

A Not-so-brief Introduction to R

Sean Davis
with large contributions by
Naomi Altman and Mark Reimers

National Cancer Institute
National Institutes of Health

sdavis2@mail.nih.gov

- 1 R Overview
- 2 R Mechanics
- 3 Resources for Getting Help
- 4 Vectors
- 5 Rectangular Data
- 6 Plotting and Graphics
- 7 Control Structures, Looping, and Applying
- 8 Functions

What is R?

- A software package
- A programming language
- A toolkit for developing statistical and analytical tools
- An extensive library of statistical and mathematical software and algorithms
- A scripting language
- . . .

Why R?

- R is cross-platform and runs on Windows, Mac, and Linux (as well as more obscure systems).
- R provides a vast number of useful statistical tools, many of which have been painstakingly tested.
- R produces publication-quality graphics in a variety of formats.
- R plays well with FORTRAN, C, and scripts in many languages.
- R scales, making it useful for small and large projects. It is NOT Excel.
- R eschews the GUI.

Anecdote

I can develop code for analysis on my Mac laptop. I can then install the *same* code on our 20k core cluster and run it in parallel on 100 samples, monitor the process, and then update a database with R when complete.

Why Not R?

- R cannot do everything.
- R is not always the “best” tool for the job.
- R will *not* hold your hand.
- The documentation can be opaque.
- R can drive you crazy (on a good day) or age you prematurely (on a bad one).
- Finding the right package to do the job you want to do can be challenging; worse, some contributed packages are unreliable.
- R eschews the GUI.


R License and the Open Source Ideal

- R is free!
- Distributed under GNU license
 - You may download the source code.
 - You may modify the source code to your heart's content.
 - You may distribute the modified source code and even charge money for it, but you must distribute the modified source code under the original GNU license

Take-home Message

This license means that R will always be available, will always be open source, and can grow organically without constraint.

Installing R



CRAN
[Mirrors](#)
[What's new?](#)
[Task Views](#)
[Search](#)

About R
[R Homepage](#)
[The R Journal](#)

Software
[R Sources](#)
[R Binaries](#)
[Packages](#)
[Other](#)

Documentation
[Manuals](#)
[FAQs](#)
[Contributed](#)

The Comprehensive R Archive Network

Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- [Download R for Linux](#)
- [Download R for \(Mac\) OS X](#)
- [Download R for Windows](#)

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

Source Code for all Platforms

Windows and Mac users most likely want to download the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!

- The latest release (2013-05-16, Masked Marvel): [R-3.0.1.tar.gz](#), read [what's new](#) in the latest version.
- Sources of [R alpha and beta releases](#) (daily snapshots, created only in time periods before a planned release).
- Daily snapshots of current patched and development versions are [available here](#). Please read about [new features and bug fixes](#) before filing corresponding feature requests or bug reports.
- Source code of older versions of R is [available here](#).
- Contributed extension [packages](#)

Questions About R

- If you have questions about R like how to download and install the software, or what the license terms

Taskbar: Paradise-redebts_en.pdf, Robinson-icebreaker.pdf, R+Pre-Course+PM (2).pptx, R+Pre-Course+AM (2).pptx, RPreCourseAnnounce....doc, CMP63_061413.pptx, Show All

Starting R

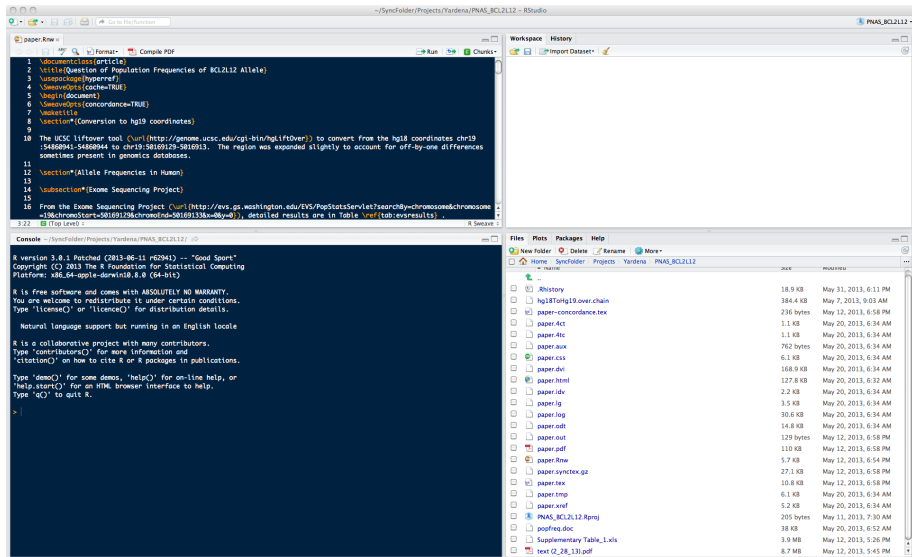
- Depends on operating system and interface

Linux command line

```
$ R
```

- In this course, we will largely be using a GUI called *RStudio*

The RStudio Interface



Getting Started

- R Commands are either:

- ① Assignments

```
x = 1  
y <- 2
```

- ② Expressions

```
1 + pi + sin(3.7)  
## [1] 3.611757
```

- The “<-” and “=” are both assignment operators.
- The standard R prompt is a “>” sign.
- If a line is not a complete R command, R will continue the next line with a “+”.

```
1 + pi +  
sin(3.7)  
## [1] 3.611757
```

Rules for Names in R

- Any combination of letters, numbers, underscore, and “.”
- May not start with numbers, underscore.
- R is case-sensitive.

Examples

```
pi
x
camelCaps
my_stuff
MY_Stuff
this.is.the.name.of.the.man
ABC123
abc1234asdf
.hi
```

R Help Functions

- If you know the name of the function or object on which you want help:

```
help(print)
help('print')
?print
```

- If you *do not* know the name of the function or object on which you want help:

```
help.search('microarray')
RSiteSearch('microarray')
```

- Many online resources which you will collect over the space of the course

Using Help

I strongly recommend using `help(newfunction)` for all functions that are new or unfamiliar to you.

- In R, even a single value is a vector with `length=1`.

```
z = 1
z
## [1] 1
length(z)
## [1] 1
```

- Vectors can contain numbers, strings (character data), or logical values (TRUE and FALSE)
- Vectors cannot contain a mix of types!

Character Vectors

Character vectors are entered with each value surrounded by single or double quotes; either is acceptable, but they must match. They are always displayed by R with double quotes.

Example Vectors

```
# examples of vectors
c('hello','world')
## [1] "hello" "world"
c(1,3,4,5,1,2)
## [1] 1 3 4 5 1 2
c(1.12341e7,78234.126)
## [1] 11234100.00      78234.13
c(TRUE,FALSE,TRUE,TRUE)
## [1] TRUE FALSE TRUE TRUE
# note how in the next case the TRUE is converted to "TRUE"
c(TRUE,'hello')
## [1] "TRUE" "hello"
```

Regular Sequences

```
# create a vector of integers from 1 to 10
x = 1:10
# and backwards
x = 10:1
# create a vector of numbers from 1 to 4 skipping by 0.3
y = seq(1,4,0.3)
# create a sequence by concatenating two other sequences
z = c(y,x)
z
## [1] 1.0 1.3 1.6 1.9 2.2 2.5 2.8 3.1 3.4 3.7 4.0 10.0 9.0 8.0
## [15] 7.0 6.0 5.0 4.0 3.0 2.0 1.0
```

Vector Operations

- Operations on a single vector are typically done element-by-element
- If the operation involves two vectors:
 - Same length: R simply applies the operation to each pair of elements.
 - Different lengths, but one length a multiple of the other: R reuses the shorter vector as needed
 - Different lengths, but one length *not* a multiple of the other: R reuses the shorter vector as needed *and* delivers a warning
- Typical operations include multiplication ("*"), addition, subtraction, division, exponentiation ("^"), but many operations in R operate on vectors and are then called "vectorized".

Summary of Simple Data Types

Data type	Stores
real	floating point numbers
integer	integers
complex	complex numbers
factor	categorical data
character	strings
logical	TRUE or FALSE
NA	missing
NULL	empty
function	function type

Vector Operations

```
x = 1:10
x+x
## [1] 2 4 6 8 10 12 14 16 18 20
y = 7
x * y
## [1] 7 14 21 28 35 42 49 56 63 70
y = c(1,2,3)
z = x * y

## Warning in x * y: longer object length is not a multiple of shorter object
length

length(z)
## [1] 10
z
## [1] 1 4 9 4 10 18 7 16 27 10
```

Logical Vectors

Logical vectors are vectors composed on only the values TRUE and FALSE. Note the all-upper-case and no quotation marks.

```
a = c(TRUE,FALSE,TRUE)
# we can also create a logical vector from a numeric vector
# 0 = false, everything else is 1
b = c(1,0,217)
d = as.logical(b)
d
## [1] TRUE FALSE TRUE
# test if a and d are the same at every element
all.equal(a,d)
## [1] TRUE
# We can also convert from logical to numeric
as.numeric(a)
## [1] 1 0 1
```

Logical Operators

Some operators like `<`, `>`, `==`, `>=`, `<=`, `!=` can be used to create logical vectors.

```
# create a numeric vector
x = 1:10
# testing whether x > 5 creates a logical vector
x > 5
## [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
x <= 5
## [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
x != 5
## [1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
x == 5
## [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
# we can also assign the results to a variable
y = (x == 5)
y
## [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
```

Indexing Vectors

- In programming, an index is used to refer to a specific element or set of elements in an vector (or other data structure).
- R uses [and] to perform indexing.

```
x = seq(0,1,0.1)
# create a new vector from the 4th element of x
x[4]
## [1] 0.3
```

- Indexing can use other vectors for the indexing

```
x[c(3,5,6)]
## [1] 0.2 0.4 0.5
y = 3:6
x[y]
## [1] 0.2 0.3 0.4 0.5
```

Indexing Vectors and Logical Vectors

Combining the concept of indexing with the concept of logical vectors results in a very power combination.

```
# use help('rnorm') to figure out what is happening next
myvec = rnorm(10)
# create logical vector that is TRUE where myvec is >0.25
gt1 = (myvec > 0.25)
sum(gt1)
## [1] 2
# and use our logical vector to create a vector of myvec values that are >0.25
myvec[gt1]
## [1] 0.8500783 1.2182233
# or <=0.25 using the logical "not" operator, "!"
myvec[!gt1]
## [1] -0.52361528 -1.96734990 0.09722437 -0.09474050 -0.92634247 0.10862202
## [7] -0.01423129 -1.10639317
# shorter, one line approach
myvec[myvec > 0.25]
## [1] 0.8500783 1.2182233
```

Concatenating Strings

R uses the `paste` function to concatenate strings.

```
paste("abc", "def")
## [1] "abc def"
paste("abc", "def", sep="THISSEP")
## [1] "abcTHISSEPdef"
paste0("abc", "def")
## [1] "abcdef"
paste(c("X", "Y"), 1:10)
## [1] "X 1" "Y 2" "X 3" "Y 4" "X 5" "Y 6" "X 7" "Y 8" "X 9" "Y 10"
paste(c("X", "Y"), 1:10, sep="_")
## [1] "X_1" "Y_2" "X_3" "Y_4" "X_5" "Y_6" "X_7" "Y_8" "X_9" "Y_10"
```

More String Functions

- Number of characters in a string

```
nchar('abc')  
## [1] 3  
nchar(c('abc','d',123456))  
## [1] 3 1 6
```

- Extract substrings

```
substr('This is a good sentence.',start=10,stop=15)  
## [1] " good "
```

- String replacement

```
sub('This','That','This is a good sentence.')  
## [1] "That is a good sentence."
```

- Finding matching strings

```
grep('bcd',c('abcdef','abcd','bcde','cdef','defg'))  
## [1] 1 2 3  
grep('bcd',c('abcdef','abcd','bcde','cdef','defg'),value=TRUE)  
## [1] "abcdef" "abcd"   "bcde"
```


Missing Values, AKA “NA”

R has a special value, “NA”, that represents a “missing” value in a vector or other data structure.

```
x = 1:5
x
## [1] 1 2 3 4 5
length(x)
## [1] 5
is.na(x)
## [1] FALSE FALSE FALSE FALSE FALSE
x[2] = NA
x
## [1] 1 NA 3 4 5
length(x)
## [1] 5
is.na(x)
## [1] FALSE TRUE FALSE FALSE FALSE
x[!is.na(x)]
## [1] 1 3 4 5
```

- A factor is a special type of vector, normally used to hold a categorical variable in many statistical functions.
- Such vectors have class “factor”.
- Factors are primarily used in Analysis of Variance (ANOVA). When a factor is used as a predictor variable, the corresponding indicator variables are created.

Note of caution

Factors in R often *appear* to be character vectors when printed, but you will notice that they do not have double quotes around them. They are stored in R as numbers with a key name, so sometimes you will note that the factor *behaves* like a numeric vector.

Factors in Practice

```
# create the character vector
citizen<-c("uk","us","no","au","uk","us","us","no","au")
# convert to factor
citizenf<-factor(citizen)
citizen
## [1] "uk" "us" "no" "au" "uk" "us" "us" "no" "au"
citizenf
## [1] uk us no au uk us us no au
## Levels: au no uk us
# convert factor back to character vector
as.character(citizenf)
## [1] "uk" "us" "no" "au" "uk" "us" "us" "no" "au"
# convert to numeric vector
as.numeric(citizenf)
## [1] 3 4 2 1 3 4 4 2 1
```

Factors in Practice

```
# R stores many data structures as vectors with "attributes" and "class"
attributes(citizenf)
## $levels
## [1] "au" "no" "uk" "us"
##
## $class
## [1] "factor"
class(citizenf)
## [1] "factor"
# note that after unclassing, we can see the
# underlying numeric structure again
unclass(citizenf)
## [1] 3 4 2 1 3 4 4 2 1
## attr(,"levels")
## [1] "au" "no" "uk" "us"
table(citizenf)
## citizenf
## au no uk us
## 2 2 2 3
```

Matrices and Data Frames

- A matrix is a rectangular array. It can be viewed as a collection of column vectors all of the same length and the same type (i.e. numeric, character or logical).
- A data frame is *also* a rectangular array. All of the columns must be the same length, but they may be of *different* types.
- The rows and columns of a matrix or data frame can be given names.
- However these are implemented differently in R; many operations will work for one but not both.

Matrix Operations

```
x<-1:10
y<-rnorm(10)
# make a matrix by column binding two numeric vectors
mat<-cbind(x,y)
mat
##           x           y
## [1,]  1  0.3842634
## [2,]  2 -1.2536470
## [3,]  3 -1.4098608
## [4,]  4  1.7139083
## [5,]  5 -0.1876347
## [6,]  6 -0.3343492
## [7,]  7  1.6553040
## [8,]  8  0.5974976
## [9,]  9 -1.8443792
## [10,] 10  1.4203497
# And the names of the rows and columns
rownames(mat)
## NULL
colnames(mat)
## [1] "x" "y"
```

Matrix Operations

Indexing for matrices works as for vectors except that we now need to include both the row and column (in that order).

```
# The 2nd element of the 1st row of mat
mat[1,2]
##           y
## 0.3842634

# The first ROW of mat
mat[1,]
##           x           y
## 1.0000000 0.3842634

# The first COLUMN of mat
mat[,1]
## [1] 1 2 3 4 5 6 7 8 9 10

# and all elements of mat that are > 4; note no comma
mat[mat>4]
## [1] 5 6 7 8 9 10
```

Matrix Operations

```
# create a matrix with 2 columns and 10 rows
# filled with random normal deviates
m = matrix(rnorm(20),nrow=10)
# multiply all values in the matrix by 20
m = m*20
# and add 100 to the first column of m
m[,1] = m[,1] + 100
# summarize m
summary(m)
```

##	V1	V2
##	Min. : 44.58	Min. : -42.4628
##	1st Qu.: 93.26	1st Qu.: -19.8712
##	Median : 100.72	Median : 3.6691
##	Mean : 95.27	Mean : -0.4271
##	3rd Qu.: 106.33	3rd Qu.: 17.3559
##	Max. : 111.29	Max. : 31.2418

Matrices Versus Data Frames

```
mat<-cbind(x,y)
class(mat[,1])
## [1] "numeric"
z = paste0('a',1:10)
tab<-cbind(x,y,z)
class(tab)
## [1] "matrix"
mode(tab[,1])
## [1] "character"
head(tab,4)
##      x      y      z
## [1,] "1"    "0.384263423333191" "a1"
## [2,] "2"    "-1.25364703104426"  "a2"
## [3,] "3"    "-1.40986082624299"  "a3"
## [4,] "4"    "1.7139082579506"    "a4"
```

Matrices Versus Data Frames

```
tab<-data.frame(x,y,z)
class(tab)
## [1] "data.frame"
head(tab)
##      x          y  z
## 1 1  0.3842634 a1
## 2 2 -1.2536470 a2
## 3 3 -1.4098608 a3
## 4 4  1.7139083 a4
## 5 5 -0.1876347 a5
## 6 6 -0.3343492 a6
mode(tab[,1])
## [1] "numeric"
class(tab[,3])
## [1] "factor"
rownames(tab)
## [1] "1" "2" "3" "4" "5" "6" "7" "8" "9" "10"
rownames(tab)<-paste0("row",1:10)
rownames(tab)
## [1] "row1" "row2" "row3" "row4" "row5" "row6" "row7" "row8"
## [9] "row9" "row10"
```

Data Frames, Continued

- Data frame columns can be referred to by name using the “dollar sign” operator

```
tab$x
## [1] 1 2 3 4 5 6 7 8 9 10
tab$y
## [1] 0.3842634 -1.2536470 -1.4098608 1.7139083 -0.1876347 -0.3343492
## [7] 1.6553040 0.5974976 -1.8443792 1.4203497
```

- Column names can be set, which can be useful for referring to data later

```
colnames(tab)
## [1] "x" "y" "z"
colnames(tab) = paste0('col',1:3)
```

Exercise: Subsetting Data Frames

Try these

```
ncol(tab)
nrow(tab)
dim(tab)
summary(tab)
tab[1:3,]
tab[,2:3]
tab[,1]>7
tab[tab[,1]>7,]
tab[tab[,1]>7,3]
tab[tab[,1]>7,2:3]
tab[tab$x>7,3]
tab$z[tab$x>3]
```

Reading and Writing Data Frames to Disk

- The `write.table` function and friends write a `data.frame` or matrix to disk as a text file.

```
write.table(tab,file='tab.txt',sep="\t",col.names=TRUE)
# remove tab from the workspace
rm(tab)
# make sure it is gone
ls(pattern="tab")
## character(0)
```

- The `read.table` function and friends read a `data.frame` or matrix from a text file.

```
tab = read.table('tab.txt',sep="\t",header=TRUE)
head(tab,3)
##           col1           col2 col3
## row1         1  0.3842634   a1
## row2         2 -1.2536470   a2
## row3         3 -1.4098608   a3
```

- A list is a collection of objects that may be the same or different types.
- The objects generally have names, and may be indexed either by name (e.g. `my.list$name3`) or component number (e.g. `my.list[[3]]`).
- A data frame is a list of matched column vectors.

Lists in Practice

- Create a list, noting the different data types involved.

```
a = list(1,"b",c(1,2,3))
a
## [[1]]
## [1] 1
##
## [[2]]
## [1] "b"
##
## [[3]]
## [1] 1 2 3
length(a)
## [1] 3
class(a)
## [1] "list"
a[[3]]
## [1] 1 2 3
```

- A data frame *is* a list.

```
# test if our friend "tab" is a list
is.list(tab)
## [1] TRUE
tab[[2]]
## [1] 0.3842634 -1.2536470 -1.4098608 1.7139083 -0.1876347 -0.3343492
## [7] 1.6553040 0.5974976 -1.8443792 1.4203497
names(tab)
## [1] "col1" "col2" "col3"
```


Summary of Simple Data Types

Data type	Stores
real	floating point numbers
integer	integers
complex	complex numbers
factor	categorical data
character	strings
logical	TRUE or FALSE
NA	missing
NULL	empty
function	function type

Summary of Aggregate Data Types

Data type	Stores
vector	one-dimensional data, single data type
matrix	two-dimensional data, single data type
data frame	two-dimensional data, multiple data types
list	list of data types, not all need to be the same type
object	a list with attributes and potentially slots and methods

Basic Plot Functions

- The command `plot(x,y)` will plot vector `x` as the independent variable and vector `y` as the dependent variable.
- Within the command line, you can specify the title of the graph, the name of the x-axis, and the name of the y-axis.
 - `main='title'`
 - `xlab='name of x axis'`
 - `ylab='name of y axis'`
- The command `lines(x,y)` adds a line segment to the plot.
- The command `points(x,y)` adds points to the plot.
- A legend can be created using `legend`.

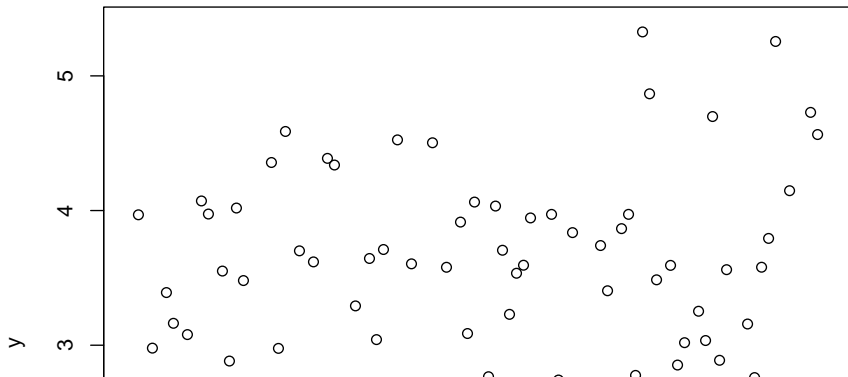
demo

```
demo(graphics)
```

Simple Plotting Example

Try this yourself:

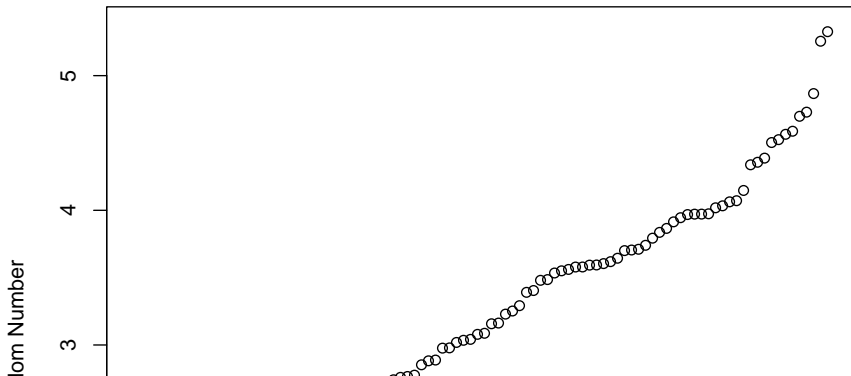
```
x = 1:100  
y = rnorm(100,3,1) # 100 random normal deviates with mean=3, sd=1  
plot(x,y)
```



More Plotting

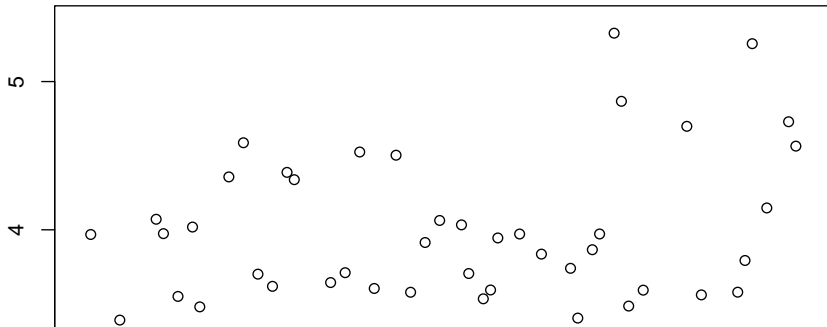
```
z=sort(y)
# plot a sorted variable vs x
plot(x,z,main='Random Normal Numbers',
     xlab='Index',ylab='Random Number')
```

Random Normal Numbers



More Plotting

```
# check margin and outer margin settings
par(c("mar", "oma"))
plot(x,y)
par(oma=c(1,1,1,1)) # set outer margin
plot(x,y)
par(mar=c(2.5,2.1,2.1,1)) # set margin
plot(x,y)
```



Graphics Devices and Saving Plots

- to make a plot directly to a file use: `png()`, `postscript()`, etc.
- R can have multiple graphics “devices” open.
 - To see a list of active devices: `dev.list()`
 - To close the most recent device: `dev.off()`
 - To close device 5: `dev.off(5)`
 - To use device 5: `dev.set(5)`

More Plotting

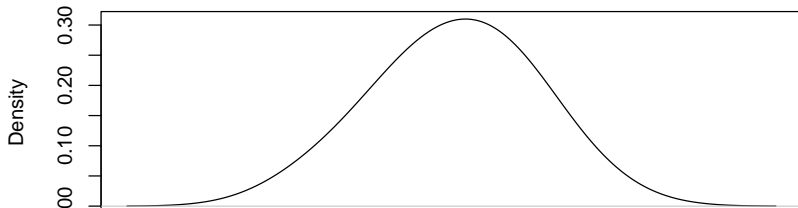
- Save a png image to a file

```
png(file="myplot.png",width=480,height=480)  
plot(density(z,adjust=2.0),sub="larger bandwidth")  
dev.off()
```

- On your own, save a pdf to a file. NOTE: The dimensions in `pdf()` are in *inches*
- Multiple plots on the same page:

```
par(mfrow=c(2,1))  
plot(density(z,adjust=2.0),sub="larger bandwidth")  
hist(z)
```

density.default(x = z, adjust = 2)



Visit these sites for some ideas.

- <http://www.sr.bham.ac.uk/~ajrs/R/r-gallery.html>
- <http://gallery.r-enthusiasts.com/>
- <http://cran.r-project.org/web/views/Graphics.html>

Control Structures in R

- R has multiple types of control structures that allows for sequential evaluation of statements.

- For loops

```
for (x in set) {operations}
```

- while loops

```
while (x in condition){operations}
```

- If statements (conditional)

```
if (condition) {  
  some operations  
} else { other operations }
```

Control Structure and Looping Examples

```
x<-1:9
length(x)
# a simple conditional then two expressions
if (length(x)<=10) {
  x<-c(x,10:20);print(x)}
# more complex
if (length(x)<5) {
  print(x)
} else {
  print(x[5:20])
}
# print the values of x, one at a time
for (i in x) print(i)
for(i in x) i # note R will not echo in a loop
```

Control Structure and Looping Examples

```
# loop over a character vector
y<-c('a','b','hi there')
for (i in y) print(i)

# and a while loop
j<-1
while(j<10) { # do this while j<10
  print(j)
  j<-j+2} # at each iteration, increase j by 2
```

Why Does R Have Apply Functions

- Often we want to apply the same function to all the rows or columns of a matrix, or all the elements of a list.
- We could do this in a loop, but loops take a lot of time in an interpreted language like R.
- R has more efficient built-in operators, the apply functions.

example

If `mat` is a matrix and `fun` is a function (such as `mean`, `var`, `lm ...`) that takes a vector as its argument, then you can:

```
apply(mat,1,fun) # over rows--second argument is 1
```

```
apply(mat,2,fun) # over columns--second argument is 2
```

In either case, the output is a vector.

Apply Function Exercise

- ➊ Using the `matrix` and `rnorm` functions, create a matrix with 20 rows and 10 columns (200 values total) of random normal deviates.
- ➋ Compute the mean for each row of the matrix.
- ➌ Compute the median for each column.

Related Apply Functions

- `lapply(list, function)` applies the function to every element of list
- `sapply(list or vector, function)` applies the function to every element of list or vector, and returns a vector, when possible (easier to process)
- `tapply(x, factor, fun)` uses the factor to split vector `x` into groups, and then applies `fun` to each group

Related Apply Function Examples

```
# create a list
my.list <- list(a=1:3,b=5:10,c=11:20)
my.list
# Get the mean for each member of the list
# return a vector
sapply( my.list, mean)
# Get the full summary for each member of
# the list, returned as a list
lapply( my.list, summary)
# Find the mean for each group defined by a factor
my.vector <- 1:10
my.factor <- factor(
  c(1,1,1,2,2,2,3,3,3,3))
tapply(my.vector, my.factor, mean)
```


Function Overview

- Functions are objects and are assigned to names, just like data.

```
myFunction = function(argument1,argument2) {  
  expression1  
  expression2  
}
```

- We write functions for anything we need to do again and again.
- You may test your commands interactively at first, and then use the `history()` feature and an editor to create the function.
- It is wise to include a comment at the start of each function to say what it does and to document functions of more than a few lines.

Example Functions

```
add1 = function(x) {  
  # this function adds one to the first argument and returns it  
  x + 1  
}  
add1(17)  
## [1] 18  
add1(c(17,18,19,20))  
## [1] 18 19 20 21
```

You can use the `edit()` function to make changes to a function. The following command will open a window, allow you to make changes, and assign the result to a new function, `add2`.

```
add2 = edit(add1)
```

The amount of learning material for R is simply astonishing!

- Thomas Girke's R and Bioconductor Manual
- A HUGE collection of contributed R documentation and tutorials
- Bioconductor course materials
- Sean Davis' website
- The Official R Manuals