R Tutorial

Hands-on course on High-Throughput Sequencing data analysis

December 1, 2014

Based on "An Introduction to R" (W. N. Venables, D. M. Smith and the R Core Team)

1 Introduction

R is a free software programming language and software environment for statistical computing and graphics. R is freely available and it compiles and runs on a wide variety of UNIX platforms, Windows and MacOS. You can download the complete source code, the precompiled binary distribution, the contributed extensions and documentation for R from the "Comprehensive R Archive Network" (CRAN): http://www.cran.r-project.org.
If you are using R under UNIX you can change your current directory to the "data" folder in your home directory with the cd command. Then start the R program with the command `R`. To use R under Windows launch the program by double clicking on the icon. At this point R commands may be issued.
To call a command you type the name followed by the arguments in parentheses. This is because in R you call functions to achieve results. If the

function takes no arguments you just type the name followed parenthesis. To quit the R program type:

```
> q()
```

at the command prompt.

R has an inbuilt help facility similar to the man facility of UNIX. To get more information on any specific named function the command is `help()`. The function name is passed as the argument. R is case sensitive as are most UNIX based packages, so "A" and "a" are different symbols and would refer to different variables. If you don't remember the exact name of a function you can search the R Help files by typing `help.search()` with a quoted search term inside the brackets.

Activity 1:

Look at the help for the function mean(). Determine whether a standard deviation function exists. Describe its syntax.

1.1 Extensions and packages

The current R is the result of a collaborative effort with contributions from all over the world. There are thousands of contributed packages for R, written by many different authors. Some of these packages implement specialized statistical methods, time series, microarrays, etc.

Most packages are available for download from CRAN (http://CRAN.R-project.org/ and its mirrors) and other repositories such as Bioconductor (http://www.bioconductor.org/). Currently, the CRAN package repository features 6120 available packages. Bioconductor is an open source, open development software project to provide tools for the analysis and comprehension of high-throughput genomic data. It is based primarily on the R programming language.

The broad goals of the Bioconductor project are:

- To provide widespread access to a broad range of powerful statistical and graphical methods for the analysis of genomic data.
- To facilitate the inclusion of biological metadata from web databases in the analysis of genomic data.
- To provide a common software platform that enables the rapid development and deployment of extensible, scalable, and interoperable software.

- To further scientific understanding by producing high-quality documentation and reproducible research.
- To train researchers on computational and statistical methods for the analysis of genomic data.

2 Simple manipulations: numbers and vectors

R operates on named data structures. The simplest such structure is the numeric vector, which is a single entity consisting of an ordered collection of numbers. To set up a variable use the assignment operator "$< -$" which represents an arrow:

```
> a <- 5
```

To set up a vector use the function c() that concatenates its arguments:

```
> x <- c(3.5, 4, 0.6, 8, 12)
```

It also works in the other direction "$- >$":

```
> c(3.5, 4, 0.6, 8, 12) -> z
```

Assignment can also be made using the function assign().
Activity 2:
Get help on the assign function syntax and use it to assign c(3.5, 4, 0.6, 8, 12) to a vector named "y".
Already existing objects can be used to create new ones:

```
> w <- c(x, 4, z, y/3)
```

R has all of the standard arithmetic operators, so you can use R as a calculator. Vectors can be used in arithmetic expressions:

```
> u1<-4*3+15
> u2 <- 4*x + y
```

In addition, all of the common artihmetic functions such as `log()`, `sin()`, `cos()`, `exp()`, `tan()` are available.
To display the content of an object, one simply types the name.
Activity 3:
Display the content of vector w.
What's the result of x+y?
Diverse functions useful in statistics are also available (`range(), mean(), length(), sum()`).
Activity 4:
Calculate the variance of vector w in one line of code.
To sort the elements of x in increasing order use sort().

Activity 5:
Use `sort()` function to sort the elements of w. Can you sort it in decreasing order?

A regular sequence of integers, for example from 1 to 100, can be generated with:

```
> hundred<-c(1,2,3,...,99,100)
```

However, R has a number of facilities for generating commonly used sequences of numbers:

```
> hundred<-1:100
```

The operator ":" has high priority within an expression.

Activity 6:
Compare the resut of the expressions $1 : n-1$ and $1 : (n-1)$, for any natural number n.
Generate a sequence of numbers from 20 to 1.

The function `seq()` is a more general facility for generating sequences. It has five arguments, only some of which may be specified in any one call. It is possible to specify the beginning, end, step size and/or the length of the sequence. R assumes that you give the arguments in exactly the same order as they're shown in the usage line of the Help page for that function. For example, to generate a sequence from 0 to 1 and step size 0.1:

```
> seq(0, 1, 0.1) or
> seq(to= 1, from= 0, by= 0.1) or
> seq(length= 11, from= 0, by= 0.1)
```

Subsets of the elements of a vector may be selected by appending to the name of the vector an index vector in square brackets. The values in the index vector must lie between 1 and $n = length(vector)$. For example, to obtain the first component of y:

```
> y[1]
```

Remember that the operator ":" is a handy way of creating sequences. To get the third, fourth and fifth element of y:

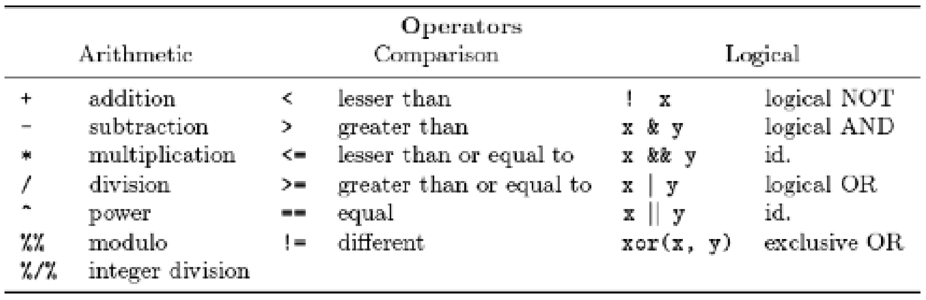
```
> y[3:5]
```

The index vector could also be a vector of character strings (this possibility only applies where an object has a names attribute or label to identify its components):

```
> z <- 1:9
> names(z) <- c(``a'',``b'',``c'',``d'',``e'',``f'',``g'',``h'',``i'')
> z
> z[``a'']
```

3 Operators

There are three main basic types of operators: arithmetic operators, comparison operators and logical operators.



Operators					
Arithmetic		Comparison		Logical	
`+`	addition	`<`	lesser than	`! x`	logical NOT
`-`	subtraction	`>`	greater than	`x & y`	logical AND
`*`	multiplication	`<=`	lesser than or equal to	`x && y`	id.
`/`	division	`>=`	greater than or equal to	`x | y`	logical OR
`^`	power	`==`	equal	`x || y`	id.
`%%`	modulo	`!=`	different	`xor(x, y)`	exclusive OR
`%/%`	integer division				

If c1 and c2 are logical expressions, then c1 & c2 is their intersection ("and"), c1 | c2 is their union ("or"), and !c1 is the negation of c1. Logical vectors may be used in ordinary arithmetic, in which case they are coerced into numeric vectors, FALSE becoming 0 and TRUE becoming 1.
An indexed expression can also appear on the receiving end of an assignment:

```
> y[y>2]
```

The expression displays the values in y greater than 2.
Activity 7:
Create a vector `m` containing the elements in vector `w` greater than 2 and less than 5.
What happens if you multiply `m*m`?

4 Missing values

In some cases the components of a vector may not be completely known. When an element or value is "not available" or a "missing value" in the statistical sense, a place within a vector may be reserved for it by assigning it the special value NA. In general any operation on an NA becomes an NA. The function `is.na(x)` gives a logical vector of the same size as x with value TRUE if and only if the corresponding element in x is NA.

Note that there is a second kind of "missing" values which are produced by numerical computation, the so-called Not a Number, NaN, values. NaN represents an undefined or unrepresentable value, such as the result of 0/0 or $Inf - Inf$. The function `is.nan(x)` returns TRUE is the value is NaN.

5 Objects

The entities R operates on are technically known as objects. Examples are vectors of numeric (real) or complex values, vectors of logical values and vectors of character strings. These are known as "atomic" structures since their components are all of the same type, or mode, namely numeric, complex, logical, character and raw. R also operates on objects called lists, which are of mode list. These are ordered sequences of objects which individually can be of any mode. Lists are known as "recursive" rather than atomic structures since their components can themselves be lists in their own right. By the mode of an object we mean the basic type of its fundamental constituents. This is a special case of a "property" of an object. Another property of every object is its length. The functions `mode()` and `length()` can be used to find out the mode and length of any defined structure
Constructing a list example:

```
> lst<-list()
> lst[[1]]<-‘‘A’’
> lst[[2]]<-3
> lst[[3]]<-rnorm(1)
```

The double square brackets operator is used to add a single element to the list. The `rnorm()` function can generate random numbers whose distribution is normal.
Activity 8:
Check the mode and length attribute of some of the vectors created.
Check the attributes of the lst list.
It is possible to convert an object from a type to another by changing some of its attributes. For example:

```
> char<-as.character(m)
> num<-as.numeric(char)
```

`ls()` and `ls.str()` can be used to display the names of the objects which are currently stored within R.
Activity 9:
In which situation would you use the argument pattern to `ls()`?

List all the names of the objects in the current environment.
What's the result of `ls(pattern=` “$^\wedge u$”`)?`
Description of the data structures defined in R:

object	modes	several modes possible in the same object?
vector	numeric, character, complex *or* logical	No
factor	numeric *or* character	No
array	numeric, character, complex *or* logical	No
matrix	numeric, character, complex *or* logical	No
data frame	numeric, character, complex *or* logical	Yes
ts	numeric, character, complex *or* logical	No
list	numeric, character, complex, logical, function, expression, …	Yes

6 Arrays and matrices

An array can be considered as a multiply subscripted collection of data entries. A dimension vector is a vector of non-negative integers. A vector can be used by R as an array only if it has a dimension vector as its dim attribute:

```
> n<-vector(‘‘numeric’’,1500)
> dim(n)<-c(5,3,100)
```

Other functions such as matrix() and array() are available for simpler and more natural looking assignments.

```
> t<-array(1:20, dim=c(4,5))
> t[1,1]<-9
> t[2,2]<-6
> t[3,3]<-7
> t[4,4]<-25
```

For example, we could create a matrix:

```
> o<-matrix(1:3,5,3)
```

The entire second row of matrix “o” be extracted as:

```
o[2, ]
```

Similarly, the entire third column of matrix “o” can be extracted as:

```
o[,3 ]
```

An array is a table with k dimensions, a matrix being a particular case of array with k = 2. R has facilities for matrix computation and manipulation, You can use `solve()` for inverting a matrix, `t()` for transposition, or the operator `%*%` to calculate the product:

```
> s<-t(t)
> t2<-s%*%t
```

Activity 10:
Create a 2×2 matrix named "xx" containing the elements $[1, 2, 3, 4]$. Create a matrix "yy" containing the elements "xx" multiplied by 3.
Create a matrix named "mat" from vectors `vec1<-c(3,5,1,4,7)` and `vec2<-c(4,6,7,9,11)`, and 3 columns of matrix t2 (columns from 3 to 5) using the function `cbind()`. Calculate the determinant of "mat". Create a matrix "mat2" from matrix "mat" substracting 3 from element (3,2) and adding 2 to element (5,4).
For matrices and data frames, `colnames()` and `rownames()` are labels of the columns and rows, respectively. They can be accessed either with their respective functions, or with `dimnames()` which returns a list with both vectors. A data frame may for many purposes be regarded as a matrix with columns possibly of differing modes and attributes. It may be displayed in matrix form, and its rows and columns extracted using matrix indexing conventions.

7 Conditional statements and loops

Commands may be grouped together in braces "{}". For example in {expr_1; expr_2; expr_3;...; expr_m} the value of the group is the result of the last expression in the group evaluated. The language has available a conditional construction of the form:

```
> if (expr_1) expr_2 else expr_3,
```

where expr 1 must evaluate to a single logical value and the result of the entire expression is then evident. It is possible to replace expr_2 and expr_3 with grouped expressions "{}".
There is also a "for" loop construction which has the form.

```
> for (name in expr_1) expr_2
```

where "name" is the loop variable. expr 1 is a vector expression, (often a sequence like 1:10), and expr 2 is often a grouped expression with its sub-expressions written in terms of the dummy name.

Suppose we have a numerical vector y:

```
> y<-c(3,2,0,6,1,0)
```

and for each element of y we want to evaluate if it is equal to 0. If so, replace the element with 10:

```
> for(i in 1:length(y)){
    if(y[i]==0){
        y[i]<-10
    }
}
```

Activity 11:
Given the vector:
`vector<-c(4,5,9,2,1,4)`,
replace even elements with "−1".

8 Download and install packages

To install a CRAN package in R, use the `install.packages(''package name'')` function. Examine the arguments of the function.
If the package you want to install is included in Bioconductor:

```
> source(''http://bioconductor.org/biocLite.R'')
> biocLite(''package name'')
```

In case you want to install all the packages:

```
> source(''http://bioconductor.org/biocLite.R'')
> biocLite()
```

To load a particular package in the current session use:
`library(package name)`.
Activity 12:
Install and load the package `limma` in the current session.

9 Reading data from files

Large data objects will usually be read as values from external files rather than entered during an R session at the keyboard. R input facilities are simple and their requirements are fairly strict and even rather inflexible. There is a clear presumption by the designers of R that you will be able to modify your input. R has many functions that allow you to import data

from other applications. `read.table()` reads any tabular data where the columns are tab separated (you can specify the separator). `read.csv()` is a simplified version of `read.table()`, where the arguments are preset to read CSV files (Excel spreadsheets).
To display the current working directory, type `getwd()`. Be sure to enter the working directory as a character string (enclose it in quotes). To get a list of files in a specific folder, use `list.files()`. The default corresponds to the working directory.
Activity 13:
Try reading the file "prokaryote_pathogenicity.csv" using `read.csv()`. This file contains bacterial phenotypic data. If you get an error take a look at the file and at the function parameters. Remember you can select a different directory with the function `setwd()`.
The `scan()` function allows you finer control over the read process. For example:

```
> mydata <- scan(``data.dat'', what = list(``'', 0, 0))
```

imports the content of file "datos.dat" into "mydata" object, specifying the first column of mode character and the remaining two of mode numeric.
Now try to read the file "human.aa". This table contains the amino acid usage in the human genome (gene names and amino acid frequences):

```
> tab<-read.table(``human.aa'',header=TRUE)
> dim(tab)
> colnames(tab)
```

Variables can be extracted using the "$" operator followed by the name of the variable. To get more information use help("$"). To obtain the frequences of alanine:

```
> tab$A
```

or we can also obtain these frequences if we know the column number:

```
> tab[ ,3]
```

Use the function `all.equal()` to check if both expressions are equivalent:

```
> all.equal(tab$A,tab[,3]) ### TRUE
```

The # symbol means that everything that follows is only comment.
Now we want to add a column to the matrix, corresponding to the sum of aminoacids per gene. Add up all the values in each row from 3 to 22 and assign the result to the object using the function `apply()`. You can use the `apply()` fuction to apply a function over every row or column of a matrix:

```
> tab$aasum<-apply(tab[,3:22],MARGIN=1,FUN=sum)
> colnames(tab)
```

You could construct a for loop to do so, but using `apply()`, you do this in only one line of code.

Activity 14:
Take a look at the help for the functions `apply()`, `lapply()` and `sapply()`. What is the difference between them?
We want to create a table consisting of the relative frequencies of each aminoacid (frequency divided by the sum of aminoacids):

```
> tabrel<-tab[,3:22]/tab$aasum
```

To change the column names:

```
> colnames(tabrel)<-paste(colnames(tabrel),‘‘rel’’,sep=‘‘’’)
> colnames(tabrel)
```

Activity 15:
Use function `cbind()` to join objects “tab” and “tabrel” into object “absrel”. The function `write.table()` writes in a file an object. Save object “absrel” to file “V.txt”.
What is the function of the argument “append”?

10 Plots

R offers a remarkable variety of graphics. It is not possible to detail here the possibilities of R in terms of graphics, particularly since each graphical function has a large number of options making the production of graphics very flexible. The way graphical functions work deviates substantially from the scheme sketched at the beginning of this document. Particularly, the result of a graphical function cannot be assigned to an object but is sent to a graphical device. A graphical device is a graphical window or a file.
The simple way to create a 2-D plot of a function is to use the `plot()` function. Let’s take a look at this example:

```
> meters<-c(0,10,20,30,40,50,60,70,80,90,100)
> speed<-c(0,1.8,3.1,4.2,5.9,6.7,7.6,8.3,9.1,9.8,10.9)
> plot(meters,speed)
```

You can create histograms with the function `hist(x)` where x is a numeric vector of values to be plotted.
Activity 16:
Investigate and test the `plot()` function arguments pch, type y main. You can create a file using `pdf("filename.pdf")` function. Then use `plot()` function to plot to pdf. Finally type `dev.off()` command to tell R that you are finished plotting