# Basic Constructs in R

## R structures, and the apply class of functions

Spencer Lourens

January 09 2017

# R Background

- R is a Statistical Programming Language used for data management, statistical analysis, graphics, and general computation

# R Background

- R is a Statistical Programming Language used for data management, statistical analysis, graphics, and general computation
  - Implementation of S language, which was created by John Chambers at Bell Labs

# R Background

- R is a Statistical Programming Language used for data management, statistical analysis, graphics, and general computation
  - Implementation of S language, which was created by John Chambers at Bell Labs
  - R was created by Ross Ihaka and Robert Gentleman, at University of Auckland, New Zealand (Hadley Wickham!!)

# R Background

- R is a Statistical Programming Language used for data management, statistical analysis, graphics, and general computation
  - Implementation of S language, which was created by John Chambers at Bell Labs
  - R was created by Ross Ihaka and Robert Gentleman, at University of Auckland, New Zealand (Hadley Wickham!!)
  - Many standard functions in R are written in R, however, more computationally intensive tasks are coded in C++, C, or Fortran. R provides methods for linking these lower level languages into the R framework

# R Background

- R is a Statistical Programming Language used for data management, statistical analysis, graphics, and general computation
  - Implementation of S language, which was created by John Chambers at Bell Labs
  - R was created by Ross Ihaka and Robert Gentleman, at University of Auckland, New Zealand (Hadley Wickham!!)
  - Many standard functions in R are written in R, however, more computationally intensive tasks are coded in C++, C, or Fortran. R provides methods for linking these lower level languages into the R framework
  - R is easily extensible through the use of packages. The primary repository for R packages is CRAN, though there are also many packages geared towards bioinformatics on Bioconductor. There were 7,801 add-on packages available for R as of January 2016

# R Implementation

- R is an interpreted language, essentially meaning that you don't need to compile your programs in order to run them. The interpreter "interprets" your code and returns a result immediately

# R Implementation

- R is an interpreted language, essentially meaning that you don't need to compile your programs in order to run them. The interpreter "interprets" your code and returns a result immediately
- A compiler for R was recently introduced, which apparently can increase speed considerably in certain situations where you may run the same code over and over. (Think R packages, common analytical techniques, etc.)

# R Implementation

- R is an interpreted language, essentially meaning that you don't need to compile your programs in order to run them. The interpreter "interprets" your code and returns a result immediately
- A compiler for R was recently introduced, which apparently can increase speed considerably in certain situations where you may run the same code over and over. (Think R packages, common analytical techniques, etc.)
- I do not intend to focus on the implementation of R and how it "works", so let's dive into the functionality of R

# R Implementation

- R is an interpreted language, essentially meaning that you don't need to compile your programs in order to run them. The interpreter "interprets" your code and returns a result immediately
- A compiler for R was recently introduced, which apparently can increase speed considerably in certain situations where you may run the same code over and over. (Think R packages, common analytical techniques, etc.)
- I do not intend to focus on the implementation of R and how it "works", so let's dive into the functionality of R
- Things I plan to cover:
  - scalars, vectors, matrices, lists, data frames, functions, and the "apply" class of functions
  - The "apply" class of functions is very useful for iterating over a list or vector conducting the same operation on each entity in the list/vector

# R Data Structures: Scalars

- scalars - one value, for example: 3, 4.31, -10024

# R Data Structures: Scalars

- scalars - one value, for example: 3, 4.31, -10024

```
y1 <- 3
y2 <- 4.31
y3 <- -10024
```

# R Data Structures: Scalars

- scalars - one value, for example: 3, 4.31, -10024

```
y1 <- 3
y2 <- 4.31
y3 <- -10024
```

- The values of y1, y2, and y3 are all set accordingly

# R Data Structures: Vectors

- vectors - a number of values, for example: [3,4,1,2], [3.4, 4.16, 7.21, 8.67]

# R Data Strucures: Vectors

- vectors - a number of values, for example: [3,4,1,2], [3.4, 4.16, 7.21, 8.67]

```
v1 <- c(3, 4, 1, 2)
v2 <- c(3.4, 4.16, 7.21, 8.65)
```

# R Data Strucures: Vectors

- vectors - a one dimensional list of values, for example: [3,4,1,2], [3.4, 4.16, 7.21, 8.67]

```
v1 <- c(3, 4, 1, 2)
v2 <- c(3.4, 4.16, 7.21, 8.65)
```

- Now v1, v2 are assigned to the vectors specified above

v1: 3, 4, 1, 2, v2: 3.4, 4.16, 7.21, 8.65

# R Data Structures: Cont'd

- How do we access elements in a vector? We use the usual index notation, which starts at 1 in R. Some languages (C, C++, Fortran, python, javaScript, etc.) start at 0.

```
v1[1]
```

```
## [1] 3
```

```
v2[1]
```

```
## [1] 3.4
```

# R Data Structures: Vectorization

- R provides convenience in calculation via a process called vectorization

# R Data Structures: Vectorization

- R provides convenience in calculation via a process called vectorization
- For instance, if we have a vector $x$ and want to add 2 to all elements in the vector we use:

# R Data Structures: Recycling for vectors

- R provides convenience in calculation via a process called recycling. (Makes the smaller argument the right length/dimensions)
- For instance, if we want to add 2 to all elements in the vector v1 we use:

```
v1 + 2
```

```
## [1] 5 6 3 4
```

# R Data Structures: Recycling for vectors

- R provides convenience in calculation via a process called recycling. (Makes the smaller argument the right length/dimensions)
- For instance, if we want to add 2 to all elements in the vector v1 we use:

```
v1 + 2
```

```
## [1] 5 6 3 4
```

- Or adding two vectors together: (recycled if necessary)

```
v1 + v2
```

```
## [1]  6.40  8.16  8.21 10.65
```

# R Data Structures: Matrices

- matrices: a two dimensional list of values, for example: [3 4 1 2, 3.4 4.16 7.21 8.67], [1 0, 0 1]

# R Data Structures: Matrices

- matrices: a two dimensional list of values, for example: [3 4 1 2, 3.4 4.16 7.21 8.67], [1 0, 0 1]

```
m1 <- matrix(c(3, 4, 1, 2, 3.4, 4.16, 7.21, 8.67),
             nrow = 2, byrow = T)
m2 <- matrix(c(1, 0, 0, 1), nrow = 2, byrow = T)
```

# R Data Structures: Matrices

- matrices: a two dimensional list of values, for example: [3 4 1 2, 3.4 4.16 7.21 8.67], [1 0, 0 1]

```
m1 <- matrix(c(3, 4, 1, 2, 3.4, 4.16, 7.21, 8.67),
             nrow = 2, byrow = T)
m2 <- matrix(c(1, 0, 0, 1), nrow = 2, byrow = T)
```

- The argument byrow tells the matrix function whether to create the matrix row by row, from first index to last index. The argument nrow specified the number of rows.

**m1:**

| | | | |
|---|---|---|---|
| **3.0** | **4.00** | **1.00** | **2.00** |
| **3.4** | **4.16** | **7.21** | **8.67** |

**Access an element in m1: 4**

# R Data Structures: Matrices (Recycling)

- Recycling also applies to Matrices - for instance

```
m1 + c(1, 1)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]  4.0 5.00 2.00 3.00
## [2,]  4.4 5.16 8.21 9.67
```

# R Data Structures: Matrices (Recycling)

- Recycling also applies to Matrices - for instance

```
m1 + c(1, 1)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]  4.0 5.00 2.00 3.00
## [2,]  4.4 5.16 8.21 9.67
```

- What happened? the vector was added to each column of m1

# R Data Structures: Matrices (Recycling)

- Recycling also applies to Matrices - for instance

```
m1 + c(1, 1)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]  4.0 5.00 2.00 3.00
## [2,]  4.4 5.16 8.21 9.67
```

- What happened? the vector was added to each column of m1

```
m1 + c(1, 2)
```

```
##      [,1] [,2] [,3]  [,4]
## [1,]  4.0 5.00 2.00  3.00
## [2,]  5.4 6.16 9.21 10.67
```

# R Data Structures: Lists

- Lists - I think of lists as a generalized version of a vector, where the type of each index need not be the same. The first index of a list can be a scalar, the second index of a list can be a vector, the third index can be another list, and so on. Examples are below:

# R Data Structures: Lists

- Lists - I think of lists as a generalized version of a vector, where the type of each index need not be the same. The first index of a list can be a scalar, the second index of a list can be a vector, the third index can be another list, and so on. Examples are below:

```
l1 <- list(y1, v1, m1)
l2 <- list(first = l1, second = v1,
           third = m1, fourth = function(x) {x ^ 2})
```

# R Data Structures: Lists

- Lists - I think of lists as a generalized version of a vector, where the type of each index need not be the same. The first index of a list can be a scalar, the second index of a list can be a vector, the third index can be another list, and so on. Examples are below:

```
l1 <- list(y1, v1, m1)
l2 <- list(first = l1, second = v1,
           third = m1, fourth = function(x) {x ^ 2})
```

- Notice that both lists above are valid in R. You can even place functions into a list, as we did in l2.

# R Data Structures: Lists

- Lists - I think of lists as a generalized version of a vector, where the type of each index need not be the same. The first index of a list can be a scalar, the second index of a list can be a vector, the third index can be another list, and so on. Examples are below:

```
l1 <- list(y1, v1, m1)
l2 <- list(first = l1, second = v1,
           third = m1, fourth = function(x) {x ^ 2})
```

- Notice that both lists above are valid in R. You can even place functions into a list, as we did in l2.

# R Data Structures: Lists

- Lists - I think of lists as a generalized version of a vector, where the type of each index need not be the same. The first index of a list can be a scalar, the second index of a list can be a vector, the third index can be another list, and so on. Examples are below:

```
l1 <- list(y1, v1, m1)
l2 <- list(first = l1, second = v1,
           third = m1, fourth = function(x) {x ^ 2})
```

- Notice that both lists above are valid in R. You can even place functions into a list, as we did in l2.
- Lists are incredibly useful, as we will see later in this lecture. Lists can allow for calculating complicated expressions involved in survival likelihoods, for example.

# R Data Structures: Lists (Cont'd)

- How do we access elements in lists? It works similar to vectors/matrices...

# R Data Structures: Lists (Cont'd)

- How do we access elements in lists? It works similar to vectors/matrices, but there is more notation

```
l1[[1]]
```

```
## [1] 3
```

```
l1[[2]]
```

```
## [1] 3 4 1 2
```

```
l2$second
```

```
## [1] 3 4 1 2
```

# R Data Structures: Lists (Cont'd)

- One element in a list can be quite complicated!

```
l2$first
```

```
## [[1]]
## [1] 3
##
## [[2]]
## [1] 3 4 1 2
##
## [[3]]
##      [,1] [,2] [,3] [,4]
## [1,]  3.0 4.00 1.00 2.00
## [2,]  3.4 4.16 7.21 8.67
```

# R Data Structures: Data Frames

- Data frames are the fundamental data structure in R. They share many similarities with matrices and lists.
- A data frame is formally a list of variables with the same number of rows. You can therefore access each variable as you would an element in a list, for instance, with the $ operator.

# R Data Structures: Data Frames

- Data frames are the fundamental data structure in R. They share many similarities with matrices and lists.
- A data frame is formally a list of variables with the same number of rows. You can therefore access each variable as you would an element in a list, for instance, with the $ operator.
- Data frames are matrix like objects which can have columns of different types. Compare this to lists, where each index can be of different type. For instance, the first column may be numeric, the second may be a factor, the third may be of type string, etc. This can be accomplished with lists, but data frames provide extra convenience, including easier access and other functionality

# R Data Structures: Data Frames

- Example:

```
mySeed <- char2seed("Spencer")
set.seed(mySeed) ## So df1 stays the same
df1 <- data.frame(y1 = rnorm(4, 0, 1), y2 = rexp(4, 1),
                  y3 = rbeta(4, 1, 1), ind = c(1,1,0,0))
knitr::kable(df1)
```

| y1 | y2 | y3 | ind |
|---:|---:|---:|---:|
| 0.6322858 | 0.1558076 | 0.5478233 | 1 |
| 0.0558107 | 1.3448398 | 0.8152160 | 1 |
| 0.8099077 | 1.5584303 | 0.6714133 | 0 |
| 0.9714999 | 1.3880223 | 0.6401974 | 0 |

```
df1$y1
```

```
## [1] 0.63228583 0.05581074 0.80990771 0.97149987
```

# R Data Structures: Data Frames

- It is very simple to calculate new variables in a data set. We will learn dplyr later in this class, but for now I will teach the "old guard" way of doing things

```
df1$y4 <- df1$y1^2 + df1$y2^2 + df1$y3^2
df1$y4
```

```
## [1] 0.7241718 2.4762859 3.5354515 3.2802706
```

- You can also look at rows or columns using the following code

```
## Access row 2
df1[2,]
## Access colum 3
df1[,3]
```

# R Data Structures: Data Frames

- R Data frames can be viewed as the building blocks for using the R language.

# R Data Structures: Data Frames

- R Data frames can be viewed as the building blocks for using the R language.
  - In almost every case, main analysis/modeling function in CRAN packages take data frames as arguments.

# R Data Structures: Data Frames

- R Data frames can be viewed as the building blocks for using the R language.
  - In almost every case, main analysis/modeling function in CRAN packages take data frames as arguments.
  - Includes the information used to construct design matrices, start the optimization algorithm for estimating model parameters

# R Data Structures: Data Frames

- R Data frames can be viewed as the building blocks for using the R language.
  - In almost every case, main analysis/modeling function in CRAN packages take data frames as arguments.
  - Includes the information used to construct design matrices, start the optimization algorithm for estimating model parameters
  - Using data frames efficiently in your own software can make writing code very efficient. As an example, check out the model.frame() and model.matrix() functions.

# R Data Structures: Functions

- Functions in R work exactly as they do in other languages.

# R Data Structures: Functions

- Functions in R work exactly as they do in other languages.
  1) You give a function an input or series of inputs

# R Data Structures: Functions

- Functions in R work exactly as they do in other languages.
  1) You give a function an input or series of inputs
  2) Actions are performed with the inputs (this is determined by the body of the function)

# R Data Structures: Functions

- Functions in R work exactly as they do in other languages.
  1) You give a function an input or series of inputs
  2) Actions are performed with the inputs (this is determined by the body of the function)
  3) A result is returned or other objects are manipulated in the process

# R Data Structures: Functions

- Functions in R work exactly as they do in other languages.
  1) You give a function an input or series of inputs
  2) Actions are performed with the inputs (this is determined by the body of the function)
  3) A result is returned or other objects are manipulated in the process
- The best practice is as follows when writing your own software, or even script for commonly used techniques/reports

# R Data Structures: Functions

- Functions in R work exactly as they do in other languages.
  1) You give a function an input or series of inputs
  2) Actions are performed with the inputs (this is determined by the body of the function)
  3) A result is returned or other objects are manipulated in the process
- The best practice is as follows when writing your own software, or even script for commonly used techniques/reports
  - Any repeated action should be made into a function

# R Data Structures: Functions

- Functions in R work exactly as they do in other languages.
  1) You give a function an input or series of inputs
  2) Actions are performed with the inputs (this is determined by the body of the function)
  3) A result is returned or other objects are manipulated in the process
- The best practice is as follows when writing your own software, or even script for commonly used techniques/reports
  - Any repeated action should be made into a function
  - Anticipate extending your functions by generalizing the procedure

# R Data Structures: Functions

- Functions in R work exactly as they do in other languages.
  1) You give a function an input or series of inputs
  2) Actions are performed with the inputs (this is determined by the body of the function)
  3) A result is returned or other objects are manipulated in the process
- The best practice is as follows when writing your own software, or even script for commonly used techniques/reports
  - Any repeated action should be made into a function
  - Anticipate extending your functions by generalizing the procedure
  - For instance, make the number of predictors a variable, or better yet let the function take a vector of strings which denotes all predictors

# R Data Structures: Function examples

```
fib <- function(number)
{
  ## Simple test for integer value - doesn't really
  ## work if number is VERY close to an integer
  isInt <- all.equal(number, as.integer(number))
  if (number <= 2)
  {
    return(1)
  } else
  {
    return(fib(number-1) + fib(number - 2))
  }
}
```

# Apply class of functions

- Apply class of functions take the place of loops, where we use a “counter" to iterate over entities in a matrix, vector, or list

# Apply class of functions

- Apply class of functions take the place of loops, where we use a “counter" to iterate over entities in a matrix, vector, or list
- First example function:**apply(X, MARGIN, FUN, ...)**

# Apply class of functions

- Apply class of functions take the place of loops, where we use a “counter" to iterate over entities in a matrix, vector, or list
- First example function: **apply(X, MARGIN, FUN, ...)**
  - X: an array, including a matrix. MARGIN: 1 (rows) or 2 (columns). FUN: the function to be applied. Returns the result applied to each row or column of X. May return a matrix if results are vectors.

# Apply class of functions: apply

- Apply class of functions take the place of loops, where we use a "counter" to iterate over entities in a matrix, vector, or list
- First example function: **apply(X, MARGIN, FUN, ...)**
  - X: an array, including a matrix. MARGIN: 1 (rows) or 2 (columns). FUN: the function to be applied. Returns the result applied to each row or column of X. May return a matrix if results are vectors.

```
## rows
apply(m1, 1, mean)
```

```
## [1] 2.50 5.86
```

```
## columns
apply(m1, 2, mean)
```

```
## [1] 3.200 4.080 4.105 5.335
```

# The aggregate function

- aggregate: Splits the data into subsets, computes summary statistics for each, and returns the result in a convenient form (most use dplyr in today's culture)

# The aggregate function

- aggregate: Splits the data into subsets, computes summary statistics for each, and returns the result in a convenient form (most use dplyr in today's culture)
- **aggregate(X, by, FUN)** - X: an R object. by: a list of grouping elements, each as long as the variables in the data frame x. The elements are coerced into factors before use. FUN: a function to compute the summary statistics which can be applied to all data subsets.

# The aggregate function

- aggregate: Splits the data into subsets, computes summary statistics for each, and returns the result in a convenient form (most use dplyr in today's culture)
- **aggregate(X, by, FUN)** - X: an R object. by: a list of grouping elements, each as long as the variables in the data frame x. The elements are coerced into factors before use. FUN: a function to compute the summary statistics which can be applied to all data subsets.

```
aggregate(df1, by = list(df1$ind), mean)
```

```
##   Group.1         y1        y2        y3 ind        y4
## 1       0 0.8907038 1.4732263 0.6558054   0 3.407861
## 2       1 0.3440483 0.7503237 0.6815196   1 1.600229
```

# Apply class of functions: apply

```
minMax <- function(x) {return(c(min(x), max(x)))}
## rows
apply(m1, 1, minMax)
```

```
##      [,1] [,2]
## [1,]    1 3.40
## [2,]    4 8.67
```

```
## columns
apply(m1, 2, minMax)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]  3.0 4.00 1.00 2.00
## [2,]  3.4 4.16 7.21 8.67
```

# Apply class of functions: apply

- Convenience functions exist for sum/mean:
- rowSums(), rowMeans(), colSums(), colMeans()
- rowSums() == apply(, 1, sum), colSums() = apply(, 2, sum), etc.

# Apply class of functions: lapply

- lapply: Apply a function over a list or vector

# Apply class of functions: lapply

- lapply: Apply a function over a list or vector
- **lapply(X, FUN)** - X: a vector (atomic or list). FUN: the function to be applied to each element of X

# Apply class of functions: lapply

- lapply: Apply a function over a list or vector
- **lapply(X, FUN)** - X: a vector (atomic or list). FUN: the function to be applied to each element of X
- In the below, each element of l3 is a vector of 10000 standard normal deviates

```
lapply(l3, mean)
```

```
## [[1]]
## [1] -0.01255568
##
## [[2]]
## [1] -0.009359543
##
## [[3]]
## [1] 0.00768243
```

# Apply class of functions: sapply

- sapply: user-friendly version and wrapper of lapply - by default returns a vector instead of a list - useful when you expect function call to return a list of scalars

# Apply class of functions: sapply

- sapply: user-friendly version and wrapper of lapply - by default returns a vector instead of a list - useful when you expect function call to return a list of scalars

```
sapply(l3, mean)
```

```
## [1] -0.012555679 -0.009359543  0.007682430
```

```
sapply(l3, function(x) {sum(x^2) + 10})
```

```
## [1] 10196.602  9673.241  9858.207
```

# Apply class of functions: tapply

- tapply: apply a function to each cell of a ragged array

# Apply class of functions: tapply

- tapply: apply a function to each cell of a ragged array
- ragged arrays are arrays, where within each index contains another array that may be of non-constant length

# Apply class of functions: tapply

- tapply: apply a function to each cell of a ragged array
- ragged arrays are arrays, where within each index contains another array that may be of non-constant length
- aggregate is a convenience function for tapply, tapply also applied to lists

# Apply class of functions: tapply

- tapply: apply a function to each cell of a ragged array
- ragged arrays are arrays, where within each index contains another array that may be of non-constant length
- aggregate is a convenience function for tapply, tapply also applied to lists

```
tapply(df1$y1, df1$ind, mean)
```

```
##         0         1
## 0.8907038 0.3440483
```

# Apply class of functions: mapply

- The mapply function has been the most useful of the apply functions for my work

# Apply class of functions: mapply

- The mapply function has been the most useful of the apply functions for my work
- It is essentially a multivariate version of lapply, with slightly different syntax

# Apply class of functions: mapply

- The mapply function has been the most useful of the apply functions for my work
- It is essentially a multivariate version of lapply, with slightly different syntax
- **mapply(FUN, ..., SIMPLIFY = TRUE)** - FUN: function to apply, ...: lists or vectors to iterate over, SIMPLIFY: logical or character string, should R attemp to reduce the result to a vector/matrix, or keep as a list?

# Apply class of functions: mapply

- The mapply function has been the most useful of the apply functions for my work
- It is essentially a multivariate version of lapply, with slightly different syntax
- **mapply(FUN, ..., SIMPLIFY = TRUE)** - FUN: function to apply, ...: lists or vectors to iterate over, SIMPLIFY: logical or character string, should R attemp to reduce the result to a vector/matrix, or keep as a list?
- I almost always use SIMPLIFY = FALSE to keep list structure of answer

# Example of using mapply

- Function mapply is a go-to when you have multiple lists or vectors of the same length and need to use the same operation on each index before gathering your results.

# Example of using mapply

- Function mapply is a go-to when you have multiple lists or vectors of the same length and need to use the same operation on each index before gathering your results.
- Recall the multivariate normal likelihood, i.e. we observe an independent sample (of size $n$), with $i$ th observation denoted $Y_i$ with dimension $q_i x 1$, and assume that:

$$Y_{ij} = X_{ij}\beta + b_{oi} + \epsilon_{ij} \quad (1)$$

$$\sim N(X_{ij}\beta, \sigma_b^2 + \sigma^2) \quad (2)$$

- Where $X_{ij}$ is a vector of covariates for the $i$th subject at the $j$th timepoint. This is referred to as a linear mixed model with random subject intercepts.

# Example of using mapply

- Calculation of the log-likelihood requires matrix multiplication of $X_i$ and a current estimate for $\beta$, i.e. $\hat{\beta}$, then subtracting this from $y_i$, and finally, forming a quadratic form with $\Sigma_i^{-1}$

# Example of using mapply

- Calculation of the log-likelihood requires matrix multiplication of $X_i$ and a current estimate for $\beta$, i.e. $\hat{\beta}$, then subtracting this from $y_i$, and finally, forming a quadratic form with $\Sigma_i^{-1}$
- The above calculation denotes $(y_i - X_i\beta)' \frac{\Sigma_i^{-1}}{2} (y_i - X_i\beta)$ from the log-likelihood in this problem

# Example of using mapply

- Calculation of the log-likelihood requires matrix multiplication of $X_i$ and a current estimate for $\beta$, i.e. $\hat{\beta}$, then subtracting this from $y_i$, and finally, forming a quadratic form with $\Sigma_i^{-1}$
- The above calculation denotes $(y_i - X_i\beta)' \frac{\Sigma_i^{-1}}{2} (y_i - X_i\beta)$ from the log-likelihood in this problem
- The following slides show how to use mapply and other base R functions to get this calculation off the ground using a toy data set.

# Example of using mapply (Cont'd)

- Consider the following dataset with $y_1$ and $x_1, x_2, x_3, x_4$ included.
- Example of what one index might look like:

```
dfYSplit[[2]]$y1
```

```
## [1] 5.270686 2.216416 2.081399 6.207842 6.689458 7.177134
```

```
knitr::kable(dfXSplit[[2]][,c("x1", "x2", "x3", "x4")])
```

| | x1 | x2 | x3 | x4 |
|---|---|---|---|---|
| 9 | 0.0218036 | 0.0787564 | 3.178310 | 2.5635652 |
| 10 | 1.9215528 | 2.0730261 | 3.485042 | 0.6018452 |
| 11 | 1.1020457 | -0.3379613 | 4.231701 | -0.2174204 |
| 12 | 1.4894253 | 0.5910463 | 2.763400 | 2.3932846 |
| 13 | 1.3648414 | 0.4576341 | 3.636919 | 2.4375363 |
| 14 | 1.6343576 | 1.0716887 | 2.232086 | 3.2910254 |

# Example of using mapply (Cont'd)

- Current beta estimate:

```
beta
```

```
## [1]  0.25 -0.25  0.50  1.25
```

# Example of using mapply (Cont'd)

- Current beta estimate:

```
beta
```

```
## [1]  0.25 -0.25  0.50  1.25
```

- Current $sigma^2$ and $sigma_b^2$ estimates:

```
sigma2
```

```
## [1] 1.25
```

```
sigma2b
```

```
## [1] 0.25
```

# Example of using mapply (Cont'd)

- The below code puts it all together!

```
quadForms <- mapply(function(a, b) {
  currY <- a[,"y1"]
  currXM <- b[,c("x1", "x2", "x3", "x4")]
  fixed <- as.matrix(currXM) %*% beta
  ni <- dim(currXM)[1]
  Sigma <- diag(sigma2, ni) + matrix(rep(sigma2b, ni * ni),
                ncol = ni, nrow = ni) - diag(sigma2b, ni)
  res <- t(currY - fixed) %*% solve(Sigma) %*%
           (currY - fixed) / 2
  return(res)
}, dfYSplit, dfXSplit, SIMPLIFY = FALSE)
```

# Example of using mapply (Cont'd)

- The below code puts it all together!

```
quadForms <- mapply(function(a, b) {
  currY <- a[,"y1"]
  currXM <- b[,c("x1", "x2", "x3", "x4")]
  fixed <- as.matrix(currXM) %*% beta
  ni <- dim(currXM)[1]
  Sigma <- diag(sigma2, ni) + matrix(rep(sigma2b, ni * ni),
                ncol = ni, nrow = ni) - diag(sigma2b, ni)
  res <- t(currY - fixed) %*% solve(Sigma) %*%
           (currY - fixed) / 2
  return(res)
}, dfYSplit, dfXSplit, SIMPLIFY = FALSE)
```

- Mapply provides a convenient way to put our "building blocks" together

# Summary

- R is a computational programming language primarily used by Statisticians/Data Scientists

# Summary

- R is a computational programming language primarily used by Statisticians/Data Scientists
- Many existing data structures, i.e. vectors, matrices, lists, functions...

# Summary

- R is a computational programming language primarily used by Statisticians/Data Scientists
- Many existing data structures, i.e. vectors, matrices, lists, functions...
- Apply class of functions very useful for replacing loops which conduct same operation many times

# Summary

- R is a computational programming language primarily used by Statisticians/Data Scientists
- Many existing data structures, i.e. vectors, matrices, lists, functions...
- Apply class of functions very useful for replacing loops which conduct same operation many times
- Mapply function in particular is very useful when we have many lists.

# Summary

- R is a computational programming language primarily used by Statisticians/Data Scientists
- Many existing data structures, i.e. vectors, matrices, lists, functions...
- Apply class of functions very useful for replacing loops which conduct same operation many times
- Mapply function in particular is very useful when we have many lists
- Note that lapply can replace mapply if each list element has multiple list entities
- Ex: l1[[1]] = list(l11, l12, l13, l14) (access with the $ operator)