function `names()` returns the names:

```
> names(employee1)
[1] "surname" "salary"
```

Lots of R functions return lists

Working with Data

Manipulation of data is an important skill

We need to

- read in data,
- store it,
- select subsets,
- make changes, and so on

R's data manipulation facilities revolve around **data frames**

Data Frames

Special kind of lists used to store and manipulate related data

Stores “columns” of related data

Typically, each column corresponds to observations on one variable

Different columns can hold different data types

Light-hearted example:

Alphaville noted correlation between speaking fees paid to NEC director Lawrence Summers and share prices during GFC

Firms: Goldman Sachs, Lehman Bros, Citigroup and JP Morgan

Speaking fees: $140,000, $70,000, $50,000 and $70,000

Share price falls in year to April 2009: 35%, 100%, 89% and 34%

Let's record this in a data frame

First we put the data in vectors

```
> fee <- c(140000, 70000, 50000, 70000)
> price <- c(-35, -100, -89, -34)
```

And then build a data frame:

```
> summers <- data.frame(fee, price)
```

As discussed above, a data frame is a list:

```
> mode(summers)
[1] "list"
```

In fact, a special kind of list called a data frame:

```
> class(summers)
[1] "data.frame"
```

Let's see what it looks like:

```
> summers
     fee price
1 140000   -35
2  70000  -100
3  50000   -89
4  70000   -34
```

R has

- numbered the rows for us
- used the variable names for columns

Can produce more descriptive column names as follows:

```
> summers <- data.frame(Speak.Fee=fee, Price.Change=
   price)
> summers
        Speak.Fee     Price.Change
1       140000         -35
2        70000        -100
3        50000         -89
4        70000         -34
```

Since a data frame is a list,

```
> summers$Speak.Fee
[1] 140000  70000  50000  70000
> summers$Price.Change
[1]  -35 -100  -89  -34
```

Note that the column names by themselves won't work:

```
> Speak.Fee
Error: object "Speak.Fee" not found
```

(Deliberate data encapsulation)

If you want direct access to column names try `attach(summers)`

Can also access data using “matrix style” index notation:

First row, second column

```
> summers[1, 2]
[1] -35
```

All of second row

```
> summers[2,]
    Speak.Fee Price.Change
2       70000         -100
```

All of second column

```
> summers[,2]
[1]  -35 -100  -89  -34
```

Let's replace row numbers with names of the firms:

```
> row.names(summers)
[1] "1" "2" "3" "4"
> firm <- c("Goldman", "Lehman", "Citi", "JP Morgan")
> row.names(summers) <- firm
> row.names(summers)
[1] "Goldman"   "Lehman"    "Citi"      "JP Morgan"
> summers
            Speak.Fee       Price.Change
Goldman        140000                -35
Lehman          70000               -100
Citi            50000                -89
JP Morgan       70000                -34
```

Many R functions can interact directly with data frames

Example: function `summary()` gives summary of the data:

```
> summary(summers)
    Speak.Fee      Price.Change
 Min.   : 50000   Min.   :-100.00
 1st Qu.: 65000   1st Qu.: -91.75
 Median : 70000   Median : -62.00
 Mean   : 82500   Mean   : -64.50
 3rd Qu.: 87500   3rd Qu.: -34.75
 Max.   :140000   Max.   : -34.00
```

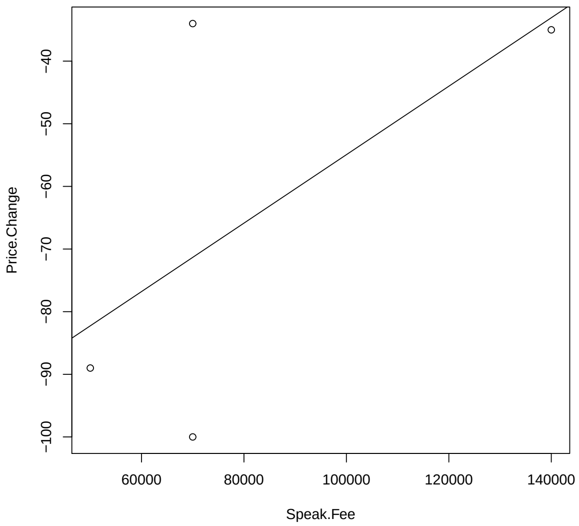
The `plot()` function can be applied to data frames:

```
> plot(summers)
```

Next, we run a linear regression and add line of best fit to the plot

```
> reg <- lm(Price.Change ~ Speak.Fee, data=summers)
> abline(reg)
```

(We'll learn more about running regressions just below)



Price.Change
−40
−50
−60
−70
−80
−90
−100
60000
80000
100000
120000
140000
Speak.Fee

Linear Regression

Let's generate some data for an example:

```
> N <- 25
> x <- seq(0, 1, length=N)
> y <- 5 + 10 * x + rnorm(N)
```

We can regress `y` on `x` as follows:

```
> results <- lm(y ~ x)
```

The function `lm` returns a list containing regression output

```
> mode(results)
[1] "list"
> class(results)
[1] "lm"
```

The list `results` object returned by `lm()` includes

- estimated coefficients,
- residuals,
- etc.

Example:

```
> results$coefficients
(Intercept)           x
   4.491593   11.456543
```

There are also "extractor" functions to get at this info. Try `coef(results)` or `fitted.values(results)`

Also try `summary(results)` to get an overview

In code `lm(y ~ x)`, the expression `y ~ x` is a **formula**

- Special syntax used for specifying statistical models in R

To regress `y` on `x` and `z` use

```
> lm(y ~ x + z)  # Not addition---a formula
```

Note intercept included by default. Can make explicit by

```
> lm(y ~ 1 + x + z)  # Equivalent to last formula
```

To remove the intercept use

```
> lm(y ~ 0 + x + z)  # Or lm(y ~ x + z - 1)
```

Working with the Interpreter

R is mainly command line interface (CLI) rather than GUI

CLI very powerful when you get used to it

Allows natural transition to programming

Let's look at some tips for working with the R CLI

How to get help?

Documentation can be accessed as follows:

```
> help(plot)
```

Now presented with manual page for the `plot` function

Pressing "q" exits man page (different for Windows or Mac?)

Shortcut:

```
> ?plot
```

More extensive search:

```
> help.search("plotting")
```

Tab expansion

Try typing following, and then press the tab key:

```
> data.f
```

It's expanded to `data.frame` , the only possible completion

Try typing following, and then press the tab key twice:

```
> dat
```

Presents list of possible expansions

Once we define our own variables, the same technique works

Good if you have a variable called

```
interwar.consumption.in.southern.alabama
```

Command History

Another useful feature is **command history**

Accessed via the up and down arrow keys

Up arrow key scrolls through previously typed commands

When you find one you want, press enter to re-run

Or edit the command and then run (use left/right arrow keys)

This will save you lots of typing

Finally

Computer labs:

- Replication of today's lecture slides
- Some small exercises