

Introduction to R

Throughout this document, things you select from the menus or things you type in the Commands window will be in **bold**; prompts from the computer will not be. Text following a “#” is a comment and need not be typed.

R is available free for Linux, Mac OS, and Windows from the website <http://cran.r-project.org/>. It is also available in the Math 206 PC computer lab. There is considerable documentation on R at the R website and at various other places online; the following serves as a brief introduction to just a few of the capabilities of R.

1. Startup

When you start R, the R console appears with the prompt “>”. This is where you type commands into R; they are executed immediately when you hit Enter. You will often find it more convenient to use a Script window which allows you to type several commands and then execute them all at once, then go back and edit the commands and re-execute them, etc. This is very useful, particularly if you are doing repetitive analyses.

To open a Script window, go to **File...New script**. A new window will appear. You type your commands in the window, one per line. No “>” prompt appears; you simply type the commands in the order you would like them executed. When you are ready to execute the commands in the Script window, click **Edit...Run all**. You can also run a subset of the commands in a window by highlighting them and hitting Ctrl-R (or selecting **Edit...Run line or selection**). You can also run a single line by putting the cursor anywhere on that line and hitting Ctrl-R (the line does not need to be highlighted).

To save a script, select **File...Save** or **Save as**. R scripts should be saved with a .R extension (you will have to type this explicitly; R will not automatically append it to the script name).

To change the default directory where script files are saved or looked for, select **File...Change dir**. This will also be the directory where data files, graphs, etc. are looked for and saved.

2. Help

Help on R commands is available in a couple of different ways. If you know the name of the command, type it with a ? in front at the command prompt, e.g.,

```
> ?mean
```

You can also type **help(mean)**.

You can also access help on commands through the **Help** menu.

3. Function Syntax

There are many built-in functions in R (see below) that operate on numbers, vectors, matrices, etc. or which generate the same. Many functions have optional arguments; these are detailed in the help files. Sometimes arguments need to be specified by name, other times they do not. The following is intended to show you when.

As an example, the **seq** function generates sequences of numbers, with the default being :

```
> seq(3,8)
[1] 3 4 5 6 7 8
```

If you look in the help file, you will see that the first argument is labeled “from” and the second is labeled “to”. So I could have said,

```
> seq(from=3,to=8)
[1] 3 4 5 6 7 8
```

Obviously, this takes more typing so I don't want to do this if I don't have to. If I give the values of the arguments in exactly the same order as specified in the Help file, then I do not need to name the arguments. If I specify them in a different order, or if I skip some optional arguments, then I need to name them. For Example, I could say:

```
> seq(to=8,from=3)
```

The command **seq** also has other optional arguments after "from" and "to" which are in this order: **by**, **length.out**, **along.with** (**by** specifies the distance between consecutive numbers in the sequence, and **length.out** specifies the number of values in the sequence; you can only specify one of these). So, I could say:

```
> seq(3,8,2)          # arguments are specified in same order as in Help file: from, to, by
```

```
[1] 3 5 7
```

```
> seq(3,8,length.out=4)  # first two arguments are in proper order, third is not so must be named
```

```
[1] 3.000000 4.666667 6.333333 8.000000
```

The names of arguments can be abbreviated as long as the argument can be uniquely identified among all the possible arguments. For example, since none of the other arguments to **seq** start with the letter "l", I could use any of the following:

```
> seq(3,8,length=4)
```

```
> seq(3,8,len=4)
```

```
> seq(3,8,l=4)
```

Some arguments have default values which are specified in the help file.

4. Arithmetic

You can perform arithmetic calculations in R using the operators +, -, *, /, ^:

```
> (11+5)/(2^3)          # ^ indicates raising to a power
```

```
[1] 2                    # the answer is 2; ignore the [1]; R generally represents any set of numbers  
                        (even one number) as a vector and the [1] simply indicates that the value 2 is the  
                        first (and only) element of a vector.
```

You can also use any of the many built-in functions for computations:

```
> sqrt(11+5)
```

```
[1] 4
```

Other functions include: **log()**, **exp()**, **log10()**, **abs()**, **sin()**, **cos()**, **gamma()**, **lgamma()**, **factorial()**, **choose(,)**

```
> factorial(10)
```

```
[1] 3628800
```

```
> choose(10,3)
```

```
[1] 120
```

For help on any command, use a question mark before the command name:

```
> ?choose
```

5. Variables

You can store objects by assigning a name using the assignment operator "=" (or, equivalently, "<-" : two characters: "less than" followed by "minus"; you will see this in much of the documentation, so it is good to be aware of it). R is case sensitive (A is different from a).

```
> a = 5
> a          # if you type the name of an object, R will give its value
[1] 5
```

```
> aa = sqrt(13*3)
> aa
[1] 6.244998
```

```
> b          # I haven't stored anything in b yet so I get an error
Error: Object "b" not found
```

Valid names in R can be of any length, must start with a letter, but can contain any combination of upper and lower-case letters, numbers, and periods. The following are all valid names:

```
a
Fish.12w
very.long.name.containing.many.characters.12345
```

You can use named variables in calculations, as in the following:

```
> N = 100
> d = 2
> N/d
[1] 50
```

6. Vectors

You can create vectors in R using the function “c”:

```
> x = c(2,3,5,1,4,4)
> x
[1] 2 3 5 1 4 4
```

Sequences of numbers can be created with some shortcuts:

```
> 1:5
[1] 1 2 3 4 5
```

```
> y = seq(1,3,by=.5)
> y
[1] 1.0 1.5 2.0 2.5 3.0
```

```
> y = seq(0,2,length=10)
> y
[1] 0.0000000 0.2222222 0.4444444 0.6666667 0.8888889 1.1111111 1.3333333
[8] 1.5555556 1.7777778 2.0000000
```

You can refer to subsets of the elements of a vector:

```
> x[3]
[1] 5
```

```
> x[1:2]      # first two elements
> x[c(1,3)]   # first and third elements
> x[-1]       # all elements except the first
```

```
>x[-c(2,4)] # all elements except the 2nd and 4th
```

There are many functions which operate on numerical vectors including **mean()**, **var()**, **stdev()**, **median()**, **sum()**, **max()**, **min()**.

So, for example, to get the sample standard deviation of the numbers in x:

```
> sqrt(var(x))
[1] 1.47196
```

Most functions which operate on a single number will also operate on a vector and will yield a vector as the result. For example:

```
> x = c(1,4,9,16)
> sqrt(x)
[1] 1 2 3 4
> c(1,2,3)^2 - 3
[1] -2 1 6
```

Most operations are done elementwise, including arithmetic operations:

```
> a = c(1,4,12)
> b = c(1,2,3)
> a/b
[1] 1 2 4
```

Vectors can also store character data:

```
> a = c("A","B","C","D")
> a[3]
[1] "C"
```

7. Logical operators

The logical operators are **<**, **>**, **==** (for equality), **<=**, **>=**, **!=** (for not equal). They operate on vectors and yield the values T (for True), F (for False) and NA (for missing). For example,

```
> x = c(3,8,6,7,3,4,9)
> x > 5
[1] F T T T F F T
```

This can be used in a variety of ways. For example, you can subset vectors based on the values in the vector or in another vector.

```
> x = c(3,8,6,7,3,4,9)
> x[x>5]
[1] 8 6 7 9
```

```
> y = c(0,1,1,0,1,0,0)
> x[y==1] # the double = is necessary
[1] 8 6 3
```

You can count the number of values meeting some condition:

```
> length(x > 5)
[1] 7
```

A logical vector is treated as a 0/1 vector if it is used in numeric operations:

```
> (1:7)*(x>5)
[1] 0 2 3 4 0 0 7
```

The proportion of values meeting a condition can therefore be calculated by

```
> mean(x > 5)
[1] 0.5714286
```

The commands **any()** and **all()** can be used on logical vectors: **any(x)** is T if any of the values in x is T and **all(x)** is true if all of the values in x are true:

```
> x = c(1,3,6,2)
> any(x>5)
[1] T
> all(x>5)
[1] F
```

8. Matrices

Matrices can be created using the **matrix** command or as the result of operations on vectors.

```
> x = matrix(1:12,nrow=3) # Creates a 3 by 4 matrix from the vector (1,2,...,12). Matrix is filled column by
column. Use option byrow=T to fill row by row.
> x
     [,1] [,2] [,3] [,4]
[1,]  1   4   7  10
[2,]  2   5   8  11
[3,]  3   6   9  12
```

You can also paste vectors together into a matrix using “cbind” (column bind) or “rbind” (row bind)

```
> cbind(1:3,4:6)
     [,1] [,2]
[1,]  1   4
[2,]  2   5
[3,]  3   6
> rbind(1:3,4:6)
     [,1] [,2] [,3]
[1,]  1   2   3
[2,]  4   5   6
```

```
> x[2,3] # Element in 2nd row, 3rd column
> x[2,]  # Whole 2nd row of x
> x[,c(1,3)] # Matrix consisting of all rows and 1st and 3rd columns of x
> t(x)      # Transpose of x
> x %*% t(x) # Matrix multiplication: multiply x by its transpose.
> x %*% c(1,3,5,7) # Multiply x by the vector (1,3,5,7)
> z = diag(c(2,4,-1)) # create a diagonal matrix(3 x 3) with entries 2, 4, -1 on the diagonal
```

The following apply only if x is a square matrix

```
> solve(x) # Inverse of a square matrix
> det(x)   # determinant of x
> eigen(y) # eigenvalues and eigenvectors of x
```

A very useful command for matrices is **apply**. It applies a function to all rows or columns of a matrix.

> **apply(x,1,mean)** Find the mean of all rows of x. The result is a vector with as many elements as there are rows of x. The second argument is 1 or 2 with 1 indicating rows and 2 indicating columns. The third argument is the function. The function can be one you have written (see last page). R actually has built-in functions to compute column or row sums and means which are faster than using apply:

```
> colMeans(x)
> colSums(x)
> rowMeans(x)
> rowSums(x)
```

9. Probability distributions

R has many probability distributions built in (see Probability Distributions and Random Numbers in Help - Contents for a list). One can compute probabilities, quantiles, and generate random numbers from these distributions. All built-in distributions have four commands associated with them to do these different things. For example, for a uniform(0,1) distribution:

```
> dunif(x)      # value of p.d.f. f(x); if x is a vector, the result is a vector
> punif(x)      # d.f F(x) = Pr(X≤x); x can be a vector
> qunif(p)      # F-1(p) = the pth quantile of X for 0<p<1; p can be a vector
> runif(m)      # generate a vector of m random values from a uniform(0,1) distribution
```

You can specify the values of the parameters of the uniform distribution as extra arguments. The default is U(0,1); for U(a,b), add the arguments a and b to the functions above. For example:

```
> dunif(2.5,0,5)      # f(2.5) for U(0,5)
> qunif(c(.1,.5,.9),0,5)  # 10th, 50th and 90th percentiles of U(0,5)
> runif(100000,0,5)     # vector of 100000 random numbers from U(0,5)
```

For a discrete distribution, the “d” form of the command gives the p.f. $f(x)=Pr(X=x)$

Discrete distributions:

1. Binomial(n,p): **dbinom(x,n,p), pbinom(x,n,p), qbinom(q,n,p), rbinom(m,n,p)**

2. Hypergeometric(A,B,n): A is number of red balls in urn, B is number of black balls, n is number drawn
dhyper(x,A,B,n), phyper(x,A,B,n), qhyper(q,A,B,n), rhyper(m,A,B,n)

Example: dhyper(3,5,7,4) is probability of drawing 3 red balls in 4 draws without replacement from an urn with 7 red balls and 4 black balls

3. Poisson(lambda): **dpois(x,lambda), ppois(x,lambda), qpois(q,lambda), rpois(m,lambda)**

4. Geometric(p): **dgeom(x,p), pgeom(x,p), qgeom(q,p), rgeom(m,p)**

Note: the geometric distribution is defined as the number of failures until the first success in Bernoulli trials. So $x=0,1,2,\dots$

Example: dgeom(1,.5) gives .25 (failure followed by a success)

5. Negative binomial(r,p): **dnbinom(x,r,p), pnbinom(x,r,p), qnbinom(q,r,p), rnbinom(m,r,p)**

Note: negative binomial is defined as the number of failures until the r^{th} success in Bernoulli trials

Continuous distributions

1. Uniform(a,b) a=0,b=1 is default **dunif(x,a,b), punif(x,a,b), qunif(q,a,b), runif(m,a,b)**
2. Normal(0,1) (default) **dnorm(x), pnorm(x), qnorm(q), rnorm(m)**
Normal(mu,sigma): **dnorm(x,mu,sigma), pnorm(x,mu,sigma), qnorm(q,mu,sigma), rnorm(m,mu,sigma)**
3. Exponential(b), b=1 is default: **dexp(x,b), pexp(x,b), qexp(q,b), rexp(m,b)**
4. Gamma(a,b): **dgamma(x,a,b), pgamma(x,a,b), qgamma(q,a,b), rgamma(m,a,b)**
5. Beta(a,b): **dbeta(x,a,b), pbeta(x,a,b), qbeta(q,a,b), rbeta(m,a,b)**

10. Sampling with and without replacement from a vector

- > **sample(x)** # random permutation of the elements in the vector x
if x is a single integer, then it is equivalent to sample(1:x)
- > **sample(x,k)** # random sample of size k without replacement from x; k must be less than length(x)
- > **sample(x,k,rep=T)** # random sample with replacement from x; k can be as large as desired

Examples:

```
> sample(5)
[1] 4 3 1 2 5
> x = c(2,4,6,8)
> sample(x,5, rep=T)
[1] 6 6 6 8 4
```

You can also specify a vector of probabilities with which to sample. For example

```
> sample(c(0,1),6,prob=c(.2,.8),rep=T) # 6 Bernoulli trials with p = .8
[1] 1 0 1 1 0 1
```

11. Graphics commands

There are many graphics commands in R

```
plot(x,y) # scatterplot of the vector x vs. the vector y
plot(x,y,type="l") # connect consecutive points with lines (for plotting functions)
hist(x) # histogram
hist(x, freq=F) # histogram scaled to have area 1 (can be compared to a theoretical density)
```

There is an automatic way to plot a function. Here's an example of how to plot the function $y=\sin(x)$ from 0 to π .

```
> curve(sin(x),0,2*pi)
```

To put 4 plots on one page (to save space), issue this command before you do any plots:

```
> par(mfrow=c(2,2)) # puts plots in a 2 by 2 matrix on each page
```

You can add points or lines to a current plot using the **points** command or the **lines** command.

To plot a theoretical density function over a histogram:

```
hist(rexp(1000),freq=F) # histogram of a random sample from exponential density with beta = 1
curve(dexp(x),add=T) # adds curve to current plot
```

You can control many aspects of plots: line type, the limits on the x and y axes, labeling, etc. see `?plot` and `?par`.

You can save R graphs in a variety of formats or copy and paste them to a document by using **File...Save As** or **...Copy to Clipboard**.

12. Creating Your Own Functions

All the commands you use in R are functions. One of the most powerful features of R is the ability to create your own functions out of the built-in R functions. The easiest way is to type the definition of a function (see example below) in a Script window and then run the script. The function will then become a permanent object in your directory. The best way to learn about functions is to see the “programs” for other functions. Type the name of any function in R to see its program. While many R built-in functions are coded in C, others are built from more elementary R functions.

For example, the following function calculates the probability, in the Gambler’s Ruin problem, that a player starting with i dollars out of a total pool of k dollars and a probability of winning each trial of p , will win the game (win all k dollars before the other player).

```
gruin = function(i,k,p)
{
q = 1-p
((q/p)^i -1)/((q/p)^k -1)
}
```

```
> gruin(90,100,.45)
[1] 0.1344306
```

We could add checks to make sure that i is less than k , p is between 0 and 1, etc.

All R commands are available to use in your function. You can also store intermediate results (e.g., $a = \text{var}(x)$); all variable names are local to the function so you don’t need to worry about duplicating names which you already used in your R session.

For more help on writing functions, see the pdf manual **Help...Online Manuals...Programmer’s Guide**.

13. Loops

Loops in R have either the simple structure:

```
for(i in 1:10) print (sum(1:i))          # an explicit print command is necessary in a loop
(The same result is more easily achieved by the built-in function cumsum(1:10) ).
```

For multiple commands, the loop has the structure:

```
for(i in 1:10) {
# commands
...
}
```

There are other looping commands like **while** and **repeat**, **break**. See the R documentation for the use of

these.

Explicit loops should be avoided if possible by using R's built-in vector and matrix operations and commands like **apply** (see Sec. 8 on matrices).

14. Runs and pattern matching

The function **rle** computes information on all runs in a vector:

```
> x = c(0,0,1,1,1,0,1,0,1,1,1)
```

```
> y = rle(x)
```

```
> y
```

```
$lengths:
```

```
[1] 2 3 1 1 1 3
```

```
$values:
```

```
[1] 0 1 0 1 0 1
```

To get the number of runs and the length of the longest run:

```
> length(y$lengths)
```

```
[1] 6
```

```
> max(y$lengths)
```

```
[1] 3
```

```
> max(y$lengths[y$values == 0])      # length of longest run of 0's
```

```
[1] 2
```

The function **grep** is useful for pattern matching in strings.

```
> x = c("123", "132", "213", "231", "312", "321")
```

```
> grep("23", x)
```

```
[1] 1 4      #This means that elements 1 and 4 of x contain the string "12".
```

Other functions useful for strings include **substring**, **paste**, **nchar**, **gsub**.

14. A list of some commands:

Miscellaneous:

args(fun)	where "fun" is the name of a function, either a built-in Splus function or one you've created: this lists the arguments of the function
seq(0,10,.1)	Generates the vector of numbers from 0 to 10 in steps of .1: 0,.1,.2,.3,...9.9,10.0
seq(0,10,length=50)	Generates the vector of 50 evenly-spaced numbers running from 0 to 10
3:20	Generates the vector of integers from 3 through 20
abs(x)	absolute value
log(x)	natural log
log10(x)	log to base 10
exp(x)	e to the power x
sqrt(x)	
gamma(x)	gamma function
lgamma(x)	log-gamma function
sin(x)	
cos(x)	
tan(x)	
asin(x)	
acos(x)	
atan(x)	
rev(x)	reverse the order of the elements in the vector x
sort(x)	sort vector x from smallest to largest
round(x,3)	# rounds the values in x to 3 decimal places
ceiling(x)	# smallest integer bigger than or equal to x
floor(x)	# largest integer smaller than or equal to x

Descriptive statistics for a data vector x:

length(x)	Number of elements in the vector x
sum(x)	
cumsum(x)	Cumulative sums of vector x
cumprod(x)	Cumulative products of vector x
cummax(x)	
cummin(x)	
table(x)	table of frequencies of values in x
mean(x)	
var(x)	
stdev(x)	
median(x)	
min(x)	
max(x)	
unique(x)	values of x with repeats eliminated
quantile(x,q)	q th quantile of x; q can be a vector
quantile(x)	5-number summary consisting of min, Q ₁ , Q ₂ , Q ₃ , max
cor(x,y)	Correlation between vectors x and y (must be of same length)
lsfit(x,y)	Least squares fit of y to x

Combinations and permutations

factorial(n)	
choose(n, k, order.matters=F)	# default is combinations. If order.matters=T then permutations, e.g.

choose(10,4) # number of combinations of 10 things taken 4 at a time
choose(10,4,T) # number of permutations of 10 things taken 4 at a time
choose.multinomial(n, m) # multinomial coefficient where m is a vector

Runs and pattern matching

The function **rle** computes information on all runs in a vector:

```
> x = c(0,0,1,1,1,0,1,0,1,1,1)
```

```
> y = rle(x)
```

```
> y
```

```
$lengths:
```

```
[1] 2 3 1 1 1 3
```

```
$values:
```

```
[1] 0 1 0 1 0 1
```

The function **grep** is useful for pattern matching in strings.

```
> x = c("123","132","213","231","312","321")
```

```
> grep("23",x)
```

```
[1] 1 4 #This means that elements 1 and 4 of x contain the string "12". grep has a variety of options;  
see help in R.
```

Other functions useful for strings include

substring("abcde",2,4) # extracts the substring between positions 2 and 4

paste("ab","cd","e",sep=",") # results in one string: "ab,cd,e"; use **sep=""** to get "abcde"

nchar number of characters in a string

as.numeric("122.3") # converts string to number (works on vectors of strings, also)

Graphics

plot(x,y) Scatterplot of the vector x vs. the vector y

plot(x,y,type="l") Scatterplot with lines connecting the points

lines(x,y) add line segments connecting (x[i],y[i]) to (x[i+1],y[i+1]) for i=1,2,...,n-1, to current plot
x,y must be vectors of the same length:

points(x,y) add points to current plot; x,y must be vectors of the same length:

hist(x) Histogram

boxplot(x) Boxplot

boxplot(x,y) Side-by-side boxplots of x and y