Project 4. EDA and Batch Effect Correction on NGS data

Table of Contents

## The Outline for Project 4

The major objective of Assignment 4 is to replicate the core analysis of the paper published in in 2016. Next-Generation Sequencing (NGS) technology is currently the most popular data type in bioinformatics. Thousands of NGS datasets are stored in published datasets every year to capture the molecular characteristics of DNA, RNA and epigenetics in cells. In this project, our goal is to perform low-level EDA based on RNA-Seq data sets generated from 2 sequencing centers. We will discover the major source of technical variation in this dataset, and we will apply a statistical tool to correct the technical error.

You could check the paper below for more of the background information:

*Love, M. I., Hogenesch, J. B., & Irizarry, R. A. (2016). Modeling of RNA-seq fragment sequence bias reduces systematic errors in transcript abundance estimation. Nature biotechnology, 34(12), 1287.*

## 1. Check the Experiment Design and Data Directories

The data set contains 4 RNA-Seq samples sequenced from 2 centers under the Illumina Hi-Seq 2000 platform. All samples originate from the same cell line and the same biological conditions in the GEUVADIS project. The raw reads have been previously aligned onto the genomic assembly hg19 using STAR, and the resulting BAM files are saved in the working directory of project 4.

The sample information is summarized in the table “samples.csv”, from which we can construct the BAM file directories. Please use the string manipulation function paste0 () to construct the name of the sorted BAM file from the run column of the table, and then usefile.exists ()to check whether all files exist in the current working directory.

## === Hint code, fill the "\_\_\_" ============= ##  
#samples <- read.csv(\_\_\_)  
#bamfiles <- paste0(\_\_\_) #Paste the sample id with "\_sort.bam"  
#file.exists(bamfiles) #Check if the bam files exist or not  
## ===== Enter your code below =============== ##  
samples <- read.csv("/home/yuxuan/BIO214/Project4/samples.csv")  
bamfiles <- paste0(samples$run,"\_sort.bam") #Paste the sample id with "\_sort.bam"  
file.exists(bamfiles) #Check if the bam files exist or not

## [1] TRUE TRUE TRUE TRUE

## 2. Extract the Longest Transcript of the Gene USF2

In the following analysis, we want to investigate the RNA-Seq fragment coverage on the transcript of the gene USF2. First, please extract the exon regions of the longest transcript of gene USF2 on hg19. The expected output in this step is a GRangesList object of length 1, which contains all exons of the transcript. Please use usf2\_longest to name the result variable.

To achieve this, you first need to use the select() function to find the transcript IDs and transcript names associated with the gene USF2. You can use keytypes(Homo.sapiens) and columns(Homo.sapiens) to check all the id conversion options present in the Human organismDb package. The keys are the items that can be converted, and the columns are entries that can be converted to.

USF2 is a gene symbol, so its key type in the organismDb object should be "SY\_\_\_", and the corresponding column should be c("TX\_\_\_", "TX\_\_\_") so that we can retrieve the transcript IDs and the transcript names from the database package.

The output of exonsBy(txdb, by = "tx") is a GRangesList, where each element corresponds to the exons of a transcript. You can use elementNROWS () to access the length of each element in the list object.

## === Hint code, fill the "\_\_\_" ============= ##  
# library(TxDb.\_\_\_.UCSC.\_\_\_.knownGene)  
# library(Homo.sapiens)  
# txdb <- TxDb.\_\_\_.UCSC.\_\_\_.knownGene  
# g <- list()  
# g[["USF2"]] <- select(Homo.sapiens, "USF2", \_\_\_ , \_\_\_)  
# ebt <- exonsBy(txdb, by="tx")  
# head(names(ebt)) #Check the structure of exonsByTranscript, the names are TXIDs  
# usf2 <- ebt[ g[["USF2"]]$TXID ]  
# usf2\_longest <- usf2[which.max(\_\_\_)] #You need to only keep the longest transcript among the isoforms  
## ===== Enter your code below =============== ##  
library(TxDb.Hsapiens.UCSC.hg19.knownGene)  
library(Homo.sapiens)  
txdb <- TxDb.Hsapiens.UCSC.hg19.knownGene  
g <- list()  
g[["USF2"]] <- select(Homo.sapiens, "USF2", columns = c("TXID","TXNAME") , keytype = "SYMBOL")  
ebt <- exonsBy(txdb, by="tx")  
head(names(ebt)) #Check the structure of exonsByTranscript, the names are TXIDs

## [1] "1" "2" "3" "4" "5" "6"

usf2 <- ebt[ g[["USF2"]]$TXID ]  
usf2\_longest <- usf2[which.max(elementNROWS(usf2))] #You need to only keep the longest transcript among the isoforms

* SAQ1: How many transcript isoforms does USF2 have? How many exons are there in each transcript? What are the transcript id and transcript name for the longest transcript of USF2. Is the longest transcript the one with the greatest number of exons?
* Your answer: USF2 has 5 transcript isoforms. There are 9,10,9,8,8 exons in 66826-66830 respectively The transcript id is 66827 and transcript name is uc002nyq.1 for the longest transcript of USF2 Yes, the one with the greatest number of exons is the longest transcript, containing 10 exons

## 3. Load the BAM Files as Fragments on Transcript

Next, we want to write an R function that will take the inputs of the BAM file directory and the GRangesList of the transcript, and it will return the GRanges of fragments “on transcript coordinates” (without introns).

The first step in this function is to read the BAM file using readGAlignmentPairs(). In this step, we need to set the param= argument with ScanBamParam() to define the filters used when reading BAM file: the reads loaded should lie within the scope of the target transcript, all of reads with mapping quality scores less than 30 will be discarded. Please see ?ScanBamParam() on how to set the BAM filtering parameters.

Subsequently, we need to only retain the aligned reads that are compatible to the exons of the transcript. This purpose can be achieved by the function findCompatibleOverlaps(). Then, the coordinates of the loaded reads are based on the genome, so we need to convert them to the coordinate of the transcript using the function map\_\_\_Transcripts().

Please fillin the missing portion of the code chunk below to realize the above mentioned computations in the function readTxFragments().

library (GenomicAlignments)  
readTxFragments <- function(file, transcript) {  
 r <- range(transcript[[1]]) # Get the range of the target transcript on genome  
 r <- keepStandardChromosomes(r) # Only keep the transcript on standard chromosome  
   
 # Read the BAM files with required filters, and find the compatible reads  
 ## === Hint code, fill the "\_\_\_" ============= ##  
 # suppressWarnings({  
 # gap <- readGAlignmentPairs(file, param=ScanBamParam(which = \_\_\_, mapqFilter = \_\_\_))  
 # }) # suppress warnings about alignments with ambiguous pairing  
 # fco <- findCompatibleOverlaps(\_\_\_, \_\_\_)  
 ## ===== Enter your code below =============== ##  
   
 suppressWarnings({  
 gap <- readGAlignmentPairs(file, param=ScanBamParam(which = r, mapqFilter = 30))  
 # gap <- readGAlignmentPairs(file[1], param=ScanBamParam(which = r, mapqFilter = 30))  
 }) # suppress warnings about alignments with ambiguous pairing  
 fco <- findCompatibleOverlaps(gap,transcript)  
   
 ## ===== Your code is finished =============== ##  
 idx <- queryHits(fco) # get index for the compatible reads  
 gr <- as(gap[idx],"GRanges") # subset the reads with index and convert it into GRanges  
 strand(gr) <- "\*" # set the strand into both stands (\*), this will fill the insert and make a pair of reads into 1 range.   
   
 # Convert the paired-end reads into fragments on exons by mapping the ends of the reads to transcript coordinate.  
   
 ## === Hint code, fill the "\_\_\_" ============= ##  
 # m2tx.start <- map\_\_\_Transcripts(resize(\_\_\_, width=1, fix="start"), \_\_\_)  
 # m2tx.end <- map\_\_\_Transcripts(resize(\_\_\_, width=1, fix="end"), \_\_\_)  
 ## ===== Enter your code below =============== ##  
 m2tx.start <- mapToTranscripts(resize(gr, width=1, fix="start"),transcript)  
 m2tx.end <- mapToTranscripts(resize(gr, width=1, fix="end"), transcript)  
 ## ===== Your code is finished =============== ##  
   
 #Flip the start & end if the transcript is on the negative strand.  
   
 tx.strand <- as.character(strand(transcript)[[1]][1])  
 if (tx.strand == "+") {  
 m2tx <- GRanges(seqnames(m2tx.start),  
 IRanges(start(m2tx.start),start(m2tx.end)))  
 } else {  
 m2tx <- GRanges(seqnames(m2tx.start),  
 IRanges(start(m2tx.end),start(m2tx.start)))  
 }  
 return(m2tx)  
}  
  
Fragment\_list <- lapply(bamfiles, readTxFragments, transcript = usf2\_longest)

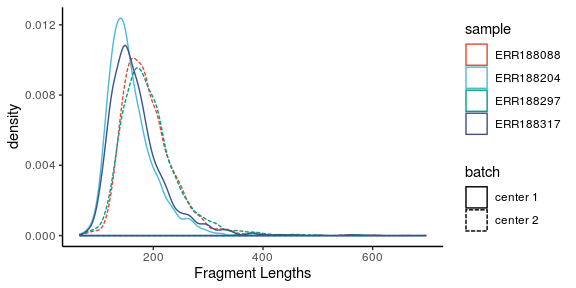
* SAQ2: After iterating the function readTxFragments over the BAM files, check the result variable Fragment\_list. For each sample, determine how many fragments are compatibly aligned to the exons of the target transcript?
* Your answer:

For each sample in “ERR188204\_sort.bam” “ERR188317\_sort.bam” “ERR188297\_sort.bam” “ERR188088\_sort.bam”, the fragment is 2158,2771,2204,2931.

## 4. Plot the Fragment Length Distribution

After obtaining the aligned fragments on the transcript, we want to compare the lengths of the fragments in different samples. In the following code chunk, please manipulate the Fragment\_list with another lapply() function to calculate the fragment lengths for each sample. Then, run the following code to plot the fragment length distributions.

## === Hint code, fill the "\_\_\_" ============= ##  
#Fragment\_len <- lapply(Fragment\_list, \_\_\_)  
## ===== Enter your code below ============== ##  
Fragment\_len <- lapply(Fragment\_list, function(x) width(x))  
## ===== Your code is finished =============== ##  
plot\_df <- data.frame( fragment\_length = unlist( Fragment\_len ),  
 sample = rep(samples$run, elementNROWS(Fragment\_list)),  
 batch = rep(paste0("center ",samples$center), elementNROWS(Fragment\_list)) )  
  
library(ggplot2)  
library(ggsci) #A package for good looking colours  
  
ggplot(plot\_df, aes(x = fragment\_length, color = sample, linetype = batch)) + geom\_density() +  
 theme\_classic() + scale\_color\_npg() + labs(x = "Fragment Lengths")



Next, please calculate the average fragment length of each sample (BAM file), and store the vector in a variable named average\_frag\_len.

## === Hint code, fill the "\_\_\_" ============= ##  
# average\_frag\_len <- sapply(Fragment\_len, \_\_\_ )  
## ===== Enter your code below ============== ##  
average\_frag\_len <- sapply(Fragment\_len, mean)  
mean(average\_frag\_len)

## [1] 178.5196

* SAQ3: Are the fragment lengths calculated in each BAM file fixed? Are the average fragment lengths different between samples? Is the reading length within each sample fixed? Given the read lengths in the BAM files are 76, what are the average “inner distances” of the 4 samples on the target transcript? Please discuss the potential source of variations for the fragment lengths in NGS.
* Your answer:

The fragment lengths calculated in each BAM file are not fixed. The average fragment lengths are different between samples The reading length within sample is not fixed Given the read lengths in the BAM files are 76, the average “inner distances” of the 4 samples on the target transcirpt is 102 The potential source of variations for the fragment lengths in NGS is induced by size selection

P.S. please see this [link](https://www.biostars.org/p/106291/) to help you clarify the differences between fragment, read, insert, and inner distance.

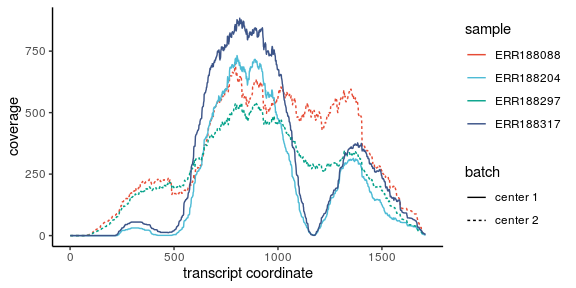
The definition of the fragment in our project actually corresponds to the ‘insert’ region mentioned in the link above. The term insert and fragment are often used interchangeably because the adaptors are removed in order to align the reads to the genome. However, we should aware that the “complete” fragments in 2nd-generation sequencing do have adaptor sequence at the ends.

## 5. Check the Fragment Coverage on the Transcript

In the next step, we want to visualize the “coverage” of the sequenced fragments on the target transcript. Coverage can be understood as the height obtained by stacking all features along the genome/transcript coordinate. In Bioconductor, we can compute the coverage of GRanges using the function coverage().

For each list element of Fragment\_list, calculate the coverage on transcript, and convert the coverage into integer vectors of counts. Store the coverage list in a variable named by coverage\_lst.

## === Hint code, fill the "\_\_\_" ============= ##  
# coverage\_lst <- lapply(\_\_\_, function(x) as.vector(\_\_\_[[1]]))  
## === Enter your code below ================= ##  
coverage\_lst <- lapply(Fragment\_list, function(x) as.vector(coverage(x)[[1]]))  
## === Your code is finished ================ ##  
plot\_df <- data.frame( tx\_position = unlist(lapply(coverage\_lst,seq\_along)),  
 coverage = unlist( coverage\_lst ),  
 sample = rep(samples$run, elementNROWS(coverage\_lst)),  
 batch = rep(paste0("center ",samples$center), elementNROWS(coverage\_lst)) )  
  
ggplot(plot\_df, aes(x = tx\_position, y = coverage, color = sample, linetype = batch)) + geom\_line() +  
 theme\_classic() + scale\_color\_npg() + labs(x = "transcript coordinate")



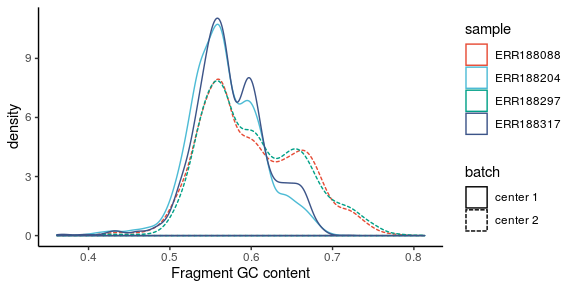
* SAQ4: The figure above shows the coverage of fragments on transcript. Ideally, what distribution should the fragment coverges have on an exon and transcript? Are the observed coverages look like what we expected? What are the possible sources of the variations of the coverages between samples? Please try to discuss some of the potential consequences if we ignore such issue in the downstream analysis.
* Your answer:

Ideally, the coverage of fragments on an exon and transcript should be normally distributed. Most of the observed coverages look like what we expected. But two samples including ERR188317 and ERR 188297 are different. The possible sources of the variations of the coverages between samples are mRNA selection techniques and RNA degradation. In addition, the failure to model fragment GC content bias. The potential consequence if we igore such issuce might impair the transcript abundance estimation, resulting in high false positive rate.

## 6. Explore Fragment level GC Content Distributions

Using the “Fragment\_list” generated previously, extract the GC content of the fragments in 4 samples. The transcript sequence can be retrieved using extractTranscriptSeqs (). Then, apply Views () to extract the sequence underneath the fragment range. Store the GC content of each list element in a variable named GC\_list.

## === Hint code, fill the "\_\_\_" ============= ##  
#library(BSgenome.\_\_\_.UCSC.\_\_\_)  
#usf2.seq <- extractTranscriptSeqs(\_\_\_,\_\_\_)[[1]]  
#GC\_list <- lapply(Fragment\_list,function(x) \_\_\_(\_\_\_(Views(\_\_\_, ranges(x)) ), \_\_\_, as.prob=\_\_\_))  
## === Enter your code below ================= ##  
library(BSgenome.Hsapiens.UCSC.hg19)  
usf2.seq <- extractTranscriptSeqs(Hsapiens,usf2\_longest)[[1]]  
GC\_list <- lapply(Fragment\_list,function(x) letterFrequency(DNAStringSet(Views(usf2.seq, ranges(x)) ), "GC", as.prob=TRUE))  
## === Your code is finished ================ ##  
plot\_df <- data.frame( GC\_content = unlist( GC\_list ),  
 sample = rep(samples$run, elementNROWS(GC\_list)),  
 batch = rep(paste0("center ",samples$center), elementNROWS(GC\_list)) )  
  
ggplot(plot\_df, aes(x = GC\_content, color = sample, linetype = batch)) + geom\_density() +  
 theme\_classic() + scale\_color\_npg() + labs(x = "Fragment GC content")



* SAQ5: After plotting the GC content densities, please discuss the potential cause of the variation observed between batches. Assume the observed bias exists across all the fragments of the library (not restricted to one transcript), will this bias lead to mistakes in the gene expression level quantification? Do you expect to observe a similar phenomenon if the reads are generated by 3rd generation sequencing? Please explain the reason.
* Your answer: The potential cause of the variation observed between batches is sample-specific biases resulting from library preparation (PCR amplification process). This bias will lead to the mistake in gene expression level quantification because slight difference in average GC content across samples lead to difference in quantification for transcripts. The similar phenomenon will not be observed in the 3rd generation sequencing. The bias is caused through the high-throughput sequencing data while 3rd generation sequencing reads nucleotide sequence at the level of a single molecule, rather than the high-throughput sequencing

## 7. Fit the Linear Relationships Between GC and Fragment Coverages

In the previous question, we have explored the “marginal distributions” of fragment GC contents in different samples. We will next attempt to investigate the “joint distributions” between GC and coverages. If the modeling on the joint is good enough, the “offsets” of the coverages can be estimated, which can be used to correct the GC content bias.

To model the relationship between GC and coverage, we first need to identify all potential positions of the sequenced fragments, because many fragments that can potentially existed in the library have 0 counts in our data. The following code chunk will help us generate the coverage for potential fragments.

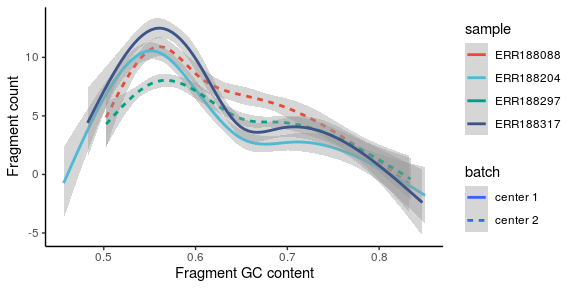
You are required to fill the central step in this procedure, which is to use the function slidingWindows() to generate potential fragments on the coordinate of the target transcript. Set the window size into the average fragment length calculated from each sample, and set the step size to 1L (moving one bp at a time).

#Convert fragment into the end POS and compute their coverages  
POS <- lapply(Fragment\_list, function(x) c(resize(x,1,fix = "start"),  
 resize(x,1,fix = "end")) )  
  
POS\_coverage <- lapply(POS, function(x) as.numeric(coverage(x)[[1]]))  
  
#Get the range of the transcript on its own coordinate  
usf2\_tx <- range(mapToTranscripts(  
 unlist(usf2\_longest),  
 usf2\_longest))   
  
#Find potential fragments for each sample   
  
## === Hint code, fill the "\_\_\_" ============= ##  
#PotentialFragments <- lapply( round(average\_frag\_len),  
# function(x) slidingWindows(\_\_\_,\_\_\_,\_\_\_)[[1]] )  
## === Enter your code below ================= ##  
PotentialFragments <- lapply( round(average\_frag\_len),  
 function(x) slidingWindows(usf2\_tx,x,step = 1L)[[1]] )  
  
## === Your code is finished ================ ##  
  
#Count both ends of the potential fragments  
Fragment\_counts <- mapply(function(x,y) y[start(x)]+y[end(x)],  
 PotentialFragments,  
 POS\_coverage)   
  
#Calculate GC on potential fragments  
GC\_listp <- lapply(PotentialFragments,  
 function(x) letterFrequency(DNAStringSet(   
 Views(usf2.seq, ranges(x)) ),   
 "GC", as.prob=TRUE))  
  
#Build a data.frame for plotting  
plot\_df <- data.frame( GC\_content = unlist( GC\_listp ),  
 Fragment\_count = unlist(Fragment\_counts),  
 sample = rep(samples$run, elementNROWS(GC\_listp)),  
 batch = rep(paste0("center ",samples$center),  
 elementNROWS(GC\_listp)) )

We will next try to fit an ordinary linear regression model using smoothing splines of 5 knots. The covariate (x) and response variable (y) are GC content and end coverage of the potential fragments, respectively.

### Gaussian Regression:

## define the smooth function for ordinary linear regression  
lm\_smooth <- function(...) {  
 geom\_smooth(method = "lm", ...)  
}  
  
ggplot(plot\_df, aes(x = GC\_content, y = Fragment\_count,  
 color = sample, linetype = batch)) +   
 lm\_smooth(formula = y ~ splines::ns(x, 5)) +  
 theme\_classic() + scale\_color\_npg() +   
 labs(x = "Fragment GC content",   
 y = "Fragment count",  
 main = "Gaussian Regression")

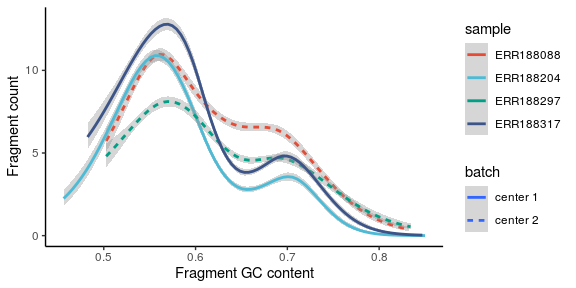


We have obtained the fitted curves between GC and coverage for different samples. Subsequently, we will try an alternative model called Poisson regression. Please fill out the function poisson\_smooth() so that ggplot2 could fit Poisson regression on our data.

P.S. the Poisson regression is a type of generalized linear model. In R, Poisson glm can be called with the function glm() and family = “poisson”. The difference between Poisson regression and ordinary linear regression is that the Poisson regression treats the noises as Poisson distributions, while the OLR teats the noises as zero-center normal distributions.

### Poisson Regression:

poisson\_smooth <- function(...) {  
## === Hint code, fill the "\_\_\_" ============= ##  
#geom\_smooth(method = \_\_\_, method.args = list(family = \_\_\_), ...)  
## === Enter your code below ================= ##  
geom\_smooth(method = "glm", method.args = list(family = "poisson"), ...)  
## === Your code is finished ================ ##   
}  
  
ggplot(plot\_df, aes(x = GC\_content, y = Fragment\_count,  
 color = sample, linetype = batch)) +   
 poisson\_smooth(formula = y ~ splines::ns(x, 5)) +  
 theme\_classic() + scale\_color\_npg() +   
 labs(x = "Fragment GC content",  
 y = "Fragment count",  
 main = "Poisson Regression")

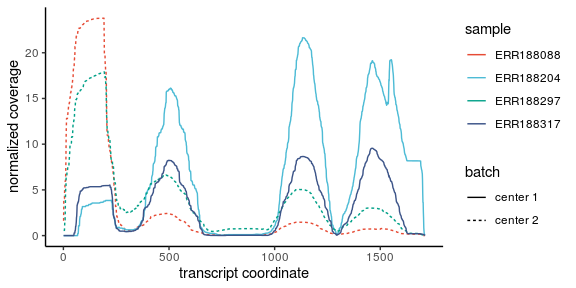


* SAQ6: The grey area around the curves represents the confidence intervals for the estimations of the conditional expectations (regression). From your opinion, which model fits better? Please give some reasons behind your choice of the model. (please think about the nature of the reads coverage datatype.)
* Your answer: Poisson regression fits better because the grey region is thinner, indicating the smaller confidence interval. In other words, Poisson regression can provide more accurate estimation at the same confidence intercal.In addition, the nature of the reads coverage type is that the fragement coverage is extended from both read pairs using the median fragment length and the Possion regression treats the noise as Possion distribution, while OLR treats the noise as zero-center normal distribution. Therefore, based on the previous points, we think Possion regression is better.

## 8. Correct the Fragment Coverage by Straightening the GC content linear effects

Using your selected model, please complete the following code to obtain the predicted values of reads coverage by fragment GC. Then, the fitted values will be used as offsets to conduct normalization on the sequencing data. Hopefully, this measure could help to remove the technical errors induced by GC.

# === Hint code, fill the "\_\_\_" ============= ## This time \_\_\_, could have no content  
#offsets <- mapply(function(x,y){predict(\_\_\_lm(y~splines::ns(x,5)\_\_\_),data.frame(x))},  
# GC\_listp, Fragment\_counts) #mapply() is an iterative functional with 2 inputs, see ?mapply  
## === Enter your code below ================= ##  
offsets <- mapply(function(x,y){predict(glm(y~splines::ns(x,5)),data.frame(x))},  
 GC\_listp, Fragment\_counts) #mapply() is an iterative functional with 2 inputs, see ?mapply  
## === Your code is finished ================ ##  
  
#The read coverage is normalized by dividing the original fragment counts by the exponentiated offsets, which is the fitted value of the linear regression model. The exponetiation is necessary if the prediction is made by poisson GLM which is on the log scale.  
normalized\_counts <- mapply(function(x,y) x/exp(y), Fragment\_counts, offsets)  
  
  
library(zoo)  
  
  
#The normalized coverage is calculated by the rolling sums (rollsum) of the normalized counts, while the ends are computed by cumulative sums (cumsum).  
normalized\_coverages <- mapply(function(x,y)   
 c(cumsum(x[1:y]),rollsum(x,y),  
 rev(cumsum(x[length(x):(length(x)-y)])))/2,  
 normalized\_counts,  
 average\_frag\_len)   
  
#Finally, plot the normalized reads coverage with ggplot2.  
plot\_df <- data.frame( tx\_position = unlist(lapply(normalized\_coverages,seq\_along)),  
 coverage = unlist( normalized\_coverages ),  
 sample = rep(samples$run, elementNROWS(normalized\_coverages)),  
 batch = rep(paste0("center ",samples$center), elementNROWS(normalized\_coverages)) )  
  
ggplot(plot\_df, aes(x = tx\_position, y = coverage, color = sample, linetype = batch)) + geom\_line() +  
 theme\_classic() + scale\_color\_npg() + labs(x = "transcript coordinate", y = "normalized coverage")



* SAQ7: From the corrected coverage plot, does the correction of fragment GC bias reduce the between batch variation? Referring to the work of Love, M. I et.al (2016), could you list some other sources of technical biases in RNA-Seq? What are the potential issues behind the assumptions made when normalizing the technical factors in NGS? (Recall the link between GC and RNA structures as we observed in project 1)
* Your answer: The correction of fragment GC bias does not reduce the bias between batch variation

Some other sources of technical bias in RNA-Seq including fragment sequence bias, GC content bias, sequence-based read start position bias, sample-specific bias, positional bias The potential issues behind the assumptions made by normalizing the technical is that the hybridized GC content might cause different RNA structure, which is significantly different from the looped GC contents, and that situation will lead to the fragment GC contents cannot be normalized.

## Session Info

sessionInfo()

## R version 3.6.3 (2020-02-29)  
## Platform: x86\_64-pc-linux-gnu (64-bit)  
## Running under: Ubuntu 16.04.6 LTS  
##   
## Matrix products: default  
## BLAS: /usr/lib/libblas/libblas.so.3.6.0  
## LAPACK: /usr/lib/lapack/liblapack.so.3.6.0  
##   
## locale:  
## [1] LC\_CTYPE=en\_US.UTF-8 LC\_NUMERIC=C   
## [3] LC\_TIME=en\_US.UTF-8 LC\_COLLATE=en\_US.UTF-8   
## [5] LC\_MONETARY=en\_US.UTF-8 LC\_MESSAGES=en\_US.UTF-8   
## [7] LC\_PAPER=en\_US.UTF-8 LC\_NAME=C   
## [9] LC\_ADDRESS=C LC\_TELEPHONE=C   
## [11] LC\_MEASUREMENT=en\_US.UTF-8 LC\_IDENTIFICATION=C   
##   
## attached base packages:  
## [1] stats4 parallel stats graphics grDevices utils datasets   
## [8] methods base   
##   
## other attached packages:  
## [1] zoo\_1.8-7   
## [2] BSgenome.Hsapiens.UCSC.hg19\_1.4.0   
## [3] BSgenome\_1.54.0   
## [4] rtracklayer\_1.46.0   
## [5] ggsci\_2.9   
## [6] ggplot2\_3.2.1   
## [7] GenomicAlignments\_1.22.1   
## [8] Rsamtools\_2.2.1   
## [9] Biostrings\_2.54.0   
## [10] XVector\_0.26.0   
## [11] SummarizedExperiment\_1.16.1   
## [12] DelayedArray\_0.12.2   
## [13] BiocParallel\_1.20.1   
## [14] matrixStats\_0.55.0   
## [15] Homo.sapiens\_1.3.1   
## [16] org.Hs.eg.db\_3.10.0   
## [17] GO.db\_3.10.0   
## [18] OrganismDbi\_1.28.0   
## [19] TxDb.Hsapiens.UCSC.hg19.knownGene\_3.2.2  
## [20] GenomicFeatures\_1.38.1   
## [21] AnnotationDbi\_1.48.0   
## [22] Biobase\_2.46.0   
## [23] GenomicRanges\_1.38.0   
## [24] GenomeInfoDb\_1.22.0   
## [25] IRanges\_2.20.2   
## [26] S4Vectors\_0.24.3   
## [27] BiocGenerics\_0.32.0   
##   
## loaded via a namespace (and not attached):  
## [1] httr\_1.4.1 splines\_3.6.3 bit64\_0.9-7   
## [4] assertthat\_0.2.1 askpass\_1.1 BiocManager\_1.30.10   
## [7] BiocFileCache\_1.10.2 RBGL\_1.62.1 blob\_1.2.1   
## [10] GenomeInfoDbData\_1.2.2 yaml\_2.2.1 progress\_1.2.2   
## [13] pillar\_1.4.3 RSQLite\_2.2.0 lattice\_0.20-41   
## [16] glue\_1.4.0 digest\_0.6.25 colorspace\_1.4-1   
## [19] htmltools\_0.4.0 Matrix\_1.2-18 XML\_3.99-0.3   
## [22] pkgconfig\_2.0.3 biomaRt\_2.42.0 zlibbioc\_1.32.0   
## [25] purrr\_0.3.4 scales\_1.1.0 tibble\_2.1.3   
## [28] openssl\_1.4.1 farver\_2.0.3 withr\_2.2.0   
## [31] lazyeval\_0.2.2 magrittr\_1.5 crayon\_1.3.4   
## [34] memoise\_1.1.0 evaluate\_0.14 graph\_1.64.0   
## [37] tools\_3.6.3 prettyunits\_1.1.1 hms\_0.5.3   
## [40] lifecycle\_0.2.0 stringr\_1.4.0 munsell\_0.5.0   
## [43] compiler\_3.6.3 rlang\_0.4.4 grid\_3.6.3   
## [46] RCurl\_1.98-1.2 rappdirs\_0.3.1 labeling\_0.3   
## [49] bitops\_1.0-6 rmarkdown\_2.1 gtable\_0.3.0   
## [52] DBI\_1.1.0 curl\_4.3 R6\_2.4.1   
## [55] knitr\_1.28 dplyr\_0.8.4 bit\_1.1-15.1   
## [58] stringi\_1.4.5 Rcpp\_1.0.3 vctrs\_0.2.2   
## [61] dbplyr\_1.4.2 tidyselect\_1.0.0 xfun\_0.12