University of Washington

STATISTICS

STAT 421

Applied Statistics and Experimental Design

Introduction/Review of **R**

Fritz Scholz

Fall Quarter 2008

Purpose

The following slides are intended as review of or as a basic introduction to the statistical analysis platform **R**. They are by no means exhaustive!

The class web page contains links to more extensive introductions and they should be consulted when these slides raise questions.

The internal documentation accompanying **R** via Help on the **R** toolbar is another resource. More on Help later.

Also, it is always useful to experiment with certain command ideas to see and understand/interpret what happens.

Start right away as you reread these slides to get a feel.
We will be using **R** extensively throughout this course.

Statistical Analysis Platform

Freely available from `http://cran.r-project.org/`

See also `http://en.wikipedia.org/wiki/R_(programming_language)`

For Windows version: at CRAN site ⟶ Windows (95 and later) ⟶ base

Download R-2.7.2-win32.exe or latest version to your desktop.

For installation double-click on this executable program `R-2.7.2-win32.exe`
and follow the instructions (use defaults when prompted).

This creates a blue **R** icon on your desktop.

Double-clicking this R icon opens up an R session.

You close the session by typing `q()` or `quit()`.

This prompts you to save workspace image (with all changes in current session)
or not (leave the workspace as it was when starting this session),
or cancel (continue working in R).

Workspaces and Directories

By default the workspace image is saved in
{\tt C:\Program Files\R\R-2.7.2\.RData}

It is a good idea to keep separate workspace images for different projects (HW?), otherwise the clutter will become unmanageable.

Keep each separate workspace in a separate directory.

To save an open workspace in a specific directory, say `R-practice`, click on File on the tool bar in the **R** work session, choose Change dir, browse to that directory `R-practice` and choose OK.

When you quit, `q()`, after that change of directories, the workspace .RData is saved in that new directory.

A new icon **R** with name .RData appears in that directory.

Starting a Session from a Directory

When you have a directory containing an **R** icon with name .RData you can open a session using that workspace by double-clicking on that icon.

To see the objects in that workspace type `ls()` or `objects()`.
When there are no objects the response is character(0).

If you want to start with a clean (empty) workspace you can remove all those objects by typing the command `rm(list=ls())`.

If you want to keep every object except specific ones, say myobject and dataset.x, you would remove them by typing rm(myobject,dataset.x).

Help in **R**

If you know the name of a data or function object you just type ?that.object.name.

For example: ?rivers or ?median

If you don't know such object names you should open the web browser based help facility in **R** by typing help.start()

This web browser interface gives access to all **R** related information.

It has a search engine and provides entry via keywords or by topic such as Basics, Graphics, Mathematics, Programming, Statistics.

The **R** Reference Manual has over 1500 pages documenting available functions, operators and data sets. Resist the temptation to print it, if it was installed

Access it via the **R** tool bar $\longrightarrow$ `Help` $\longrightarrow$ `Manuals` $\longrightarrow$ `R Reference Manual`.

?rivers yields

rivers(datasets) R Documentation

Lengths of Major North American Rivers

Description

This data set gives the lengths (in miles) of 141 major rivers in North America, as compiled by the US Geological Survey.

Usage

rivers

Format

A vector containing 141 observations.

Source

World Almanac and Book of Facts, 1975, page 406.

References

McNeil, D. R. (1977) Interactive Data Analysis. New York: Wiley.

?median yields

```
Median Value
Description
Compute the sample median.

Usage
median(x, na.rm = FALSE)

Arguments
x an object for which a method has been defined, or a numeric vector containing
the values whose median is to be computed. na.rm a logical value indicating
whether NA values should be stripped before the computation proceeds.

Details
This is a generic function for which methods can be written. However, the default
method makes use of sort and mean, both of which are generic, and so the default
method will work for most classes (e.g. "Date") for which a median is a reasonable
concept.

References
Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) The New S Language.
Wadsworth & Brooks/Cole.

See Also
quantile for general quantiles.  ..... and more
```

Naming Conventions in **R**

R is case sensitive. Object names (data or functions) should only contain alpha-numeric characters (A-Z, a-z, 0-9) or a period.

Such names cannot start with a digit.

Object names can start with a period, but they are hidden when you type `ls()`.

This is useful when you want to define hidden or background objects.

You should avoid using object names that are already used by **R**, such as `t, c, q, T, F, ls, pt, mean, var, pi`, etc.. Use descriptive names.

Any object that you create using such system names would mask the built-in **R** object. For example, `pi=3` would create a new object `pi` in your workspace, with value `3` and not `3.141593`.

You get the old `pi` back by removing the masking `pi` via `rm(pi)` from your workspace.

Basic Usage of R

We can use R as an oversized scientific calculator.

```
> 5+6*3
[1] 23
> exp(log(10))
[1] 10
> pi
[1] 3.141593
> sin(pi)
[1] 1.224606e-16     # practically zero
> sin(2)^2+cos(2)^2  # text after # is treated as a comment
[1] 1                # and is not executed
> 1/Inf
[1] 0
> Inf      # Inf stands for infinity
[1] Inf    # and operations with it will yield sensible results
> exp(-Inf)
[1] 0
> exp(Inf)
[1] Inf
```

Vectors and Vectorized Calculations

```
> ls()
character(0)  # the workspace is empty
> x=1:5       # we assign the vector (1,2,3,4,5) to the object name x
> y=c(5,4,3,2,1) # c for concatenate
> x   # typing the object name displays the object content
[1] 1 2 3 4 5
> y
[1] 5 4 3 2 1
> ls()
[1] "x" "y"   # now the workspace contains the objects x and y
> x^2
[1]  1  4  9 16 25   # the square operation is vectorized
> y+x
[1] 6 6 6 6 6        # the summation is elementwise, vectorized
> x^y                # same here
[1]  1 16 27 16  5
```

Functions Creating Vectors

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(-4,4,1) # sequence from -4 to in increments of 1
[1] -4 -3 -2 -1  0  1  2  3  4
> c(1,2,4,6,7) # concatenation of 1,2,4,6,7
[1] 1 2 4 6 7
> rep(1,10) # vector of 10 repeat 1's
 [1] 1 1 1 1 1 1 1 1 1 1
> rep(c(1,2),5) # vector of 5 repeat c(1, 2)
 [1] 1 2 1 2 1 2 1 2 1 2
> rev(1:5) # reverse the vector 1:5
[1] 5 4 3 2 1
> t(1:5) # transpose the column vector to a row vector (1 row matrix)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
```

Functions of Vectors

```
> mean(y)  # mean is a system function, called with parentheses
[1] 3      # typing mean without (..) would display the function body
> median(x)
[1] 3
> sum(x)
[1] 15
> var(x)
[1] 2.5
> sort(y)
[1] 1 2 3 4 5
```

Data Modes

We saw that vectors can be made up of numbers or numeric mode elements.

We can also use character or logic data in vectors

```
> x=c("abc","ABC","xyzXYZ")  # example of a character vector
> x
[1] "abc" "ABC" "xyzXYZ"
>Y=c(TRUE,T,FALSE,F)          # example of a logic vector
>Y
[1]  TRUE  TRUE FALSE FALSE
```

`T` and `TRUE` are equivalent, same with `F` and `FALSE`

Note that we use no quotes on `T, TRUE, F` and `FALSE`

Logical Operators

symbol	function	symbol	function
<	less than	&&	logical AND
>	greater than	||	logical OR
<=	less than or equal to	!	logical NOT
>=	greater than or equal to		
==	equal to		
!=	not equal to		

```
> 1:5<=3
[1]  TRUE  TRUE  TRUE FALSE FALSE
> "abc"<"a"    # logical operations work on character data as well
[1] FALSE
> "abc"<"abd"  # lexicographical ordering
[1] TRUE
> 1<"a"
[1] TRUE
```

Note the mode coercion $\longrightarrow$ next slide

Mode Coercion

Mixed mode elements in expressions are coerced to a

lowest common denominator mode

mode order: logical $\longrightarrow$ numeric $\longrightarrow$ character

```
> c(T,F,0)
[1] 1 0 0
> T+3    # in arithmetic expression T & F
[1] 4    # are interpreted as 1 & 0, respectively
> c(T,3,"abc")
[1] "TRUE" "3"    "abc"
```

When in doubt, experiment!!

Subvectors of Vectors

```
> y
[1] 5 4 3 2 1
> y[c(5,3,1)]  # subvectors can be extracted by giving
[1] 1 3 5      # the index positions as a vector
> y[3:5]
[1] 3 2 1
> y[6]         # an nonexisting index position returns NA
[1] NA
> y[-(1:3)]    # negative index positions are omitted
[1] 2 1        # while the rest are returned
> y>3
[1]  TRUE  TRUE FALSE FALSE FALSE
> y[y>3]   # we can also extract desired index positions
[1] 5 4    # by specifying a logic vector of same length as y
> y[c(T,T,F,F,F)] # this is an equivalent extraction
[1] 5 4 # we get those elements with index TRUE or T
```

Matrices

```
> mat=cbind(x,y) # cbind combines vectors of same length
 > mat           # vertically positioned next to each other
     x y
[1,] 1 5
[2,] 2 4
[3,] 3 3
[4,] 4 2
[5,] 5 1
> mat=cbind(x,y,y^2)
> mat
     x y
[1,] 1 5 25
[2,] 2 4 16
[3,] 3 3  9
[4,] 4 2  4
[5,] 5 1  1   # all columns of a matrix have to have same length
```

dimnames of Matrices

```
> dimnames(mat)
[[1]]
NULL

[[2]]
[1] "x" "y" ""
```

Note that the 5 rows of mat don't have names, columns 1 & 2 have the original vector names but column 3 has an empty string name.

Note the list nature result of dimnames(mat), list of vectors unequal in length. More on lists later.

```
>dimnames(mat)[[2]][3]="y2"
> mat[1,]
 x  y y2
 1  5 25
```

Submatrices of Matrices

```
> mat[1:3,1:2]
     x y
[1,] 1 5
[2,] 2 4
[3,] 3 3

> mat[,-2]
     x y2
[1,] 1 25
[2,] 2 16
[3,] 3  9
[4,] 4  4
[5,] 5  1
```

Character Matrices

```
> letters[1:3]
[1] "a" "b" "c"
> LETTERS[1:3]
[1] "A" "B" "C"
> ABC=cbind(letters[1:3],LETTERS[1:3])
> ABC
     [,1] [,2]
[1,] "a"  "A"
[2,] "b"  "B"
[3,] "c"  "C"
> cbind(1:3,letters[1:3])
     [,1] [,2]
[1,] "1"  "a"
[2,] "2"  "b"
[3,] "3"  "c"
# Numbers were coerced to characters
# Elements of matrices have to be of same mode.
```

Matrices via `rbind`

```
> rmat=rbind(1:3,2:4,5:7)
> rmat
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    2    3    4
[3,]    5    6    7
> mat[1:3,]
     x y y2
[1,] 1 5 25
[2,] 2 4 16
[3,] 3 3  9
> mat[1:3,]+rmat
     x y y2
[1,] 2 7 28
[2,] 4 7 20
[3,] 8 9 16
```

Matrices via `matrix`

```
> matrix(1:12,ncol=3,nrow=4)
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
>  matrix(1:12,ncol=3,nrow=4,byrow=T)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[4,]   10   11   12

# byrow = F is default
```

Custom Functions

R can be extended by writing your own function.

It is my way of building up an analysis or plot construction in stepwise fashion without having to retype all commands.

```
> square
Error: object "square" not found
> fix(square) #here we edit the function object square & save it.
> square
function (x)
{
x^2
}
> square(1:5)
[1]  1  4  9 16 25
```

Note the difference in calling `square` with and without arguments.

The `fix` Editor

The command `fix(pythagoras)` opens up an editor showing

```
function ()
{
}
```

which you can modify, save (as function object `pythagoras`) and then use, e.g.

```
function (x,y)
{sqrt(x^2+y^2)}
> pythagoras(1:2,2)
[1] 2.236068 2.828427
shorter vector is recycled until length of longer vector is matched.
```

but

```
> pythagoras(1:2,1:3)
[1] 1.414214 2.828427 3.162278
Warning message:
longer object length
        is not a multiple of shorter object length in: x^2 + y^2
```

Another Way to Construct Functions

We can define the function object `Pythagoras` directly from the command line.

```
> Pythagoras=function(x,y){
+ sqrt(x^2+y^2)} # the function body is between { and }
> Pythagoras
function(x,y){
sqrt(x^2+y^2)}
> Pythagoras(1:3,2:4)
[1] 2.236068 3.605551 5.000000
```

Note the `+` character replacing the prompt `>` when the command is incomplete.

This prompts you to complete an incomplete expression.

A function expression is not complete until the opening `{` is matched by a closing `}`

Functions and Local Variables

```
> x
[1] 1 2 3 4 5
> square
function (x)
{x=x^2 # inside the function x is local
x}
> square()
Error in square() : argument "x" is missing, with no default
> square(x=x)
[1]  1  4  9 16 25
> x
[1] 1 2 3 4 5
```

the right `x` in `square(x=x)` comes from workspace, the left `x` is local to the function.

Note that the workspace `x` is unchanged.

Using the NotePad or WordPad Editors

Using NotePad or WordPad (or any other straight text editor) you can also edit a file to contain the function body

`function(...){...}`

Save the file as straight text without any formatting (suffix `.txt` not .rtf).

Then import that function into your workspace via the **R** command `fun.1=dget("FileName")`. See documentation for `dget`.

The suffix `.txt` in `FileName.txt` is not needed, could always rename to `FileName`.

This assumes that the function file with name `FileName` is in the directory from which **R** was initiated or in the most recent directory to which you changed from within your **R** session.

You can always find the path of the current working directory by calling `getwd()`

Another Style of Executing Code

Rather than building up functions via function objects and executing them,
some people prefer to accumulate commands in an outside (**R**) text editor
and cut and paste the lines to be executed as a block onto the **R** command line.

Maybe the preference is because of easier error detection?!

It tends to clutter up the command window as code spools by.

Which way to choose is a matter of taste.

The keyboard up and down arrows $\uparrow\downarrow$ let you redisplay (and execute)
previous commands.
This is useful when you explore the functionality of some commands.
You just change some aspects of the redisplayed command and hit Enter.

Saving Objects outside the **R** Workspace

`dput` is the counterpart to `dget`.

Usage: `dput(obj1,"obj1.file.name")` saves the object (data or function) `obj1` to a file named `obj1.file.name` in the current working directory for that **R** session.

You can use any file name you like and usually you would use `dput(obj1,"obj1")`

Unfortunately the above usage strips any comments that you may have added to your code. Don't know why that is the default for `dput`.

To avoid this stripping of comments in the saved file use `dput(obj1,"obj1",control="all")` instead.

Commenting your function code is strongly advised.

You just forget what you were doing in no time!

Getting Data into **R** from Text File

If your **R** data set was saved to a file via `dput` you can get it back into **R** via `dget`.

`dput(data.obj,"data.obj")` and `data.obj=dget("data.obj")`

Most commonly your data can be represented in rectangular array format,

n rows with same number of entries per row.

The entry type should be consistent from row to row. The first row may contain the names of the variables represented by the array columns.

Save this array as text file `dat.txt` (without extraneous formatting, NotePad).

Follow the last data line by `Enter` (carriage return/new line).

Import this data into **R** via: `data.set.name=read.table("dat.txt",header=T)`

The `,header=T` part should only be used if the `dat.txt` has variable names in the first row. **R** will provide default names otherwise.

The new object `data.set.name` will be a data.frame ready for analysis/manipulations.

Getting Data into **R** from Excel

Getting data from an Excel spreadsheet: The block of data should be a rectangular array, consisting of adjacent and equal length columns of numbers, logical or character data. The first row of that array may contain the names of the variables represented by the columns.

1) save the data block from Excel as a single sheet in (DOS) csv format
   to a file with name `dat.csv`, for example.
2) import this data into **R** via: `data.set.name=read.csv("dat.csv",header=T)`
   The `,header=T` part should only be used if the `dat.csv` has variable names in the first row. **R** will provide default names otherwise.

The new object `data.set.name` will be a data.frame ready for analysis/manipulations.

Further Comments on **R** Data Import

There are many other options of getting data into **R**.

They are documented in the R Data Import/Export manual.

It can be accessed from the **R** tool bar via:

$\longrightarrow$ `Help` $\longrightarrow$ Manuals (in ../PDF) $\longrightarrow$ `R Data Import/Export`.

For more details on `read.table` and `read.csv` see the **R** help system.

For the most part you should get by with these.

See also the documentation for `scan`.

Lists

Lists allow more general forms of data structures.

While data.frames and matrices are rectangular arrays, lists are simply a collection of objects, such as vectors, matrices, functions, data.frames, and also lists.

```
> a.list=list(y,square) # creating a list with 2 objects that exist
> a.list                # in the workspace
[[1]]
[1] 5 4 3 2 1 # this object is a vector

[[2]]
function (x)  # this object is a function
{
x^2
}
```

Referencing List Objects

```
> a.list[1]          # gives a sublist of size 1
[[1]]
[1] 5 4 3 2 1

> a.list[[1]]        # gives the actual object in the previous sublist
[1] 5 4 3 2 1

> a.list[[1]][1:3] # gives the first 3 elements of that object
[1] 5 4 3
```

Lists with Named Objects

```
> b.list=list(my.y=y, # this creates a list
+ my.square=square)   # with object names my.y & my.square
> b.list
$my.y  # note the use of $my.y in place of [[1]]
[1] 5 4 3 2 1

$my.square
function (x)
{
x^2
}

> b.list$my.y  # this is equivalent to referencing b.list[[1]],
[1] 5 4 3 2 1  # but by name
> b.list$my.square(3) # equivalent to b.list[[2]](3)
[1] 9
```

Programming Control Tools: `if` or `if-else`

The `if` or `if-else` construct allows conditional execution of commands

```
if(expression){ execute some commands }
```

```
if(expression){ execute some commands } else
{execute some other commands}
```

Here `expression` is a command expression that evaluates to a single `T` or `F`.

Example expression: `min(x) >0`, `if(min(x) >0){log(x)}`

here `x` is a vector and the logic expression returns `T` or `F`.

I sometimes use `if(1==0){....}` to block out the running of the commands indicated by `.....` Useful in debugging.

Programming Control Tools: `for loop`

The `for loop` allows you to repeat a large number of similar calculations

```
for( index.variable in index.array){ execute some commands
     that may change with the index.variable
    }
#Example
sq.vec=NULL # this initializes the vector sq.vec (empty value)
for( i in 1:1000){sq.vec=c(sq.vec,i^2)
                        # concatenates another value i^2
                        # to the previous vector sq.vec, but all
                        # the previous memory locations need to be
                        # shifted into a new sq.vec
    }

# better and more efficient is
sq.vec=NULL
for( i in 1:1000){sq.vec[i]=i^2}
```

Programming Control Tools: `while loop`

The `while loop` allows a repeated number of similar calculations as long as some logical condition is met. That condition status is checked at the start of each loop. When the condition is not met the loop is exited and commands that follow the loop are executed.

```
x=0
while(x^2 < 10000){
x=x+3
}
# at the end of this while loop x has the value
# of the smallest integer divisible by 3 for which
# x^2 >= 10000, i.e., x=102.
```

Graphics in **R**

Some people use **R** just for producing graphics alone.

A graphic is much more effective in conveying the essence of data.

Use functions to build up a graphic until you are satisfied with it.

You can save the graphic in one of several graphics formats

Metafile, ../PDF, Postscript , Jpeg, Png, Bmp,

or save it to the Clipboard to import it to other applications like MS Word, etc.

When you invoke a graphics command like `plot` or `hist` a graphics window opens in addition to the command console. Click on that graphics window, then click File on the menu bar to see about save options.

Mathematical Annotations

You can add mathematical annotations,

Greek letters, mathematical symbols & expressions.

```
text(90,.04,expression(sigma==5), adj=0)
# adds a Greek sigma = 5 left adjusted to plot at (90,.04)
text(90,.04,substitute(sigma==sigx~",~~mu==muy,
  list(sigx=sig,muy=mu)),adj=1)
# adds Greek sigma = sig ,  Greek mu = mu right adjusted to plot
# at (90,.04). This assumes variables sig and mu exist and have values.
# These values are transferred via sigx and muy to the right of ==.
```

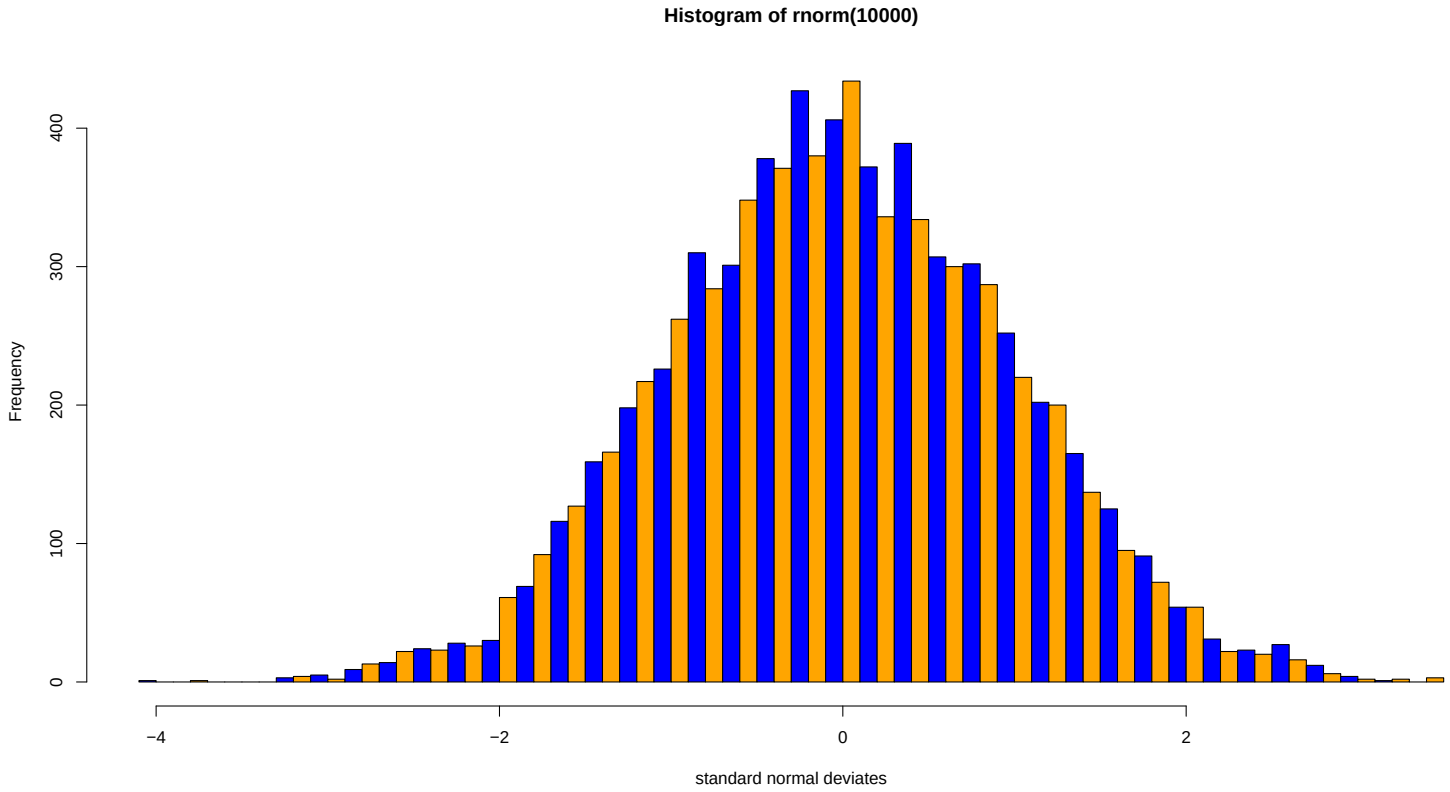
Execute demo(plotmath) on the command line to see which constructs create

which mathematical output on plots.

Also look at plotmath under the help.start() web-based interface.

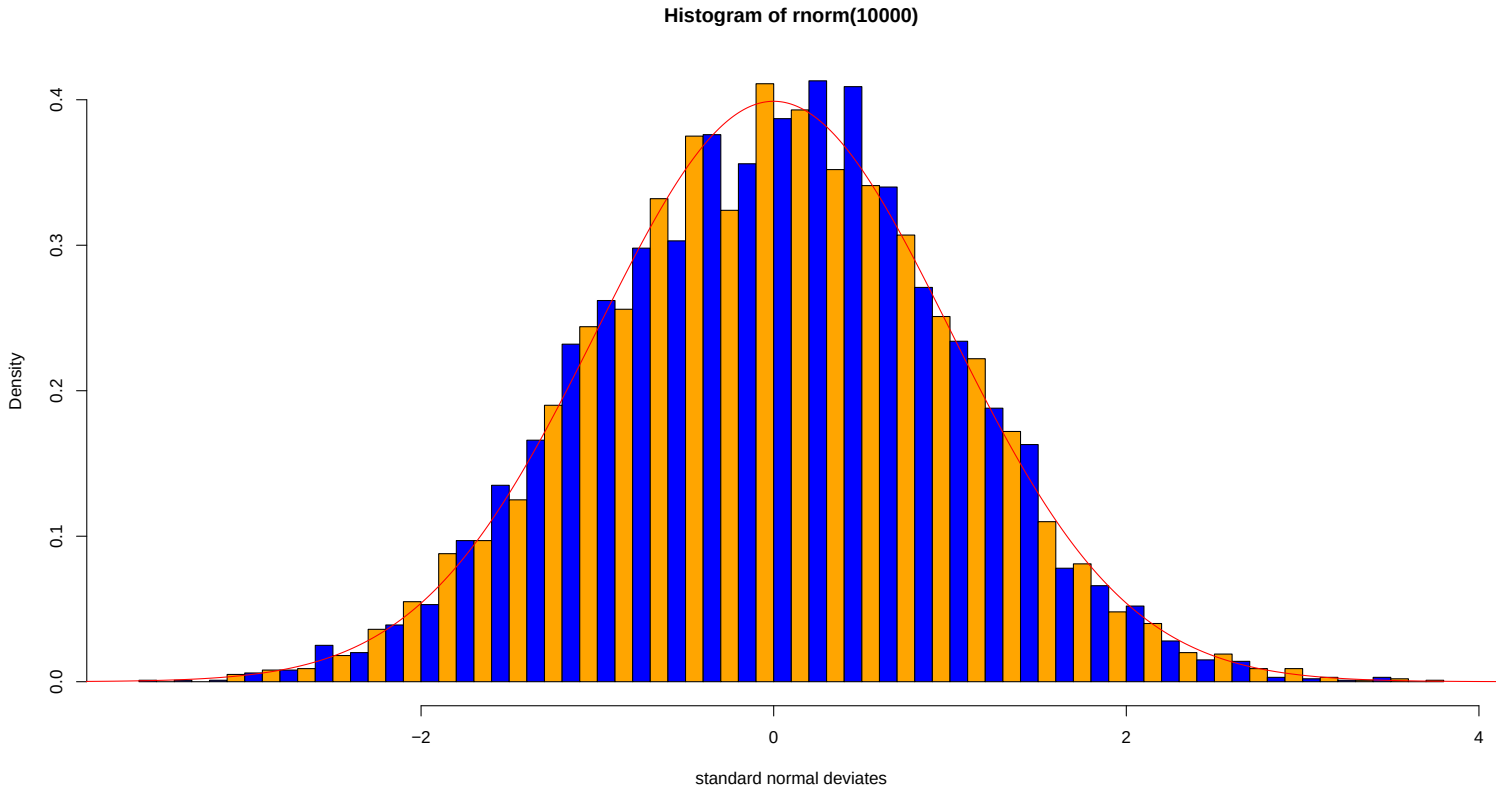
Simple Command Line Graphics

```
> hist(rnorm(10000),nclass=101,col=c("blue","orange"),
+ xlab="standard normal deviates") # rnorm(10000) produces a vector
>                                  # of 10000 standard normal deviates
```



Augmented Command Line Graphics

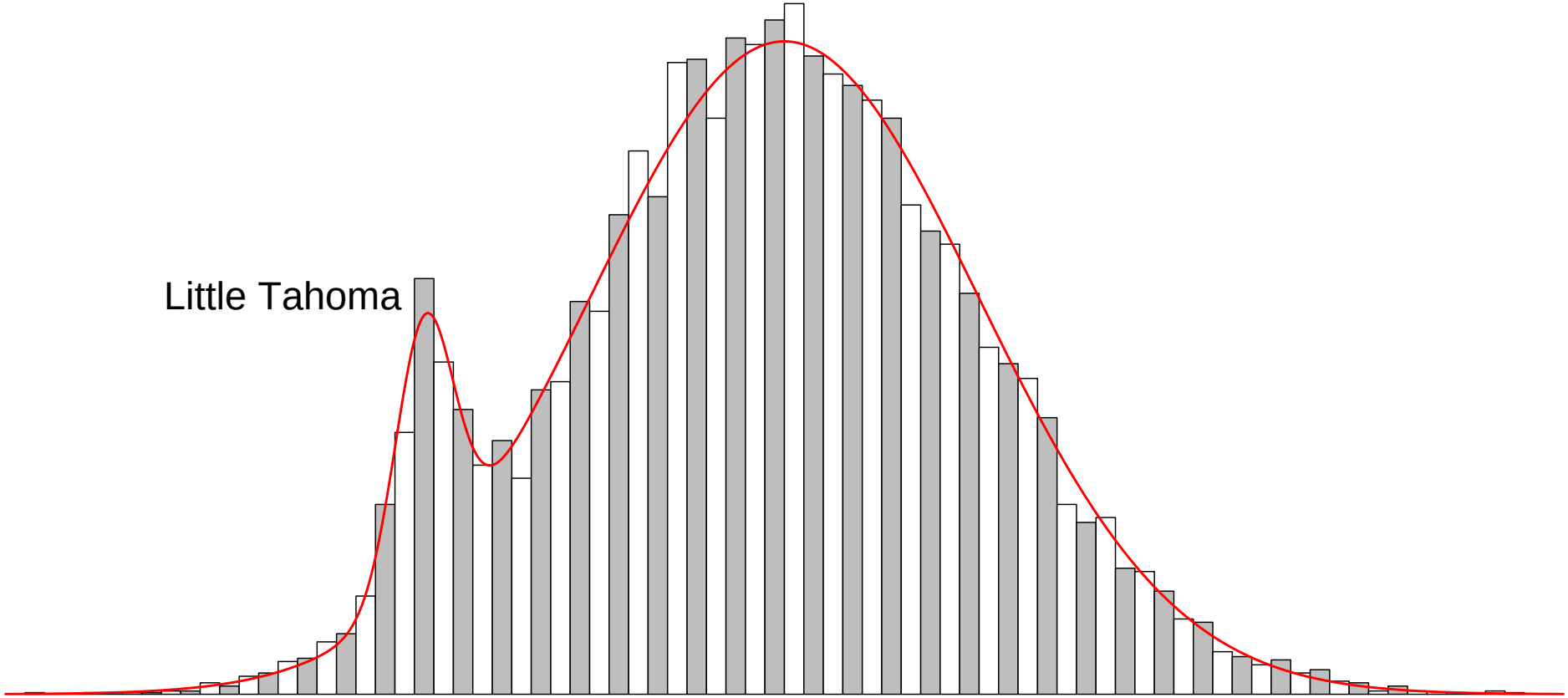
```
> hist(rnorm(10000),nclass=101,col=c("blue","orange"),probability=T,
+ xlab="standard normal deviates")
> lines(seq(-5,5,.01),dnorm(seq(-5,5,.01)),col="red")
# the probability = T option renders the histogram with area 1
# for better comparison with the superimposed density.
```



Text Annotated Graphics

```
hist.rainier=function ()
{
x=rnorm(10000)
y=rnorm(600,-1.85,.15)
hist(c(x,y),nclass=91,probability=T,main="",xlab="",ylab="",axes=F,
col=c("grey","white"))
xx=seq(-4,4,.01)
yy=600/10600*dnorm(xx,-1.85,.15)+10000/10600*dnorm(xx)
lines(xx,yy,lwd=2,col="red")
text(-2,.23,"Little Tahoma",adj=1)
}
```

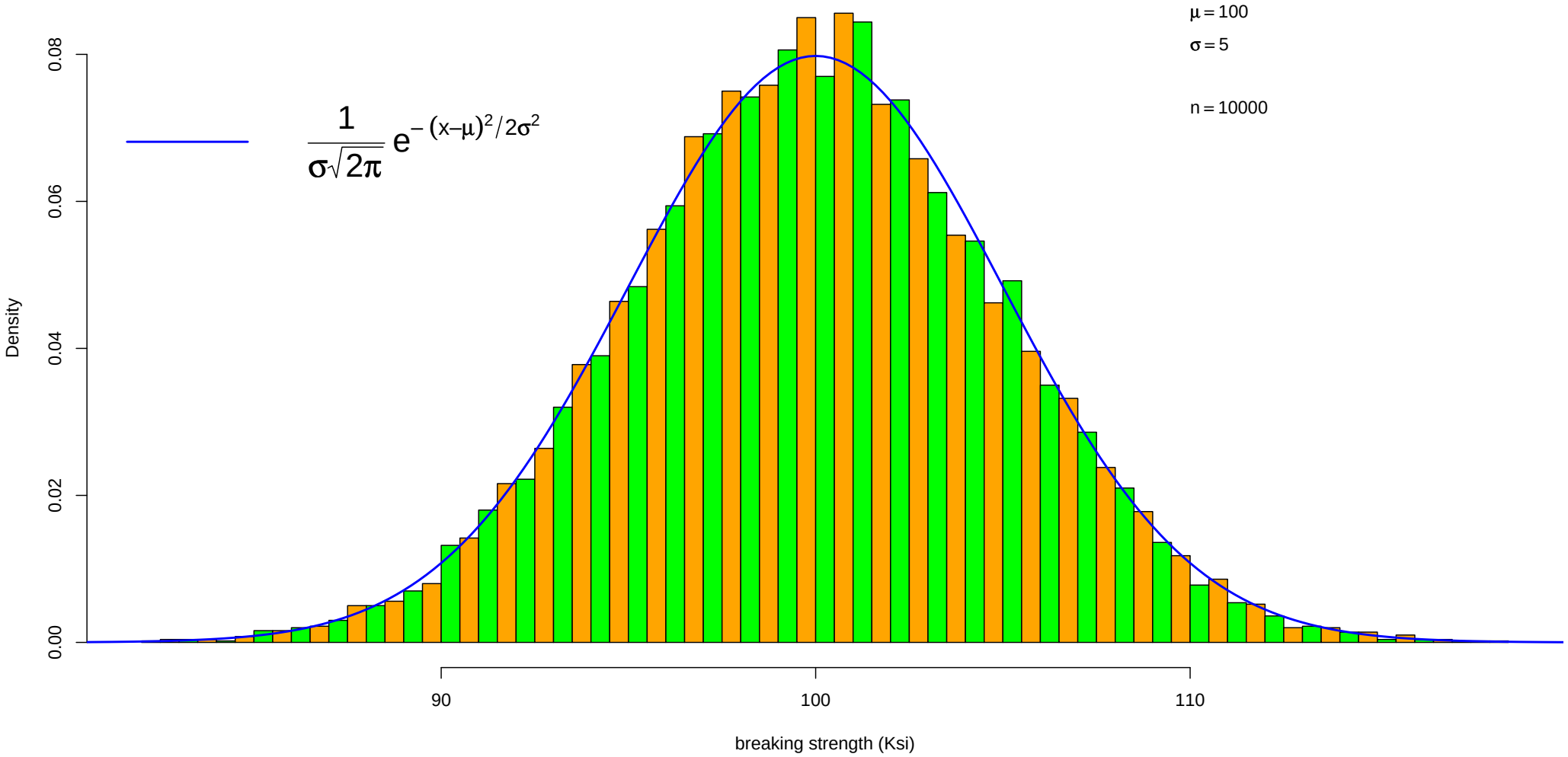
The Resulting Plot



Math Annotated Graphics

```
plot.fun=function (mu=100,sig=5,n=10000)
{
x=rnorm(n,mu,sig)
out=hist(x,nclass=101,probability=T,xlab="breaking strength (Ksi)",
main="",col=c("green","orange"))
M=max(out$density) # M is height of highest bar in histogram
text(mu+2*sig,.95*M,substitute(sigma==sigx,list(sigx=sig)),adj=0)
text(mu+2*sig,M,substitute(mu==mux,list(mux=mu)),adj=0)
text(mu+2*sig,.85*M, substitute(n==nx,list(nx=n)),adj=0)
xx=seq(mu-4*sig,mu+4*sig,length.out=201)
lines(xx,dnorm(xx,mu,sig),lwd=2,col="blue")
legend(80,.08,expression(over(1,sigma*sqrt(2*pi))*~
e^{-~(x-mu)^2/2*sigma^2}),
bty="n",lty=1,col="blue",lwd=2,cex=1.8)
}
```

The Resulting Math Annotated Plot



Factors

```
Torque=read.csv("Torque.csv",header=T) # Torque.csv is a data set
> Torque[1:5,]                  # with 3 columns: Screw Plating Torque
  Screw Plating Torque
1  bolt       CW     20
2  bolt       CW     16
3  bolt       CW     17
4  bolt       CW     18
5  bolt       CW     15
> Torque[31:35,]
     Screw Plating Torque
31 mandrel       CW     24
32 mandrel       CW     18
33 mandrel       CW     17
34 mandrel       CW     17
35 mandrel       CW     15
> is.factor(Torque$Plating)
[1] TRUE
> is.factor(Torque$Torque)
[1] FALSE
```

About Factors

read.table or read.csv reads character data as a factor, coded internally as numbers. Factors are useful for categorical or grouped data.

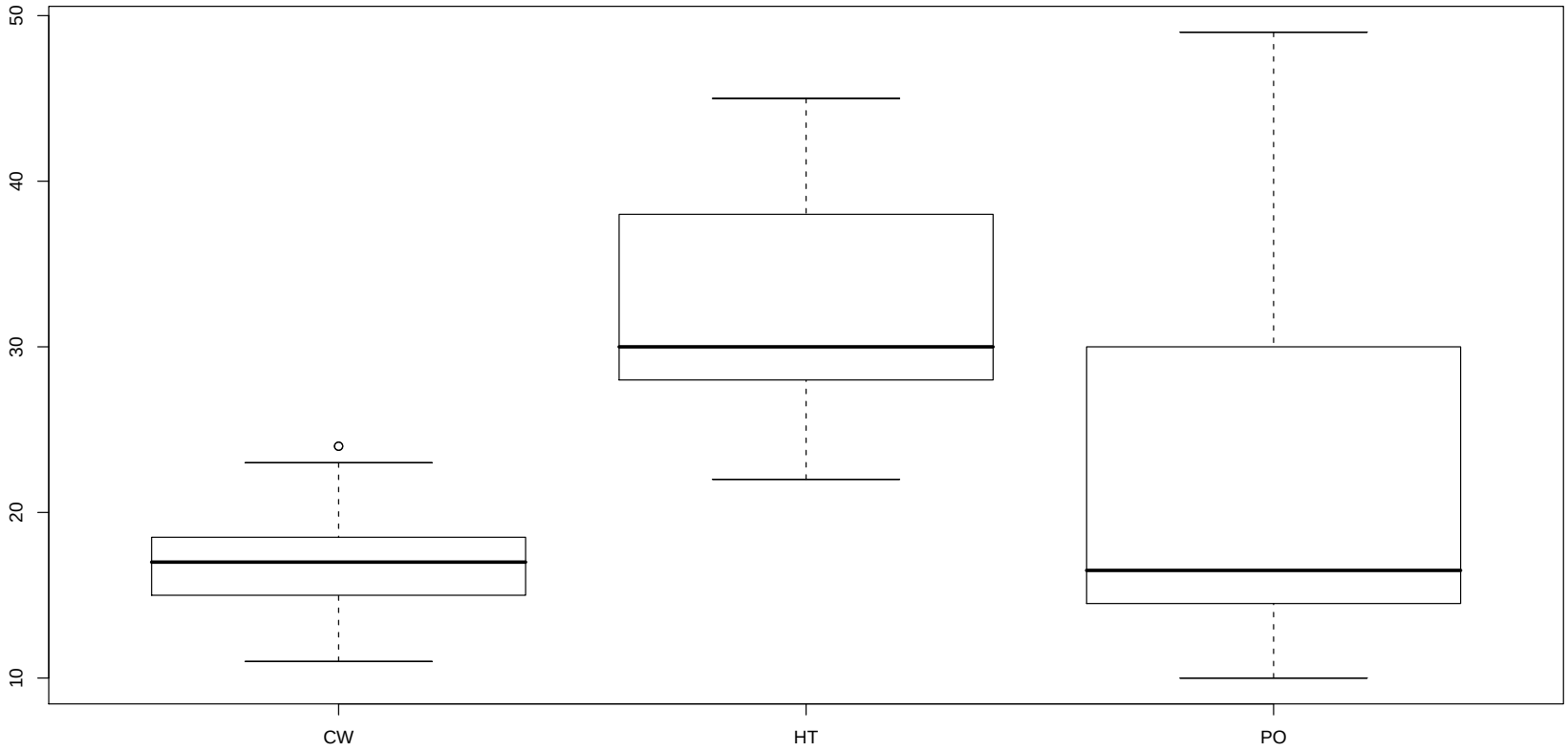
They are treated in special ways in many functions.

Each number or category also has a name, stored in levels.

```
> Torque$Plating[1:20]
 [1] CW CW CW CW CW CW CW CW CW CW HT HT HT HT HT HT HT HT
[19] HT HT
Levels: CW HT PO
>  as.numeric(Torque$Plating)
 [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3
[28] 3 3 3 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 3 3 3 3
[55] 3 3 3 3 3 3
> levels(Torque$Plating)
[1] "CW" "HT" "PO"
> Torque$Plating[1]=="CW"
[1] TRUE
```

An Example Use of Factors

```
> boxplot(Torque~Plating,data=Torque) #data=Torque identifies the
  # data.frame Torque. Torque~Plating identifies the variable Torque
  # in that data.frame. More on ~ later in the course.
```



attributes and mode

Sometimes you don't know or forget the nature of objects in your workspace.

```
> x=as.factor(letters[1:4])
> x
[1] a b c d
Levels: a b c d
> mode(x)
[1] "numeric"
> attributes(x)
$levels
[1] "a" "b" "c" "d"

$class
[1] "factor"
```

Packages

Aside from your own extensions of **R** via functions there are numerous packages of functions that were contributed by others.

A frivolous example: click on Packages on the tool bar of an **R** session and select Install Package(s). Select USA (WA) under the offered CRAN mirror sites.

Under the offered packages list scroll down to sudoku and select it for installation.

After it is installed you still need to invoke `library(sudoku)` on the command line for each new **R** session in order to have access to the sudoku functions.

To see the functions available in this package look under sudoku in the packages section of the web help initiated by `help.start()`. Or use `?playSudoku`.

To play, type `playSudoku()`. The command window gives cryptic instructions, to be carried out on graphics window (console is inactive until you quit the game).

The Future of Statistics by Bradley Efron

AMSTAT News, September 2007

"My own life as a consulting biostatistician has been revolutionized in the space of a single career.

Sitting in front of a powerful terminal and calling up R functions to all the dozen advances and a lot more really does put seven-league boots an a statistician's feet.

Maybe 77-league boots lie in our immediate future.. . . "