

Topics for today

- Input / Output
- Using data frames
- Mathematics with vectors and matrices
- Summary statistics
- Basic graphics

Input:

Data files

For rectangular data files (n rows, c columns) you usually want to use `read.table()`.

```
read.table(file, header = F, sep = "",
           row.names = NULL,
           col.names = paste("V", 1:fields, sep = ""),
           as.is = F, na.strings = "NA", skip = 0)
```

The arguments you are normally going to want to deal with are `header`, and `sep`.

`header`

logical flag: if TRUE, then the first line of the file is used as the variable names of the resulting data frame. The default is FALSE, unless there is one less field in the first line of the file than in the second line.

`sep`

the field separator (single character), often `"\t"` for tab. If omitted, any amount of white space (blanks or tabs) can separate fields. To read fixed format files, make `sep` a numeric vector giving the initial columns of the fields.

If the data file doesn't have as nice structure as required for `read.table`, you probably want to use `scan` instead.

```
scan(file="", what=numeric(), n=<<see below>>,
      sep=<<see below>>, multi.line=F, flush=F,
      append=F, skip=0, widths=NULL,
      strip.white=<<see below>>)
```

The important arguments, besides `file`, are `what`, `sep` and `flush`

`what`

a vector of mode numeric, character, or complex, or a list of vectors of these modes. Objects of mode logical are not allowed. If `what` is a numeric, character, or complex vector, `scan` will interpret all fields on the file as data of the same mode as that object. So, `what=character()` or `what=""` causes `scan` to read data as character fields. If `what` is missing, `scan` will interpret all fields as numeric.

If `what` is a list, then each record is considered to have `length(what)` fields and the mode of each field is the mode of the corresponding component in `what`. When `widths` is given as a vector of length greater than one, `what` must be a list of the same length as `widths`.

`sep`

separator (single character), often `"\t"` for tab or `"\n"` for newline. If omitted, any amount of white space (blanks, tabs, and possibly newlines) can separate fields. If `widths` is specified, then `sep` tells what separator to insert into fixed-format records.

`flush`

if TRUE, `scan` will flush to the end of the line after reading the last of the fields requested. This allows putting comments after the last field that are not read by `scan`, but also prevents putting multiple sets of items on one line.

While data files in text format are extremely common, you may need to deal with data coming from other packages, such as SAS, Excel, SPSS, etc. These can be read in with `sas.get` for SAS and `importData` for many packages, including Excel and SPSS. Note that the version of `importData` in version 5.1 will often limit the versions of the data files in the other programs. For example, Excel files must be from version 4 or earlier. It appears that the same or similar restrictions hold for version 6 as well.

Exporting S data

Most of the functions mentioned earlier have counterparts for exporting your S data to other programs. Since text files are usually the easiest to work with, `write.table` is the one you will use the most.

```
write.table(data, file = "", sep = ",", append = F,  
            quote.strings = F, dimnames.write = T, na = NA,  
            end.of.row = "\n")
```

Its arguments are similar to `read.table`. One change I suggest is to give a `sep` argument and not use the default of `" , "`. Instead I would use a space `" "`, or a tab `"\t"`, as it will be easier to read into a program such as Excel.

The counterpart to `scan` is `write`.

```
write(x, file="data", ncolumns=<<see below>>,  
      append=F)
```

Usually I think that `write.table` is the way to go, Also you need to be careful with the default for `ncolumns`.

`ncolumns`

number of data items to put on each line of file.
Default is 5 per line for numeric data, 1 per line for character data.

importData also has its counterpart for exporting data. Not surprisingly its exportData. Also this is the only way to export SAS files as I can't find the counterpart to sas.get.

Running scripts

Like with unix, it is possible to write scripts of S commands, instead of having to type the commands in one by one at the prompt. As part of my example last week, I read in a dataset, generated some plots and created some new variables.

```
cars<-read.table("/home/irwin/Scourse/93cars.dat",
  header=T,row.names=NULL)
```

```
postscript("citympg.ps",horiz=T)
plot(cars.df$weight, cars.df$citympg, xlab="Weight",
  ylab="CityMPG", main="City MPG versus Weight")
abline(lsfite(cars.df$weight, cars.df$citympg))
dev.off()
```

```
cars.df$cityfuel <- 100/cars.df$citympg
```

```
postscript("cityfuel.ps",horiz=T)
plot(cars.df$weight,cars.df$cityfuel,xlab="Weight",
  ylab="CityFuel",main="City Fuel versus Weight")
abline(lsfite(cars.df$weight,cars.df$cityfuel))
dev.off()
```

When I did it, I just typed in the command. However, I might have wanted to redo these commands another time. With the `source` function, its easy to run scripts.

Assume the above commands are in a file `testscript.s`. Then the command `source('testscript.s')` will run the commands in the above file and return you to the S prompt.

```
source(file, local=F, echo=<<see below>>, n = -1,  
       immediate = NULL)
```

The important argument for `source` is `echo`. It determines the amount of output generated by the `source` command

`echo`

if `TRUE`, each expression will be printed, along with a prompt, before it is evaluated. The default is `TRUE` if `options(echo=T)` has been set and `length(recordConnection())==0`.

For example,

```
> source('testscript.s')
```

Generated postscript file "citympg.ps".

Generated postscript file "cityfuel.ps".

```
> source('testscript.s',echo=T)
```

```
> cars.df<-read.table("/home/irwin/Scourse/93cars.dat",  
  header = T, row.names = NULL)
```

```
> postscript("citympg.ps", horiz = T)
```

```
> plot(cars.df$weight, cars.df$citympg, xlab =  
  "Weight", ylab = "CityMPG",  
  main = "City MPG versus Weight")
```

```
> abline(lsfite(cars.df$weight, cars.df$citympg))
```

```
> dev.off()
```

Generated postscript file "citympg.ps".

```
> cars.df$cityfuel <- 100/cars.df$citympg
```

```
> postscript("cityfuel.ps", horiz = T)
```

```
> plot(cars.df$weight, cars.df$cityfuel,  
  xlab = "Weight", ylab = "CityFuel",  
  main = "City Fuel versus Weight")
```

```
> abline(lsfite(cars.df$weight, cars.df$cityfuel))
```

```
> dev.off()
```

Generated postscript file "cityfuel.ps".

Another use of script files is for running long jobs in the background. To do this, give the command at the unix prompt

```
SPlus BATCH inputfile outputfile
```

This command will run the script contained in `inputfile` and write the output to `outputfile`, returning you to the unix prompt while the script is running. The contents for `outputfile` will look exactly like what you would get on the screen if you typed the commands in yourself.

Saving output:

Besides running a job in the background, which will automatically save all of your output to a file, you can also do it from the S prompt with the `sink` command.

```
> summary(cars.df$citympg)
  Min. 1st Qu. Median  Mean 3rd Qu.  Max.
15.00 18.00   21.00 22.37 25.00   46.00

> sink('sinktest.out')

> summary(cars.df$citympg)

> sink()

> summary(cars.df$citympg)
  Min. 1st Qu. Median  Mean 3rd Qu.  Max.
15.00 18.00   21.00 22.37 25.00   46.00

>
```

Using data frames

If you use the most common approach for reading in data, `read.table`, you end up with a data frame. As we saw last time one way to access the entries of a data frame was to do something like `cars.df$citympg`. However there are different ways of accessing the components of a data frame.

Treat it like a matrix

```
> simple.df<-data.frame(norm=rnorm(5,2,0.5),
  chi=rchisq(5,3))
> objects(simple.df)
[1] "chi"  "norm"
> simple.df
      norm      chi
1 2.742964 9.488857
2 1.945042 2.252652
3 2.120086 1.705713
4 2.438854 1.738452
5 2.111714 5.695775
```

So instead of accessing the chi-squared column with `simple.df$chi`, you can also use `simple.df[,2]`.

```
> simple.df$chi
[1] 9.488857 2.252652 1.705713 1.738452 5.695775
> simple.df[,2]
[1] 9.488857 2.252652 1.705713 1.738452 5.695775
> simple.df[, 'chi']
[1] 9.488857 2.252652 1.705713 1.738452 5.695775
```

Attaching the data frame

This allows you to access the variables in the data frame without having to specify which data frame contains the variables. Note that if you attach two data frames at the same time, and they both have a variable with the same name, you will only be able to access one of them (the one higher in the search list) by just giving the variable name.

```

> search()
[1] ".Data" "splus" "stat" "data" "trellis" "main"
> chi
Problem: Object "chi" not found
> simple.df$chi
[1] 9.488857 2.252652 1.705713 1.738452 5.695775
> attach(simple.df)
> search()
[1] ".Data" "simple.df" "splus" "stat" "data"
[6] "trellis" "main"
> chi
      1      2      3      4      5
9.488857 2.252652 1.705713 1.738452 5.695775
> simple.df$chi
      1      2      3      4      5
9.488857 2.252652 1.705713 1.738452 5.695775
> chi<-chi*2
> ls()
[1] ".Last.value" ".Random.seed" "cars.df" "chi"
[5] "iris3" "last.dump" "simple.df"
[8] "u" "v" "w"

```

```
> chi
      1      2      3      4      5
18.97771 4.505304 3.411426 3.476903 11.39155
> simple.df$chi
      1      2      3      4      5
9.488857 2.252652 1.705713 1.738452 5.695775
```

If I remove `simple.df` from the search path with `detach`

```
> detach(what="simple.df")
NULL
> chi
      1      2      3      4      5
18.97771 4.505304 3.411426 3.476903 11.39155
> simple.df$chi
[1] 9.488857 2.252652 1.705713 1.738452 5.695775
```

The changes made to `chi` don't get updated in the data frame.

Lets reset things

```
> ls()

[1] ".Last.value" ".Random.seed" "cars.df"
[4] "iris3"       "last.dump"     "simple.df"
[7] "u"           "v"             "w"

> attach(simple.df,1)

> search()

[1] "simple.df" ".Data" "splus" "stat" "data"
[6] "trellis" "main"

> chi<-chi*2

> detach(1,save="simple2.df")
```

This should save the updated version of chi to simple.df. Unfortunately something isn't working right with Splus. Instead for now use

```
> simple.df$chi <- simple.df$chi *2.

> attach(simple.df)

> summary(norm)

  Min. 1st Qu. Median  Mean 3rd Qu.  Max.
1.945 2.112   2.120  2.272 2.439   2.743

> summary(simple.df$norm)

  Min. 1st Qu. Median  Mean 3rd Qu.  Max.
1.945 2.112   2.120  2.272 2.439   2.743

> summary(chi)

  Min. 1st Qu. Median  Mean 3rd Qu.  Max.
1.706 1.738   2.253  4.176 5.696   9.489
```

Even when a data frame is attached, you can still access it the other way I've been talking about.

```
> summary(simple.df)
```

norm	chi
Min.:1.945	Min.:1.706
1st Qu.:2.112	1st Qu.:1.738
Median:2.120	Median:2.253
Mean:2.272	Mean:4.176
3rd Qu.:2.439	3rd Qu.:5.696
Max.:2.743	Max.:9.489

Creating Matrices

There are a number of ways of creating matrices in S. First you can treat a data frame as a matrix as we've seen (though I just had a problem with it). You might want to do something like

```
simple.mat <- as.matrix(simple.df)
```

There is the `matrix` function

```
matrix(data=NA, nrow=<<see below>>,
        ncol=<<see below>>, byrow=F, dimnames=NULL)
```

Data is a numeric vector, with NAs allowed. You only need to give one of the `nrow` and `ncol` arguments (but giving both is ok). The important argument is probably `byrow`.

`byrow`

logical flag: if TRUE, the data values are assumed to be the first row, then the second row, etc. If FALSE, the values are assumed to be the first column, then the second column, etc. (The latter is how the data are stored internally.)

```

> matrix(1:6,ncol=3)
      [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> matrix(1:6,ncol=3,byrow=T)
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
> matrix(1:6,nrow=2,byrow=T)
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
> matrix(1:6,nrow=2,ncol=6,byrow=T)
      [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    2    3    4    5    6
[2,]    1    2    3    4    5    6

```

You can also combine vectors and matrices to make bigger ones with `cbind` and `rbind`

```
> X <- matrix(1:6,ncol=3)
> X
      [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> Y <- matrix(1:6,ncol=3,byrow=T)
> Y
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
> cbind(X,Y)
      [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    3    5    1    2    3
[2,]    2    4    6    4    5    6
> rbind(X,Y)
      [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
[3,]    1    2    3
[4,]    4    5    6
```

These two commands also work with vectors (or vectors and matrices)

```
> cbind(rep(1,5),1:5)
```

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

```
> rbind(rep(1,5),1:5)
```

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

Matrix and vector math

```
> x<-1:5
> y<-seq(2,10,2) > x
[1] 1 2 3 4 5
> y
[1] 2 4 6 8 10
> x+y
[1] 3 6 9 12 15
> x*y
[1] 2 8 18 32 50
> X+Y
      [,1] [,2] [,3]
[1,]    2    5    8
[2,]    6    9   12
> X*Y
      [,1] [,2] [,3]
[1,]    1    6   15
[2,]    8   20   36
```

Matrix multiplication is done with `%*%`. Also to get the transpose of a matrix, use the `t()` function.

```
> X %*% Y
```

```
Problem in "%*%.default"(X, Y): Number of columns of x  
should be the same as number of rows of y
```

Use `traceback()` to see the call stack

```
> t(X) %*% Y
```

```
      [,1] [,2] [,3]  
[1,]    9   12   15  
[2,]   19   26   33  
[3,]   29   40   51
```

Also the standard math functions work component wise

```
> log(x)
```

```
[1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379
```

```
> log(X)
```

```
      [,1]      [,2]      [,3]  
[1,] 0.0000000 1.098612 1.609438  
[2,] 0.6931472 1.386294 1.791759
```

Summary stats

Useful functions to get summary statistics are

mean, var, stdev, median, min, max, range, quantile, and summary.

```
> mean(1:100)
```

```
[1] 50.5
```

```
> var(1:100)
```

```
[1] 841.6667
```

```
> stdev(1:100)
```

```
[1] 29.01149
```

```
> median(1:100)
```

```
[1] 50.5
```

```
> quantile(1:100,seq(0,1,0.2))
```

```
0%  20%  40%  60%  80% 100%  
1  20.8  40.6  60.4  80.2  100
```

```
> quantile(1:100)
```

```
0%   25%  50%   75% 100%  
1  25.75  50.5  75.25  100
```

```
> quantile(1:100,c(0.3, 0.6, 0.9))
```

```
30%  60%  90%  
30.7  60.4  90.1
```

```
> summary(1:100)
```

```
Min. 1st Qu. Median Mean 3rd Qu. Max.  
1.00  25.75  50.50  50.50  75.25 100.00
```

Basic graphics

Stem and Leaf, Histograms, Boxplots, Scatterplots

```
> stem(cars.df$engsize)
```

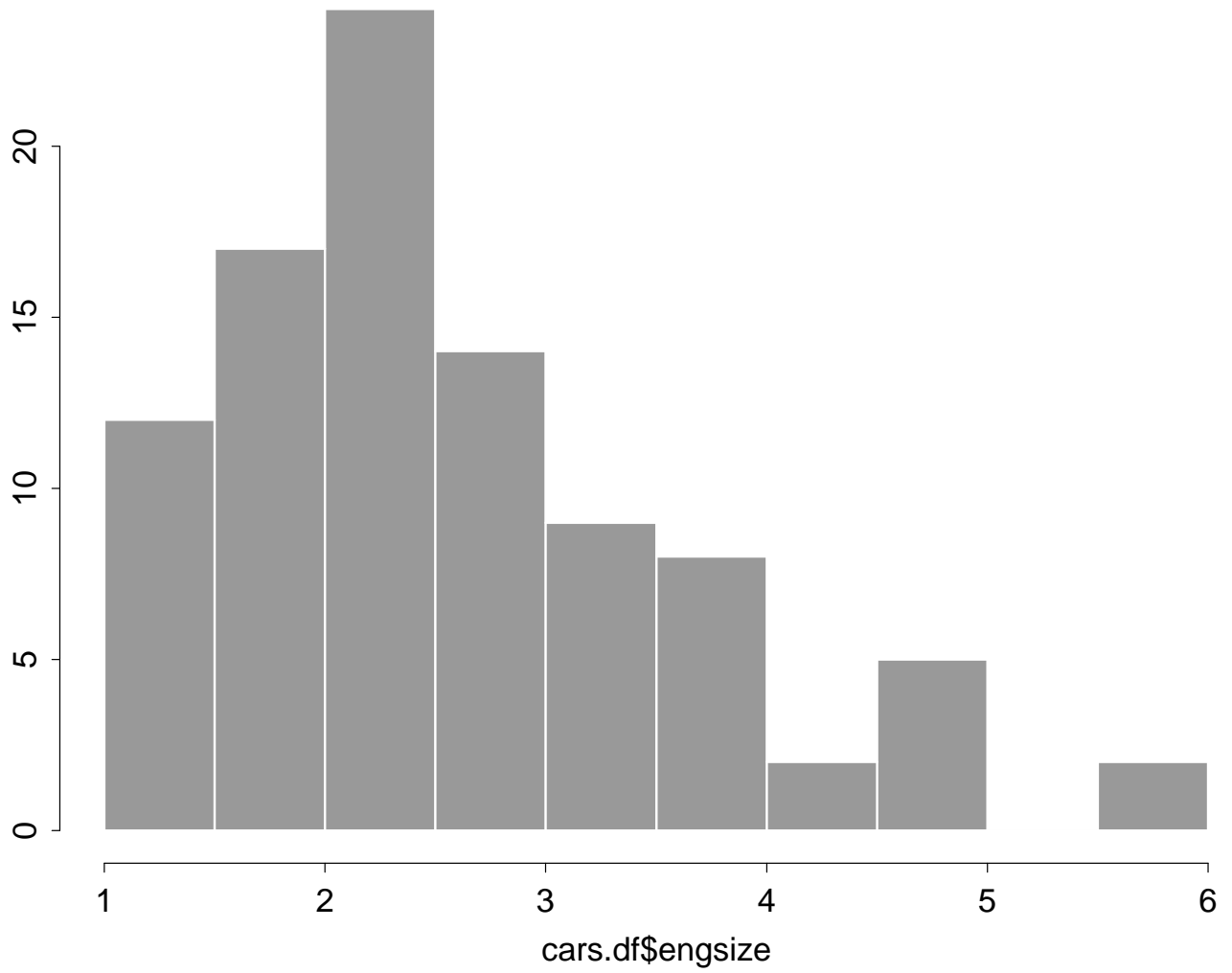
```
N = 93    Median = 2.4
```

```
Quartiles = 1.8, 3.3
```

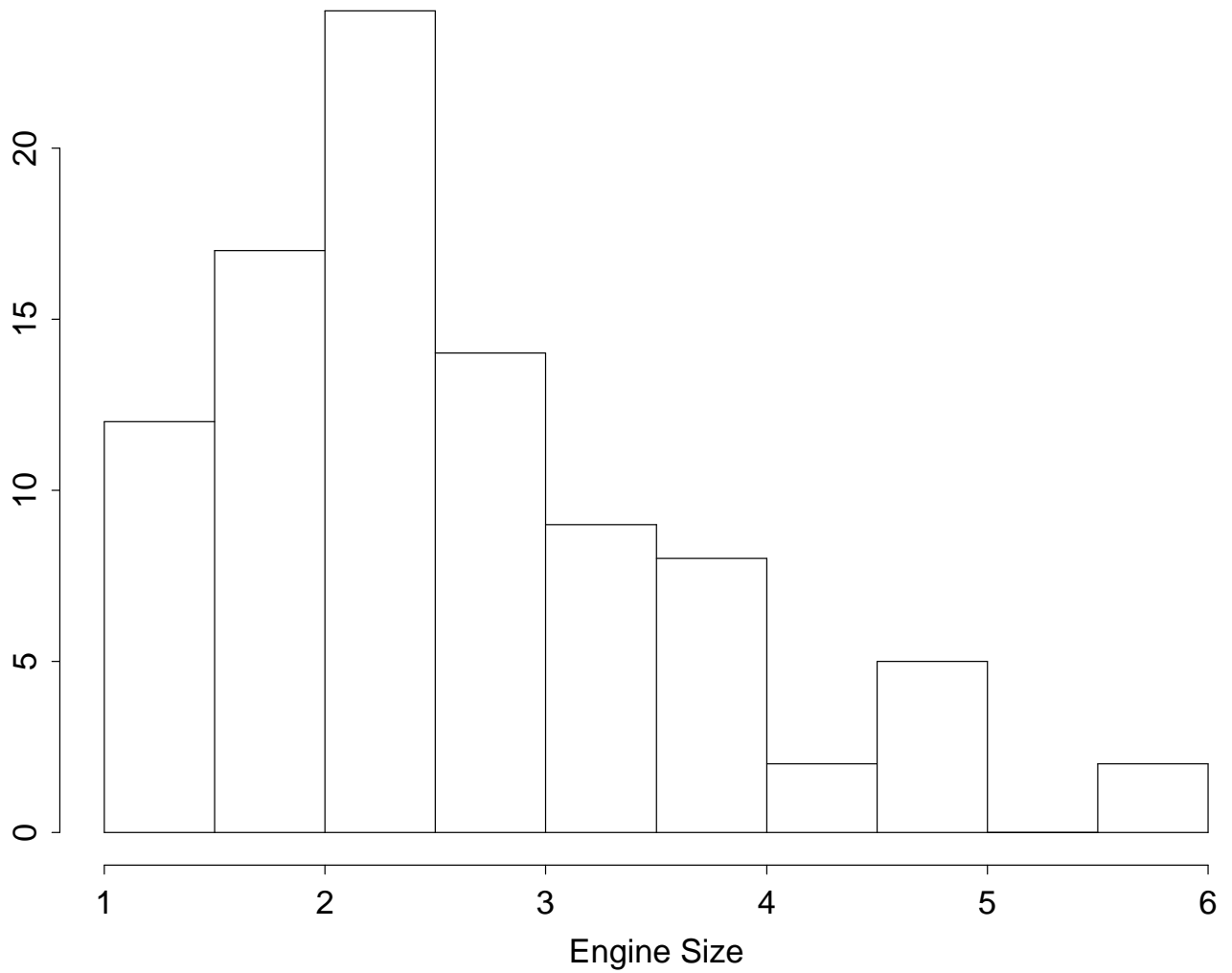
Decimal point is at the colon

```
1 : 02333
1 : 5555556666688888889
2 : 00001222222222333333444
2 : 5555888
3 : 000000000002233444
3 : 55888888888
4 : 3
4 : 56669
5 : 0
5 : 77
```

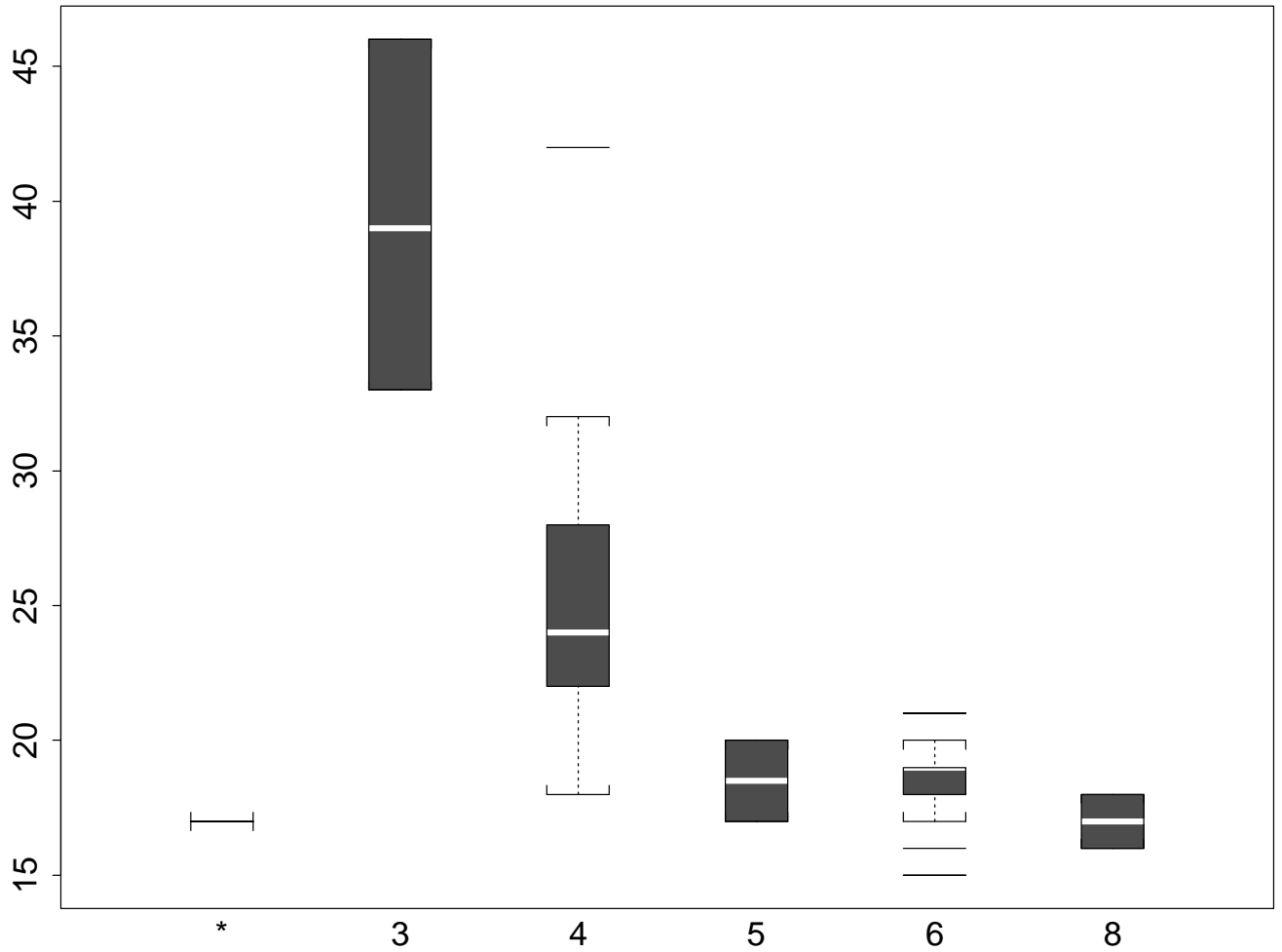
```
> hist(cars.df$engsize)
```



```
> hist(cars.df$engsize, col=0, xlab="Engine Size")
```

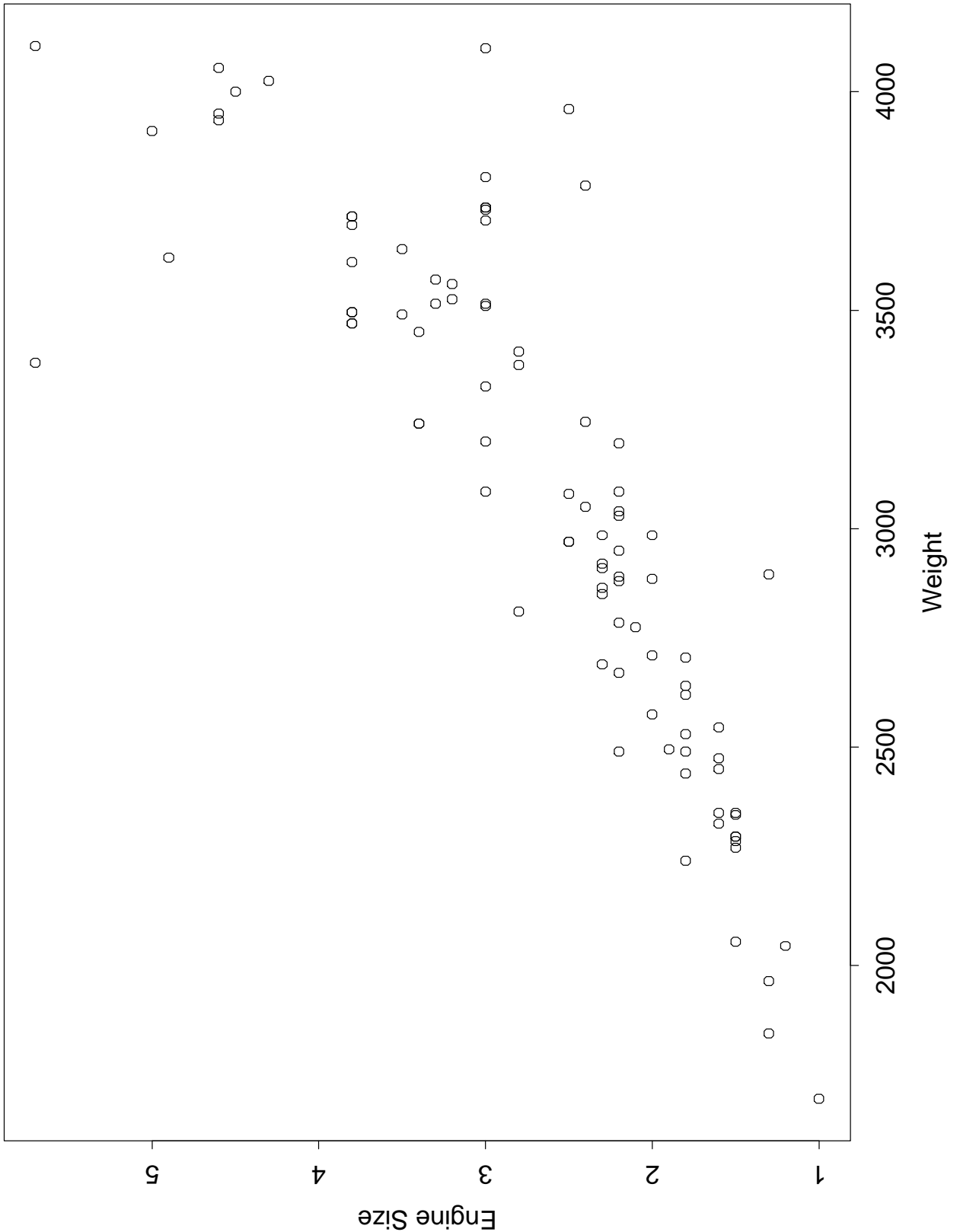


```
> boxplot(split(citympg,cylinders))
```



```
plot(weight,engsize,pch=1,xlab="Weight",ylab="Engine Size",main="93 Cars Dataset")
```

93 Cars Dataset



If you don't give an output device, the output will go to the screen. The usual on screen graphic devices are `motif` (Splus) and `X11/x11` (R). Once you have the plot the way you want it, you can create a postscript file with the `printgraph` command.

```
> printgraph(file='boxplot.ps', width=6, height=5,
             horiz=F)
```

You can also choose the graphics driver and give the plot commands. For example, the boxplot could have been created by

```
> postscript(file='boxplot.ps', width=6, height=5,
             horiz=F)
> boxplot(split(citympg,cylinders))
> dev.off()
```

The `dev.off` command is needed to indicate the figure is finished and to save the file.

Other graphics devices are `pdf.graph` (pdf files), `wmf.graph` (Windows metafile, version 6 or later of Splus). In R there is `pdf`, `xfig` (a unix graphics program), `png` (png bitmat), and `jpeg`.