An Introduction to R

Biostatistics 615/815

Last Week
An Introduction to C

- Strongly typed language
  - Variable and function types set explicitly
- Functional language
  - Programs are a collection of functions
- Rich set of program control options
  - `for, while, do …while, if` statements
- Compiling and debugging C programs

Homework Notes

- Due on Wednesday (by end of the day)
  - Dr. Abecasis' Departmental Mailbox
  - Provide hard copy
- Write specific answer to each question
  - Text, supported by table or graph if appropriate
- Source code
  - Indented and commented, if appropriate

This Week

- The R programming language
  - Syntax and constructs
  - Variable initializations
  - Function declarations

- Introduction to R Graphics Functionality
  - Some useful functions

The R Project

- Environment for statistical computing and graphics
  - Free software
- Associated with simple programming language
  - Similar to S and S-plus
- www.r-project.org

The R Project

- Versions of R exist of Windows, MacOS, Linux and various other Unix flavors
- R was originally written by Ross Ihaka and Robert Gentleman, at the University of Auckland
- It is an implementation of the S language, which was principally developed by John Chambers

On the shoulders of giants…

- In 1998, the Association for Computing Machinery gave John Chambers its Software Award. His citation reads:

- *“S has forever altered the way people analyze, visualize, and manipulate data … It is an elegant, widely accepted, and enduring software system, with conceptual integrity.”*

Compiled C vs Interpreted R

- C requires a complete program to run
  - Program is translated into machine code
  - Can then be executed repeatedly

- R can run interactively
  - Statements converted to machine instructions as they are encountered
  - This is much more flexible, but also slower

R Function Libraries

- Implement many common statistical procedures
- Provide excellent graphics functionality
- A convenient starting point for many data analysis projects

R Programming Language

- Interpreted language
- To start, we will review
  - Syntax and common constructs
  - Function definitions
  - Commonly used functions

Interactive R

- R defaults to an interactive mode
- A prompt “>” is presented to users
- Each input expression is evaluated…
- … and a result returned

R as a Calculator

```
> 1 + 1         # Simple Arithmetic
[1] 2
> 2 + 3 * 4     # Operator precedence
[1] 14
> 3 ^ 2         # Exponentiation
[1] 9
> exp(1)        # Basic mathematical functions are available
[1] 2.718282
> sqrt(10)
[1] 3.162278
> pi            # The constant pi is predefined
[1] 3.141593
> 2*pi*6378     # Circumference of earth at equator (in km)
[1] 40074.16
```

Variables in R

- Numeric
  - Store floating point values
- Boolean (T or F)
  - Values corresponding to True or False
- Strings
  - Sequences of characters
- Type determined automatically when variable is created with "<-" operator

R as a Smart Calculator

```
> x <- 1                # Can define variables
> y <- 3                # using "<-" operator to set values
> z <- 4
> x * y * z
[1] 12

> X * Y * Z             # Variable names are case sensitive
Error: Object "X" not found

> This.Year <- 2004     # Variable names can include period
> This.Year
[1] 2004
```

R does a lot more!

- Definitely not just a calculator
- R thrives on vectors
- R has many built-in statistical and graphing functions

R Vectors

- A series of numbers

- Created with
  - `c()` to concatenate elements or sub-vectors
  - `rep()` to repeat elements or patterns
  - `seq()` or `m:n` to generate sequences

- Most mathematical functions and operators can be applied to vectors
  - Without loops!

Defining Vectors

```
> rep(1,10)          # repeats the number 1, 10 times
[1] 1 1 1 1 1 1 1 1 1 1
> seq(2,6)           # sequence of integers between 2 and 6
[1] 2 3 4 5 6        # equivalent to 2:6
> seq(4,20,by=4)     # Every 4th integer between 4 and 20
[1]  4  8 12 16 20
> x <- c(2,0,0,4)    # Creates vector with elements 2,0,0,4
> y <- c(1,9,9,9)
> x + y              # Sums elements of two vectors
[1]  3  9  9 13
> x * 4              # Multiplies elements
[1]  8  0  0 16
> sqrt(x)                 # Function applies to each element
[1] 1.41 0.00 0.00 2.00 # Returns vector
```

Accessing Vector Elements

- Use the [ ] operator to select elements

- To select specific elements:
  - Use index or vector of indexes to identify them

- To exclude specific elements:
  - Negate index or vector of indexes

- Alternative:
  - Use vector of T and F values to select subset of elements

Accessing Vector Elements

```
> x <- c(2,0,0,4)
> x[1]        # Select the first element, equivalent to x[c(1)]
[1] 2
> x[-1]       # Exclude the first element
[1] 0 0 4
> x[1] <- 3 ; x
[1] 3 0 0 4
> x[-1] = 5 ; x
[1] 3 5 5 5
> y < 9       # Compares each element, returns result as vector
[1]  TRUE FALSE FALSE FALSE
> y[4] = 1
> y < 9
[1]  TRUE FALSE FALSE  TRUE
> y[y<9] = 2  # Edits elements marked as TRUE in index vector
> y
[1] 2 9 9 2
```

Data Frames

- Group a collection of related vectors
- Most of the time, when data is loaded, it will be organized in a data frame
- Let’s look at an example …

Setting Up Data Sets

- Load from a text file using `read.table()`
  - Parameters `header, sep, and na.strings` control useful options
  - `read.csv()` and `read.delim()` have useful defaults for comma or tab delimited files
- Create from scratch using `data.frame()`
  - Example:

```
data.frame(height=c(150,160),
           weight=(65,72))
```

Blood Pressure Data Set

HEIGHT	WEIGHT	WAIST	HIP	BPSYS	BPDIA
172	72	87	94	127.5	80
166	91	109	107	172.5	100
174	80	95	101	123	64
176	79	93	100	117	76
166	55	70	94	100	60
163	76	96	99	160	87.5

...

Read into R using:

```
bp <-
  read.table("bp.txt",header=T,na.strings=c("x"))
```

Accessing Data Frames

- Multiple ways to retrieve columns…
- The following all retrieve weight data:
  - `bp["WEIGHT"]`
  - `bp[,2]`
  - `bp$WEIGHT`
- The following excludes weight data:
  - `bp[,-2]`

Lists

- Collections of related variables
- Similar to records in C
- Created with list function
  - `point <- list(x = 1, y = 1)`
- Access to components follows similar rules as for data frames, the following all retrieve x:
  - `point$x; point["x"]; point[1]; point[-2]`

So Far …
Common Forms of Data in R

- Variables are created as needed

- Numeric values
- Vectors
- Data Frames
- Lists

- Used some simple functions:
  - `c(), seq(), read.table(), …`

Next …

- More detail on the R language, with a focus on managing code execution
  - Grouping expressions
  - Controlling loops

Programming Constructs

- Grouped Expressions
- Control statements
  - `if … else …`

  - `for` loops
  - `repeat` loops
  - `while` loops

  - `next, break` statements

Grouped Expressions

```
{expr_1; expr_2; … }
```

- Valid wherever single expression could be used
- Return the result of last expression evaluated
- Relatively similar to compound statements in C

if … else …

if (expr_1**)** expr_2 **else** expr_3

- The first expression should return a single logical value
  - Operators && or | | may be used
- Conditional execution of code

Example: if … else …

```
# Standardize observation i
if (sex[i] == “male”)
  {
   z[i] <- (observed[i] –
  males.mean) / males.sd;
   }
else
   {
   z[i] <- (observed[i] –
```

for

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

- Name is the loop variable
- `expr_1` is often a sequence
  - e.g. `1:20`
  - e.g. `seq(1, 20, by = 2)`

Example: for

```
# Sample M random pairings in a set of N objects
for (i in 1:M)
   {
   # As shown, the sample function returns a
   single
    # element in the interval 1:N
   p = sample(N, 1)
   q = sample(N, 1)

   # Additional processing as needed…
  ProcessPair(p, q);
   }
```

repeat

```
repeat expr
```

- Continually evaluate expression
- Loop must be terminated with `break` statement

Example: repeat

```
# Sample with replacement from a set of N objects
# until the number 615 is sampled twice
M <- matches <- 0
repeat
   {
   # Keep track of total connections sampled
   M <- M + 1

   # Sample a new connection
   p = sample(N, 1)

   # Increment matches whenever we sample 615
   if (p == 615)
      matches <- matches + 1;

   # Stop after 2 matches
   if (matches == 2)
      break;
   }
```

while

while (expr_1**)** expr_2

- While `expr_1` is false, repeatedly evaluate `expr_2`
- `break` and `next` statements can be used within the loop

Example: while

```
# Sample with replacement from a set of N objects
# until 615 and 815 are sampled consecutively
match <- false
while (match == false)
   {
   # sample a new element
   p = sample(N, 1)

   # if not 615, then goto next iteration
   if (p != 615)
      next;

   # Sample another element
   q = sample(N, 1)

   # Check if we are done
   if (q != 815)
      match = true;
   }
```

Functions in R

- Easy to create your own functions in R
- As tasks become complex, it is a good idea to organize code into functions that perform defined tasks
- In R, it is good practice to give default values to function arguments

Function definitions

```
name <- function(arg1, arg2, …)
            expression
```

- Arguments can be assigned default values:

  `arg_name = expression`

- Return value is the last evaluated expression or can be set explicitly with `return()`

Defining Functions

```
> square <- function(x = 10) x * x
> square()
[1] 100
> square(2)
[1] 4

> intsum <- function(from=1, to=10)
    {
    sum <- 0
    for (i in from:to)
      sum <- sum + i
    sum
    }
> intsum(3)          # Evaluates sum from 3 to 10 …
[1] 52
> intsum(to = 3)     # Evaluates sum from 1 to 3 …
[1] 6
```

Some notes on functions ...

- You can print the arguments for a function using `args()` command

```
> args(intsum)
function (from = 1, to = 10)
```

- You can print the contents of a function by typing only its name, without the `( )`
- You can edit a function using

```
> my.func <- edit(my.old.func)
```

Debugging Functions

- Toggle debugging for a function with `debug()`/`undebug()` command

- With debugging enabled, R steps through function line by line
  - Use `print()` to inspect variables along the way
  - Press `<enter>` to proceed to next line

```
> debug(intsum)
> intsum(10)
```

So far …

- Different types of variables
  - Numbers, Vectors, Data Frames, Lists
- Control program execution
  - Grouping expressions with `{}`
  - Controlling loop execution
- Create functions and edit functions
  - Set argument names
  - Set default argument values

Useful R Functions

- Online Help
- Random Generation
- Input / Output
- Data Summaries
- Exiting R

Random Generation in R

- In contrast to many C implementations, R generates pretty good random numbers
- `set.seed(seed)` can be used to select a specific sequence of random numbers
- `sample(x, size, replace = FALSE)` generates a sample of size elements from `x`.
  - If `x` is a single number, sample is from `1:x`

Random Generation

- `runif(n, min = 1, max = 1)`
  - Samples from Uniform distribution
- `rbinom(n, size, prob)`
  - Samples from Binomial distribution
- `rnorm(n, mean = 0, sd = 1)`
  - Samples from Normal distribution
- `rexp(n, rate = 1)`
  - Samples from Exponential distribution
- `rt(n, df)`
  - Samples from T-distribution
- And others!

R Help System

- R has a built-in help system with useful information and examples

- `help()` provides general help
- `help(plot)` will explain the plot function
- `help.search("histogram")` will search for topics that include the word histogram

- `example(plot)` will provide examples for the plot function

Input / Output

- Use `sink(file)` to redirect output to a file
- Use `sink()` to restore screen output

- Use `print()` or `cat()` to generate output inside functions

- Use `source(file)` to read input from a file

Basic Utility Functions

- `length()` returns the number of elements
- `mean()` returns the sample mean
- `median()` returns the sample mean
- `range()` returns the largest and smallest values
- `unique()` removes duplicate elements
- `summary()` calculates descriptive statistics
- `diff()` takes difference between consecutive elements
- `rev()` reverses elements

Managing Workspaces

- As you generate functions and variables, these are added to your current workspace
- Use `ls()` to list workspace contents and `rm()` to delete variables or functions
- When you quit, with the `q()` function, you can save the current workspace for later use

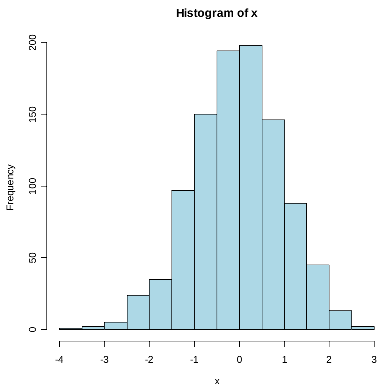
Summary of Today’s Lecture

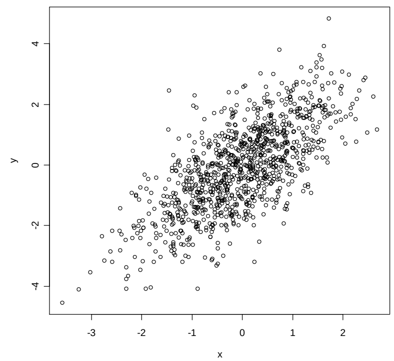
- Introduction to R

- Variables in R
- Basic Loop Syntax in R
- Functions in R

- Examples of useful built-in functions

Next Lecture…
Introduction to R Graphics






```
> x <- rnorm(1000)
> y <- rnorm(1000) + x
> summary(y)
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-4.54800 -1.11000 -0.06909 -0.09652  0.86200  4.83200
> var(y)
[1] 2.079305
> hist(x, col="lightblue")
> plot(x,y)
```

Learning More About R

- Excellent documentation is available at www.r-project.org
  - "An Introduction to R"
    by Venables and Smith
    in the Documentation Section
- Good book to browse is "Data Analysis and Graphics in R" by Maindonald and Braun

For your review

- Implementations of the three Union-Find algorithms (from Lecture 1) are provided in the next few pages…

Example: Quick Find Function

```
QuickFind <- function( N = 100, M = 100)
  {
  a <- seq(1, N)              # initialize array

  for (dummy in seq(1,M))     # for each connection
    {
    p <- sample(N, 1)         # sample random objects
    q <- sample(N, 1)

    if (a[p] == a[q])         # check if connected
      next

    a[a == a[p]] <- a[q]      # update connectivity array
    }
  }
```

Example: Quick Union Function

```
QuickUnion <- function( N = 100, M = 100)
  {
  a <- seq(1, N)                     # initialize array

  for (dummy in seq(1,M))            # for each connection
     {
     p <- sample(N, 1)               # sample random objects
     q <- sample(N, 1)

     # check if connected
     i = a[p]; while (a[i] != i) i <- a[i]
     j = a[q]; while (a[j] != j) j <- a[j]

     if (i == j)
        next

     a[i] = j                        # update connectivity array
     }
  }
```

Example: Weighted Quick Union

```
WeightedQuickUnion <- function( N = 100, M = 100)
  {
  a <- seq(1, N)                          # initialize arrays
  weight <- rep(1, N)

  for (dummy in seq(1,M))                 # for each connection
     {
     p <- sample(N, 1)                    # sample random objects
     q <- sample(N, 1)

     i = a[p]; while (a[i] != i) i <- a[i]    # FIND
     j = a[q]; while (a[j] != j) j <- a[j]

     if (i == j) next

     if (weight[i] < weight[j])               # UNION
        { a[i] = j; weight[j] <- weight[j] + weight[i]; }
     else
        { a[j] = i; weight[i] <- weight[i] + weight[j]; }
     }
  }
```

Benchmarking a function

- To conduct empirical studies of a functions performance, we don’t always need a stopwatch.

- Relevant functions
  - `Sys.time()` gives current time
  - `difftime(stop, start)` difference between two times

Example: Slower Quick Find...

```
QuickFind2 <- function( N = 100, M = 100)
  {
  a <- seq(1, N)          # initialize array

  for (dummy in seq(1,M)) # for each connection
     {
     p <- sample(N, 1)            # sample random objects
     q <- sample(N, 1)

     if (a[p] == a[q])            # check if connected
        next

     set <- a[p]                  # update connectivity array
     for (i in 1:N)
        if (a[i] == set)
            a[i] = a[q]
     }
  }
```

Example: Slower Quick Find...

```
> bench <- function(f, N = 100, M = 100)
   {
   cat(" N = ", N, ", M = ", M, "\n")

   start <- Sys.time()
   f(N, M)
   stop <- Sys.time()
   difftime(stop, start)
   }
> bench(QuickFind, 4000, 4000)
 N =  4000 , M =  4000
Time difference of 2 secs
> bench(QuickFind2, 4000, 4000)
 N =  4000 , M =  4000
Time difference of 1.066667 mins
```