ALTUNA AKALIN

NOTES ON COMPUTATIONAL GENOMICS WITH R

Contents

1
Computational genomics and R

1.1 Computational Genomics

The aim of computational genomics is to do biological interpretation of high dimensional genomics data. Generally speaking, it is similar to any other kind of data analysis but often times doing computational genomics will require domain specific knowledge and tools. The tasks of computational genomics can be roughly summarized as in Figure 1.1.

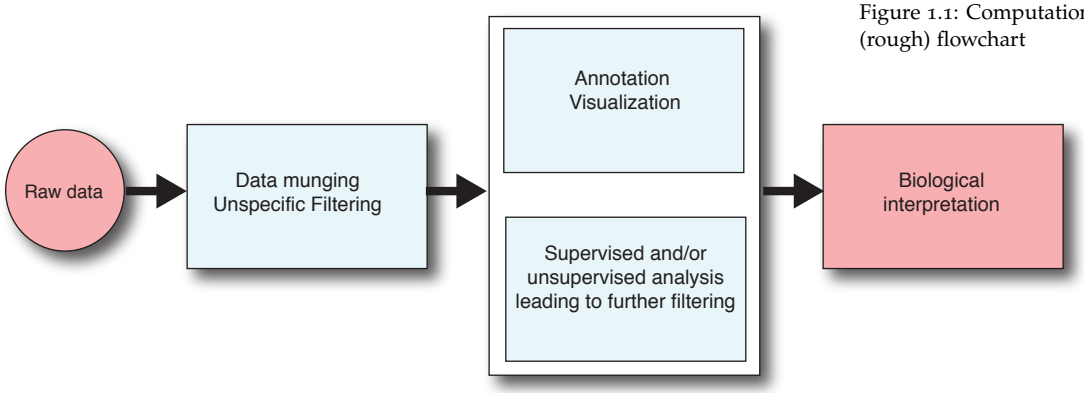

Figure 1.1: Computational genomics (rough) flowchart

Most of the time, the analysis starts with the raw data (if you are somehow served with already processed data, consider yourself lucky). The raw data could be image files from a microarray, or text files from a sequencer. That means the analysis starts with processing the data to a manageable format. This processing includes data munging which is transforming the data from one format to another, data point transformations such as normalizing and log-transformations, and filtering and removing data points with specific thresholds. For example. you may want to remove data points that have unreliable measurements or missing values. The next step is to apply supervised/unsupervised learning algorithms to test the

hypothesis that lead to experiments that generated the data. This step also includes quality checks about your data. Another important thing to do is to annotate and visualize your data. For example, after finding transcription factor binding sites using data from ChIP-seq experiments, you would like to see what kind of genes they are nearby or what kind of other genomic features they overlap with. Do they overlap with promoters or they are distal? You may also like to see your binding sites on the genome or make a summary plot showing distances to nearest transcriptions start sites. All of this tasks can be classified as visualization and annotation tasks. All of these steps should hopefully lead you to the holy grail which is biological interpretation of the data and hopefully to some new insights about genome biology.

1.2 What can you do with R?

The tasks that are descrived above can be accomplished by R. R is not only a powerful statistical programming language but also go-to data analysis tool for many computational genomics experts. High-dimensional genomics datasets are usually suitable to be analyzed with core R packages and functions. On top of that, Bioconductor and CRAN have an array of specialized tools for doing genomics specific analysis.

Here is a list of computational genomics tasks that can be completed using R.

Data munging (pre-processing)

Often times, the data will not come in ready to analyze format. You may need to convert it to other formats by transforming data points (such as log transforming, normalizing etc), or remove columns/rows that , or remove data points with empty values and, and subset the data set with some arbitrary condition. Most of these tasks can be achieved using R. In addition, with the help of packages R can connect to databases in various formats such as mySQL, mongoDB, etc., and query and get the data to R environment using database specific tools. Unfortunately, not all data muging and processing tasks can be accomplished only by R. At times, you may need to use domain specific software or software dealing better with specific type of data sets. For example, R is not great at dealing with character strings, if you are trying to filter a large dataset based on some regular expression you may be better of with perl or awk.

General data analysis and exploration

Most genomics data sets are suitable for application of general data analysis tools. In some cases, you may need to preprocess the data to get it to a state that is suitable for application such tools.

- unsupervised data analysis: clustering (k-means, hierarchical), matrix factorization (PCA, ICA etc)
- supervised data analysis: generalized linear models, support vector machines, randomForests

Visualization

Visualization is an important part of all data analysis techniques including computational genomics. Again, you can use core visualization techniques in R and also genomics specific ones with the help of specific packages.

- Basic plots: Histograms, scatter plots, bar plots, box plots
- ideograms and circus plots for genomics
- heatmaps
- meta-profiles of genomic features, read enrichment over all promoters
- genomic track visualization for given locus

Dealing with genomic intervals

Most of the genomics data come in a tabular format that contains the location in the genome and some other relevant values, such as scores for those genomic features and/or names. R/Bioconductor has dedicated methods to deal with such data. Here are a couple of example tasks that you can achieve using R.

- Overlapping CpG islands with transcription start sites, and filtering based on overlaps
- Aligning reads and making read enrichment profiles
- Overlapping aligned reads with exons and counting aligned reads per gene

Application of other bioinformatics specific algorithms

In addition to genomic interval centered methods, R/Bioconductor gives you access to multitude of other bioinformatics specific algorithms. Here are some of the things you can do.

- Sequence analysis: TF binding motifs, GC content and CpG counts of a given DNA sequence
- Differential expression (or arrays and sequencing based measurements)
- Gene set/Pathway analysis: What kind of genes are enriched in my gene set

2
Quick introduction to R

R is a free statistical programming language that is popular among researchers and data miners to build software and analyze data[1]. In next sections, we will first get you started with the setup of environment for using R and then introduce some basic R operations and data structures that will be good to know if you do not have prior experience with R. If you need more in depth R introduction you will want to check out some beginner level books and online tutorials. This website has a bunch of resources listed: `http://www.introductoryr.co.uk/R_Resources_for_Beginners.html`

[1] if you want to know more about details about R and its history here is a good place to start `http://en.wikipedia.org/wiki/R_(programming_language)`

2.1 The setup

Download and install R `http://cran.r-project.org/` and RStudio `http://www.rstudio.com/` if you do not have them already. Rstudio is optional but it is a great tool if you are just starting to learn R. You will need specific data sets to run the codes in this document. Download the data.zip[URL to come] and extract it to your directory of choice. The folder name should be "data" and your R working directory should be level above the data folder. That means in your R console, when you type "*dir("data")*" you should be able to see the contents of the data folder. You can change your working directory by *setwd()* command and get your current working directory with *getwd()* command in R[2]. In RStudio, you can click on the top menu and change the location of your working directory via user interface.

[2] TIP: dir() gives you files in your current working directory. getwd() gets current directory. You may need these at some point.

Installing packages

R packages are add-ons to base R that help you achieve additional tasks that are not directly supported by base R. It is by the action of these extra functionality that R excels as a tool for computational genomics. Bioconductor project (`http://bioconductor.org/`) is a dedicated package repository for computational biology related

packages. However main package repository of R, called CRAN, has also computational biology related packages. In addition, R-Forge(`http://r-forge.r-project.org/`), GitHub(`https://github.com/`), and googlecode(`http://code.google.com`) are other locations where R packages might be hosted.

You can install CRAN packages using install.packages(). (# is the comment character in R)

```
# install package named 'randomForests' from CRAN
install.packages("randomForests")
```

You can install packages from bioconductor with their specific installer script.

```
# get the installer package
source("http://bioconductor.org/biocLite.R")
# install bioconductor package 'rtracklayer'
biocLite("rtracklayer")
```

Installing packages from GitHub via install_github function from devtools package.

```
library(devtools)
install_github("roxygen")
```

You can install packages from the source files, that usually have .tar.gz suffix.

```
# download the source file
download.file("http://goo.gl/3pvHYI",
              destfile="methylKit_0.5.7.tar.gz")
# install the package from the source file
install.packages("methylKit_0.5.7.tar.gz",
                 repos=NULL,type="source")
# delete the source file
unlink("methylKit_0.5.7.tar.gz")
```

You can also update installed packages from CRAN and Bioconductor.

```
# updating CRAN packages
update.packages()
# updating bioconductor packages
source("http://bioconductor.org/biocLite.R")
biocLite("BiocUpgrade")
```

installing packages in custom locations

If you will be using R on servers or computing clusters rather than your personal computer it is unlikey that you will have administrator access to install packages. In that case, you can install packges in custom locations by telling R where to look for additional packages. This is done by setting up an *.Renviron* file in your home directory and add the following line:

```
R_LIBS=~/Rlibs
```

This tells R that "Rlibs" directory at your home directory will be the first choice of locations to look for packages and install packages (The directory name and location is up to you above is just an example). You should go and create that directory now. After that, start a fresh R session and start installing packages. From now on, packages will be installed to your local directory where you have read-write access.

Getting help on R functions and packages

You can get help on functions by help() and help.search() functions. You can list the functions in a package with ls() function

```
libray(MASS)
ls("package:MASS")  # functions in the package
ls()  # objects in your R enviroment
# get help on hist() function
`?`(hist)
help("hist")
# search the word 'hist' in help pages
help.search("hist")
`?`(`?`(hist))
```

In addition, check package vignettes for help and practical understanding of the functions. All Bionconductor packages have vignettes that walk you thorugh example analysis. Google search will always be helpful as well, there are many blogs and web pages that have posts about R.

2.2 *Computations in R*

R can be used as an ordinary calculator. Here are a few examples:

```r
2 + 3 * 5  # Note the order of operations.

## [1] 17

log(10)  # Natural logarithm with base e

## [1] 2.303

5^2  # 5 raised to the second power

## [1] 25

3/2  # Division

## [1] 1.5

sqrt(16)  # Square root

## [1] 4

abs(3 - 7)  # Absolute value of 3-7

## [1] 4

pi  # The number

## [1] 3.142

exp(2)  # exponential function

## [1] 7.389

# This is a comment line
```

2.3 *Vectors*

Vectors is one the core R data structures. R handles vectors easily and intuitively. You can create vectors with *c()* function, however that is not the only way. The operations on vectors will propagate to all the elements of the vectors.

```r
x <- c(1, 3, 2, 10, 5)  #create a vector x with 5 components
x

## [1]  1  3  2 10  5

y <- 1:5  #create a vector of consecutive integers y
y + 2  #scalar addition
```

```
## [1] 3 4 5 6 7

2 * y  #scalar multiplication

## [1]  2  4  6  8 10

y^2  #raise each component to the second power

## [1]  1  4  9 16 25

2^y  #raise 2 to the first through fifth power

## [1]  2  4  8 16 32

y  #y itself has not been unchanged

## [1] 1 2 3 4 5

y <- y * 2
y  #it is now changed

## [1]  2  4  6  8 10

r1 <- rep(1, 3)  # create a vector of 1s, length 3
length(r1)  #length of the vector

## [1] 3

class(r1)  # class of the vector

## [1] "numeric"

a <- 1  # this is actually a vector length one
```

Data types in R

There are four comon data types in R, they are numeric, logical and character and integer. All these data types can be used to create vectors natively.

```
# create a numeric vector x with 5 components
x <- c(1, 3, 2, 10, 5)
x

## [1]  1  3  2 10  5

# create a logical vector x
x <- c(TRUE, FALSE, TRUE)
x
```

```
## [1]  TRUE FALSE  TRUE

# create a character vector
x <- c("sds", "sd", "as")
x

## [1] "sds" "sd"  "as"

class(x)

## [1] "character"

# create an integer vector
x <- c(1L, 2L, 3L)
x

## [1] 1 2 3

class(x)

## [1] "integer"
```

2.4 *Matrices*

A matrix refers to a numeric array of rows and columns. You can think of it as a stacked version of vectors where each row or column is a vector. One of the easiest ways to create a matrix is to combine vectors of equal length using *cbind(),* meaning 'column bind'

```
x <- c(1, 2, 3, 4)
y <- c(4, 5, 6, 7)
m1 <- cbind(x, y)
m1

##      x y
## [1,] 1 4
## [2,] 2 5
## [3,] 3 6
## [4,] 4 7

t(m1)  # transpose of m1

##   [,1] [,2] [,3] [,4]
## x    1    2    3    4
## y    4    5    6    7

dim(m1)  # 2 by 5 matrix
```

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

You can also directly list the elements and specify the matrix:

```
m2 <- matrix(c(1, 3, 2, 5, -1, 2, 2, 3, 9), nrow = 3)
m2

##      [,1] [,2] [,3]
## [1,]    1    5    2
## [2,]    3   -1    3
## [3,]    2    2    9
```

2.5 *Data Frames*

A data frame is more general than a matrix, in that different columns can have different modes (numeric, character, factor, etc.). A data frame can be constructed by *data.frame()* function. For example, we illustrate how to construct a data frame from genomic intervals or coordinates.

```
chr <- c("chr1", "chr1", "chr2", "chr2")
strand <- c("-", "-", "+", "+")
start <- c(200, 4000, 100, 400)
end <- c(250, 410, 200, 450)
mydata <- data.frame(chr, start, end, strand)
# change column names
names(mydata) <- c("chr", "start", "end", "strand")
mydata  # OR this will work too

##    chr start end strand
## 1 chr1   200 250      -
## 2 chr1  4000 410      -
## 3 chr2   100 200      +
## 4 chr2   400 450      +

mydata <- data.frame(chr = chr, start = start, end = end,
    strand = strand)
mydata

##    chr start end strand
## 1 chr1   200 250      -
## 2 chr1  4000 410      -
## 3 chr2   100 200      +
## 4 chr2   400 450      +
```

There are a variety of ways to extract the elements of a data frame. You can extract certain columns using column numbers or names, or you can extract certain rows by using row numbers. You can also extract data using logical arguments, such as extracting all rows that has a value in a column larger than your threshold.

```
mydata[, 2:4]  # columns 2,3,4 of data frame

##   start end strand
## 1   200 250      -
## 2  4000 410      -
## 3   100 200      +
## 4   400 450      +

mydata[, c("chr", "start")]  # columns chr and start from data frame

##    chr start
## 1 chr1   200
## 2 chr1  4000
## 3 chr2   100
## 4 chr2   400

mydata$start  # variable start in the data frame

## [1]  200 4000  100  400

mydata[c(1, 3), ]  # get 1st and 3rd rows

##    chr start end strand
## 1 chr1   200 250      -
## 3 chr2   100 200      +

mydata[mydata$start > 400, ]  # get all rows where start>400

##    chr start end strand
## 2 chr1  4000 410      -
```

2.6 *Lists*

An ordered collection of objects (components). A list allows you to gather a variety of (possibly unrelated) objects under one name.

```
# example of a list with 4 components
# a string, a numeric vector, a matrix, and a scalar
w <- list(name="Fred",
       mynumbers=c(1,2,3),
```

```
       mymatrix=matrix(1:4,ncol=2),
       age=5.3)
w

## $name
## [1] "Fred"
##
## $mynumbers
## [1] 1 2 3
##
## $mymatrix
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## $age
## [1] 5.3
```

You can extract elements of a list using the [[]] convention using either its position in the list or its name.

```
w[[3]]  # 3rd component of the list

##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4

w[["mynumbers"]]  # component named mynumbers in list

## [1] 1 2 3
```

2.7 *Factors*

Factors are used to store categorical data. They are important for statistical modeling since categorical variables are treated differently in statistical models than continuos variables. This ensures categorical data treated accordingly in statistical models.

```
features <- c("promoter", "exon", "intron")
f.feat <- factor(features)
```

Important thing to note is that when you are reading a data.frame with read.table() or creating a data frame with data.frame() character columns are stored as factors by default, to change this behaviour you need to set *stringsAsFactors=FALSE* in read.table() and/or data.frame() function arguments.

2.8 *Reading/Writing data*

Most of the genomics data are in the form of genomic intervals associated with a score. That means mostly the data will be in table format with columns denoting chromosome, start positions, end positions, strand and score. One of the popular formats is BED format used primarily by UCSC genome browser but most other genome browsers and tools will support BED format. We have all the annotation data in BED format. In R, you can easily read tabular format data with *read.table()* function.

```
enh.df <- read.table("data/subset.enhancers.hg18.bed",
    header = FALSE)  # read enhancer marker BED file
cpgi.df <- read.table("data/subset.cpgi.hg18.bed",
    header = FALSE)  # read CpG island BED file
# check first lines to see how the data looks like
head(enh.df)

##      V1     V2     V3 V4   V5 V6    V7    V8 V9
## 1 chr20 266275 267925  . 1000  .  9.11 13.17 -1
## 2 chr20 287400 294500  . 1000  . 10.53 13.02 -1
## 3 chr20 300500 302500  . 1000  .  9.10 13.39 -1
## 4 chr20 330400 331800  . 1000  .  6.39 13.51 -1
## 5 chr20 341425 343400  . 1000  .  6.20 12.99 -1
## 6 chr20 437975 439900  . 1000  .  6.31 13.52 -1

head(cpgi.df)

##      V1     V2     V3       V4
## 1 chr20 195575 195851  CpG:_28
## 2 chr20 207789 208148  CpG:_32
## 3 chr20 219055 219437  CpG:_33
## 4 chr20 225831 227155 CpG:_135
## 5 chr20 252826 256323 CpG:_286
## 6 chr20 275376 276977 CpG:_116
```

You can save your data by writing it to disk as a text file. A data frame or matrix can be written out by using *write.table()* function. Now let us write out cpgi.df, we will write it out as a tab-separated file, pay attention to the arguments.

```
write.table(cpg.df, file = "cpgi.txt", quote = FALSE,
    row.names = FALSE, col.names = FALSE, sep = "\t")
```

You can save your R objects directly into a file using *save()* and *saveRDS()* and load them back in with load() and readRDS(). By

using these functions you can save any R object whether or not they are in data frame or matrix classes.

```
save(cpg.df, enh.df, file = "mydata.RData")
load("mydata.RData")
# saveRDS() can save one object at a type
saveRDS(cpg.df, file = "cpg.rds")
x <- readRDS("cpg.rds")
head(x)
```

One important thing is that with save() you can save many objects at a time and when they are loaded into memory with load() they retain their variable names. For example, in the above code when you use load("mydata.RData") in a fresh R session, an object names "cpg.df" will be created. That means you have to figure out what name you gave it to the objects before saving them. On the contrary to that, when you save an object by saveRDS() and read by readRDS() the name of the object is not retained, you need to assign the output of readRDS() to a new variable ("x" in the above code chunk).

2.9 *Plotting in R*

R has great support for plotting and customizing plots. We will show only a few below. Let us sample 50 values from normal distribution and plot them as a histogram (See Figure2.1).

```
# sample 50 values from normal distribution
# and store them in vector x
x<-rnorm(50)
hist(x) # plot the histogram of those values
```

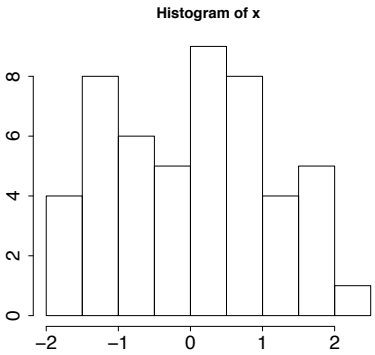

Figure 2.1: Histogram of 50 sampled values

We can modify all the plots by providing certain arguments to the plotting function. Now let's give a title to the plot using *'main'* argument. We can also change the color of the bars using *'col'* argument.

You can simply provide the name of the color. Below, we are using '*red*' for the color. See Figure 2.2 for the result this chunk.

```
hist(x, main = "Hello histogram!!!", col = "red")
```

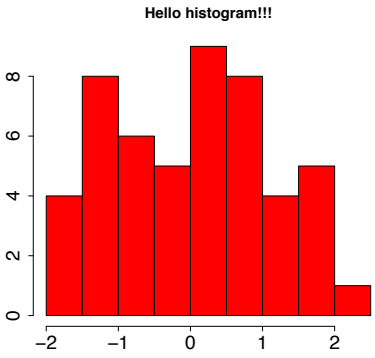

Figure 2.2: Histogram with a title

Next, we will plot a scatter plot. Scatter plots are one os the most common plots you will encounter in data analysis. We will sample another set of 50 values and plotted those against the ones we sampled earlier. Scatterplot shows values of two variables for a set of data points. It is useful to visualize relationships between two variables. It is frequently used in connection with correlation and linear regression. There are other variants of scatter plots which show density of the points with different colors. We will show examples of those that in following chapters. The scatter plot from our sampling experiment is shown in Figure 2.3. Notice that, in addition to main we used "xlab" and "ylab" arguments to give labels to the plot. You can customize the plots even more than this. See ?plot and ?par for more arguments that can help you customize the plots.

```
# randomly sample 50 points from normal distribution
y<-rnorm(50)
#plot a scatter plot
# control x-axis and y-axis labels
plot(x,y,main="scatterplot of random samples",
        ylab="y values",xlab="x values")
```

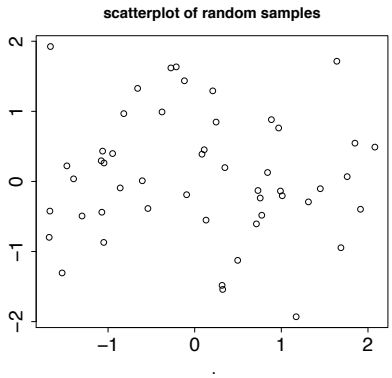

Figure 2.3: scatter plot of two random variables

we can also plot boxplots for vectors x and y. Boxplots depict groups of numerical data through their quartiles. The edges of the box denote 1st and 3rd quartile, and the line that crosses the box is the median. Whiskers usually are defined using interquantile range,$lowerWhisker = Q1 - 1.5 * IQR$ and $upperWhisker = Q1 + 1.5 * IQR$. In addition outliers can be depicted as dots. In this case, outliers are the values that remain outside the whiskers.

```
boxplot(x,y,main="boxplots of random samples")
```

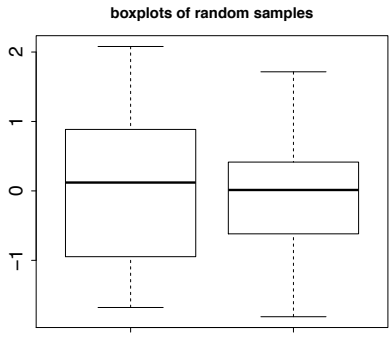

Figure 2.4: Boxplots for x and y vectors

Next up is bar plot which you can plot by barplot() function. We are going to plot four imaginary percentage values and color them with two colors, and this time we will also show how to draw a legend on the plot using legend() function.

```
perc=c(50,70,35,25)
barplot(height=perc,names.arg=c("CpGi","exon","CpGi","exon"),
        ylab="percentages",main="imagine %s",
        col=c("red","red","blue","blue"))
legend("topright",legend=c("test","control"),fill=c("red","blue"))
```

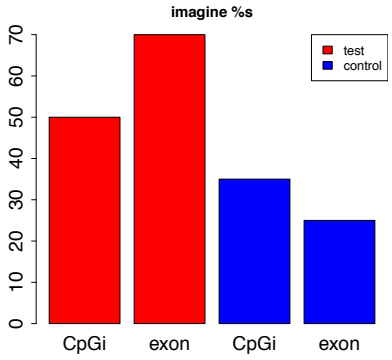

Figure 2.5: Bar plot and legend example

Saving Plots

If you want to save your plots to an image file there are couple of ways of doing that. Normally, you will have to do the following:

1. Open a graphics device
2. Create the plot

3. Close the graphics device

```
pdf("mygraphs/myplot.pdf", width = 5, height = 5)
plot(x, y)
dev.off()
```

Alternatively, you can first create the plot then copy the plot to a graphical device.

```
plot(x, y)
dev.copy(pdf, "mygraphs/myplot.pdf", width = 7, height = 5)
dev.off()
```

2.10 *User defined functions*

Functions are useful for transforming larger chunks of code to reusable pieces of code. Generally, if you need to execute certain tasks with variable parameters then it is time you write a function. A function in R takes different arguments and returns a definite output, much like mathematical functions. Here is a simple function takes two arguments, x and y, and returns the sum of their squares.

```
sqSum <- function(x, y) {
    result <- x^2 + y^2
    return(result)
}
# now try the function out
sqSum(2, 3)

## [1] 13
```

Functions can also output plots and/or messages to the terminal. Here is a function that prints a message to the terminal:

```
sqSumPrint <- function(x, y) {
    result <- x^2 + y^2
    cat("here is the result:", result, "\n")
}
# now try the function out
sqSumPrint(2, 3)

## here is the result: 13
```

2.11 *if-else control structures*

Sometimes we would want to execute a certain part of the code only if certain condition is satisfied. This condition can be anything from the type of an object (Ex: if object is a matrix execute certain code), or it can be more compicated such as if object value is between certain thresholds. Let us see how they can be used[3]. They can be used anywhere in your code, now we will use it in a function.

[3] see ?Control for more

```
# function takes input one row of CpGi data frame
largeCpGi <- function(bedRow) {
    cpglen <- bedRow[3] - bedRow[2] + 1
    if (cpglen > 1500) {
        cat("this is large\n")
    } else if (cpglen <= 1500 & cpglen > 700) {
        cat("this is normal\n")
    } else {
        cat("this is short\n")
    }
}
largeCpGi(cpgi.df[10, ])
largeCpGi(cpgi.df[100, ])
largeCpGi(cpgi.df[1000, ])
```

2.12 *Loops and looping structures in R*

When you need to repeat a certain task or a execute a function multiple times, you can do that with the help of loops. A loop will execute the task until a certain condition is reached. The loop below is called a "for-loop" and it executes the task sequentially 10 times.

```
for (i in 1:10) {
    # number of repetitions
    cat("This is iteration")  # the task to be repeated
    print(i)
}

## This is iteration[1] 1
## This is iteration[1] 2
## This is iteration[1] 3
## This is iteration[1] 4
## This is iteration[1] 5
## This is iteration[1] 6
## This is iteration[1] 7
```

```
## This is iteration[1] 8
## This is iteration[1] 9
## This is iteration[1] 10
```

The task above is a bit pointless, normally in a loop, you would want to do something meaningful. Let us calculate the length of the CpG islands we read in earlier. Although this is not the most efficient way of doing that particular task, it serves as a good example for looping. The code below will be execute hundred times, and it will calculate the length of the CpG islands for the first 100 islands in the data frame (by subtracting the end coordinate from the start coordinate) [4].

[4] TIP: If you are going to run a loop that has a lot of repetitions, it is smart to try the loop with few repetitions first and check the results. This will help you make sure the code in the loop works before executing it for thousands of times.

```
# this is where we will keep the lenghts for now it
# is an empty vector
result <- c()
# start the loop
for (i in 1:100) {
    # calculate the length
    len <- cpgi.df[i, 3] - cpgi.df[i, 2] + 1
    # append the length to the result
    result <- c(result, len)
}
# check the results
head(result)

## [1]  277  360  383 1325 3498 1602
```

apply family functions instead of loops

R has other ways of repeating tasks that tend to be more efficient than using loops. They are known as the *"apply"* family of functions, which include *apply,lapply, mapply and tapply* (and some other variants). All of these functions apply a given function to a set of instances and returns the result of those functions for each instance. The differences between them is that they take different type of inputs. For example apply works on data frames or matrices and applies the function on each row or column of the data structure. lapply works on lists or vectors and applies a function which takes the list element as an argument. Next we will demonstrate how to use apply() on a matrix. The example applies the sum function on the rows of a matrix, it basically sums up the values on each row of the matrix, which is conceptualized in Figure 2.6

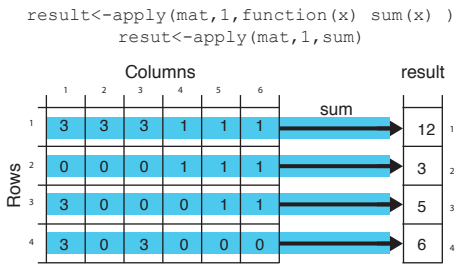

Figure 2.6: apply on rows

```
mat <- cbind(c(3, 0, 3, 3), c(3, 0, 0, 0), c(3, 0,
    0, 3), c(1, 1, 0, 0), c(1, 1, 1, 0), c(1, 1, 1,
    0))
result <- apply(mat, 1, sum)
result

## [1] 12  3  5  6

# OR you can define the function as an argument to
# apply()
result <- apply(mat, 1, function(x) sum(x))
result

## [1] 12  3  5  6
```

Notice that we used a second argument which equals to 1, that indicates that rows of the matrix/ data frame will be the input for the function. If we change the second argument to 2, this will indicate that columns should be the input for the function that will be applied. See Figure 2.7 for the visualization of apply() on columns.

```
result <- apply(mat, 2, sum)
result

## [1] 9 3 6 2 3 3
```

next we will use lapply, which applies a function on a list or a vector. The function that will be applied is a simple function that takes the square of a given number.

```
input <- c(1, 2, 3)
lapply(input, function(x) x^2)

## [[1]]
## [1] 1
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 9
```

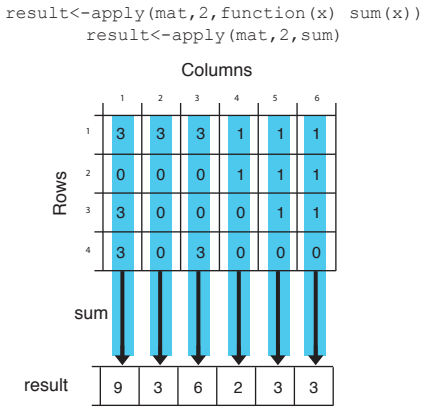

Figure 2.7: apply on columns

mapply is another member of apply family, it can apply a function on an unlimited set of vectors/lists, it is like a version of lapply that can handle multiple vectors as arguments. In this case, the argument to the mapply() is the function to be applied and the sets of parameters to be supplied as arguments of the function. This conceptualized

Figure 2.8, the function to be applied is a function that takes to arguments and sums them up. The arguments to be summed up are in the format of vectors, Xs and Ys. mapply() applies the summation function to each pair in Xs and Ys vector. Notice that the order of the input function and extra arguments are different for mapply.

```
Xs <- 0:5
Ys <- c(2, 2, 2, 3, 3, 3)
result <- mapply(function(x, y) sum(x, y), Xs, Ys)
result

## [1] 2 3 4 6 7 8
```

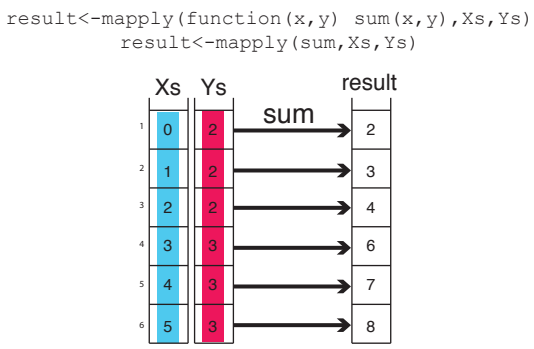

Figure 2.8: mapply takes in multiple/vectors and lists

apply family functions on multiple cores

If you have large data sets apply-family functions can be slow (although probably still better than for loops). If that is the case, you can easily use the parallel versions of those functions from parallel package. These functions essentially divide your tasks to smaller chunks run them on separate CPUs and merge the results from those parallel operations. This concept is visualized at Figure 2.9, mcapply runs the summation function on three different processors. Each processor executes the summation function on a part of the data set, and the results are merged and returned as a single vector that has the same order as the input parameters Xs and Ys.

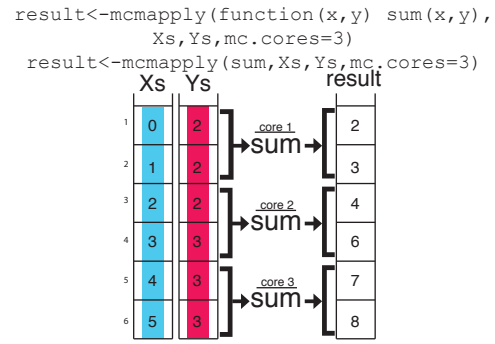

Figure 2.9: mcmapply on 3 cores

Vectorized functions in R

The above examples have been put forward to illustrate functions and loops in R because functions using sum() are not complicated and easy to understand. You will probably need to use loops and looping structures with more complicated functions. In reality, most of the operations we used do not need the use of loops or looping structures because there are already vectorized functions that can achieve the same outcomes, meaning if the input arguments are R vectors the output will be a vector as well, so no need for loops or vectorization.

For example, instead of using mapply() and sum() functions we can just use + operator and sum up Xs and Ys.

```
result <- Xs + Ys
result

## [1] 2 3 4 6 7 8
```

In order to get the column or row sums, we can use the vectorized functions colSums() and rowSums()

```
colSums(mat)

## [1] 9 3 6 2 3 3

rowSums(mat)

## [1] 12  3  5  6
```

However, remember that not every function is vectorized in R, use the ones that are. But sooner or later, apply family functions will come in handy.

2.13 *Session Info*

```
sessionInfo()

## R version 3.0.2 (2013-09-25)
## Platform: x86_64-apple-darwin10.8.0 (64-bit)
##
## locale:
## [1] C
##
## attached base packages:
## [1] parallel  methods   stats     graphics
## [5] grDevices utils     datasets  base
##
## other attached packages:
## [1] codetools_0.2-8       Rsamtools_1.13.49
## [3] GenomicRanges_1.13.51 Biostrings_2.29.19
## [5] XVector_0.1.4         IRanges_1.19.38
## [7] BiocGenerics_0.7.5    knitr_1.5
##
## loaded via a namespace (and not attached):
## [1] bitops_1.0-6   digest_0.6.3   evaluate_0.5
## [4] formatR_0.9    highr_0.2.1    stats4_3.0.2
## [7] stringr_0.6.2  tools_3.0.2    zlibbioc_1.7.0
```

2.14 *Acknowledgements*

Chapter is initiated by Altuna Akalin.