primaryTranscriptAnnotation

Warren Anderson, Mete Civelek, Michael Guertin

2019-06-03

This vignette describes basic usage of the primaryTranscriptAnnotation package. The package is currently available on github.com.

1. Overview

This package was developed for two purposes: (1) to generate data-driven transcript annotations based on nascent transcript sequencing data, and (2) to annotate de novo identified transcriptional units (e.g., genes and enhancers) based on existing annotations such as those from (1). The code for this package was employed in analyses underlying our manuscript in preparation: Transcriptional mechanisms and gene regulatory networks underlying early adipogenesis.

For these analyses, we used bigWig format files derived from aggregating the reads across 7 timepoints during the initial phase of 3T3-L1 cell differentiation into adipocytes. For analyses such as those indicated below, we recommend aggregating the data from all of the conditions for which the derived coordinates will be applied in analysis.

The user should be mindful to organize data objects as shown in this vignette (e.g., data frame and list names), otherwise some functions may not run.

Annotating nascent transcripts from run-on data can be a nuanced process. The functions underlying this package are designed to address discrete problems that arise in the process of transcript annotation. This framework reflects a tradeoff between flexibility and simplicity, in which we lean towards the former. The general pipeline illustrated in sections 6 and 7 contains several instances of recommendation for manual analysis and refinements of the annotations based in inspection of various performance metrics and quality checks. Such a process is recommended for imparting intuition and confidence to the analyst.

2. Installation

```r
# install the package
library(devtools)
install_github("WarrenDavidAnderson/genomicsRpackage/primaryTranscriptAnnotation")
```

3. Dependencies

The primaryTranscriptAnnotation R package has the following dependencies that should be loaded along with the package itself. Here are web resources for package install:

dplyr: link

pracma: link

bigWig: link

```r
library(dplyr)
library(pracma)
library(bigWig)
library(denovoGeneAnnotation)
```

4. Data included in the package

We include sample data related to our studies of murine adipogenesis to illustrate the use of this package. Here are some of the data frames. We include subsets of raw data for illustrations of the analyses and we also include the results of particular analyses that take several hours so that the user can reproduce our analysis pipeline in a streamlined fashion.

```
# GENCODE gene annotations
head(gencode.bed)
#>    chr   start     end gene xy strand
#> 1 chr1 3205901 3671498 Xkr4 na      -
#> 2 chr1 3205901 3216344 Xkr4 na      -
#> 3 chr1 3213609 3216344 Xkr4 na      -
#> 4 chr1 3205901 3207317 Xkr4 na      -
#> 5 chr1 3206523 3215632 Xkr4 na      -
#> 6 chr1 3213439 3215632 Xkr4 na      -

# Inferred gene coordinates
head(bed.tss.tts)
#>     chr     start       end          gene    xy strand
#> 1 chr18  38250239  38270040 0610009O20Rik 65964      +
#> 2 chr19   3708303   3730423 1810055G02Rik 49793      +
#> 3 chr17  46772803  46777397 2310039H08Rik     0      +
#> 4 chr18  24470814  24483170 2700062C07Rik     0      +
#> 5  chr3  88685787  88716816 2810403A07Rik 96108      +
#> 6  chr4 108780423 108830806 3110021N24Rik 47628      +

# groHMM-derived transcription unit coordinates
head(hmm.bed)
#>    chr   start     end names scores strand
#> 1 chr1 3010249 3010399     .      .      +
#> 2 chr1 3021349 3033799     .      .      +
#> 3 chr1 3042299 3053499     .      .      +
#> 4 chr1 3077049 3087399     .      .      +
#> 5 chr1 3094599 3108649     .      .      +
#> 6 chr1 3118999 3161849     .      .      +
```

We parsed the data in gencode.bed, reprsenting the coordinates for multiple gene forms, from a comprehensive GENCODE annotation (link). We downloaded that file corresponding to chrom.sizes (R object available with the package) directly from the UCSC browser website (link).

```
# get gencode comprehensive annotations
# https://www.gencodegenes.org/mouse_releases/current.html
# https://www.gencodegenes.org/data_format.html
# gene_type https://www.gencodegenes.org/gencode_biotypes.html
wget ftp://ftp.ebi.ac.uk/pub/databases/gencode/Gencode_mouse/release_M18/gencode.vM18.annotation.gtf.gz
gunzip gencode.vM18.annotation.gtf

# get 'comprehensive' protein coding genes
grep 'gene_type "protein_coding"' gencode.vM18.annotation.gtf | cut -f1,4,5,7,9 | \
tr ";" "\t" > vm18.annotation_unfilt1.bed
grep 'gene_id' vm18.annotation_unfilt1.bed > vm18.annotation_unfilt2.bed
grep 'gene_name' vm18.annotation_unfilt2.bed > vm18.annotation_protcode.bed
awk '{for(i=5;i<=NF;i++){if($i~/^gene_name/){a=$(i+1)}} print $1,$2,$3,a,"na",$4}' \
vm18.annotation_protcode.bed | tr " " "\t" > gencode.vM18.annotation.bed
```

```
rm *unfilt*

# get mm10 chromosome sizes
wget http://hgdownload.cse.ucsc.edu/goldenPath/mm10/bigZips/mm10.chrom.sizes
```

The package also includes PRO-seq read data in the bigWig format (subsampled from the original data set). The PRO-seq data used for this analysis consist of plus and minus bigWig files with reads aggregated across all pre-adipocyte time points (t0 - 4hr post application of an adipogenic cocktail). These data can be loaded as follows, using the bigWig library:

```
plus.file = system.file("data", "preadip_plus_scaled_merged.bigWig",
                        package = "denovoGeneAnnotation")
minus.file = system.file("data", "preadip_minus_scaled_merged.bigWig",
                         package = "denovoGeneAnnotation")
bw.plus = load.bigWig(plus.file)
bw.minus = load.bigWig(minus.file)
```

5. Quick start

Here are basic uses of some of the key package functions. Further details that should be considered for primary transcript annotation are described below.

I will revisit this, some functions need error handling.

```
# get intervals for furthest TSS and TTS +/- interval
bed.long = get.largest.interval(bed=gencode.bed)

# get read counts and densities at the end of each annotated gene
fraction.end = 0.1
add.to.end = 0
end.reads = read.count.end(bed=bed.long, bw.plus=bw.plus, bw.minus=bw.minus,
                           fraction.end=fraction.end, add.to.end=add.to.end)

# specify which genes to cut based on low expression
den.cut = -6
cnt.cut = 3
ind.cut.den = which(log(end.reads$density) < den.cut)
ind.cut.cnt = which(log(end.reads$counts) < cnt.cut)
ind.cut = union(ind.cut.den, ind.cut.cnt)

# remove "unexpressed" genes
unexp = names(end.reads$counts)[ind.cut]
bed.long.filtered = bed.long[!(bed.long$gene %in% unexp),]

# select the TSS for each gene
bp.range = 1200
bp.delta = 10
bp.bin = 50
delta.tss = 50
cnt.thresh = 5
TSS.gene.partial = get.TSS(bed=gencode.bed[1:20000,], bw.plus=bw.plus,
                   bw.minus=bw.minus, bp.range=bp.range, bp.delta=bp.delta,
                   bp.bin=bp.bin, delta.tss=delta.tss, cnt.thresh=cnt.thresh)
```

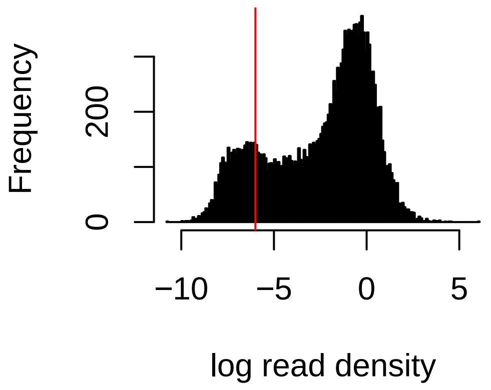
```r
# apply TSS coordinates to the expression-filtered long gene annotations
in.tss.remove = which(TSS.gene == 0)
tss.remove = names(TSS.gene)[in.tss.remove]
bed.long.filtered = bed.long.filtered[!(bed.long.filtered$gene %in% tss.remove),]
bed.long.filtered.tss = apply.TSS.coords(bed=bed.long.filtered, tss=TSS.gene)

# run overlap analysis, remove overlap genes
overlap.data = gene.overlaps( bed = bed.long.filtered.tss )
has.start.inside = overlap.data$has.start.inside
is.a.start.inside = overlap.data$is.a.start.inside
overlap.genes = c(has.start.inside$gene, is.a.start.inside$gene) %>% unique
bed.long.filtered.tss = bed.long.filtered.tss[!(bed.long.filtered.tss$gene %in% overlap.genes),]

# get intervals for TTS evaluation
add.to.end = 100000
fraction.end=0.1
dist.from.start=50
bed.for.tts.eval = get.end.intervals(bed=bed.long.filtered.tss,
                                     add.to.end=add.to.end,
                                     fraction.end=fraction.end,
                                     dist.from.start=dist.from.start)

# check for problems and modify accordingly
full.tts.eval = apply.TSS.coords(bed=bed.for.tts.eval, tss=TSS.gene)
ind.problem1 = which(full.tts.eval$end < full.tts.eval$start)
ind.problem2 = which(full.tts.eval$end < 0)
ind.problem3 = which(full.tts.eval$start < 0)
ind.problem = c(ind.problem1,ind.problem2,ind.problem3,ind.problem4) %>% unique
full.tts.eval = full.tts.eval[-ind.problem,]
bed.for.tts.eval = bed.for.tts.eval[!(bed.for.tts.eval$gene %in% full.tts.eval$gene),]

# identify gene ends
bed.for.tts.eval = bed.for.tts.eval[bed.for.tts.eval$xy != "na",]
bed.for.tts.eval$xy = as.numeric(bed.for.tts.eval$xy)
add.to.end = max(bed.for.tts.eval$xy)
knot.div = 40
pk.thresh = 0.05
bp.bin = 50
knot.thresh = 5
cnt.thresh = 5
tau.dist = 10000
frac.max = 1
frac.min = 0.2
gene.ends.partial = get.gene.end(bed=bed.for.tts.eval[c(1:200,12000:12200),],
                         bw.plus=bw.plus, bw.minus=bw.minus,
                         bp.bin=bp.bin, add.to.end=add.to.end, knot.div=knot.div,
                         pk.thresh=pk.thresh, knot.thresh=knot.thresh,
                         cnt.thresh=cnt.thresh, tau.dist=tau.dist,
                         frac.max=frac.max, frac.min=frac.min)
```

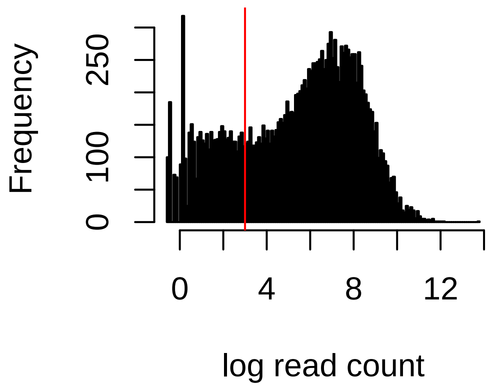
6. Data-driven gene annotation

(a) Comprehensive annotation and removal of unexpressed genes

Here we describe the initial procedures for defining gene boundries with the use of both existing annotations and PRO-seq data. First we perform some simple processing steps. We get comprehensive gene annotations using get.largest.interval() and we count reads at gene ends using read.count.end(). We obtain the largest intervals by taking the minimal start coordinates and maximal end coordinates from the bed-format annotation. To evaluate reads at gene ends, we select a fraction of the annotated gene end (fraction.end = 0.1) and we consider an additional number of base pairs beyond the gene end within which to count reads (add.to.end; we set this to zero). For more details, see the function documentation. We visualize the gene end read density and read count distributions and we define cutoffs below which gene expression is considered negligible. We remove such “unexpressed” genes from further downstream analysis.

```
#####################################################################################
## get the largest interval for each gene
#####################################################################################

# function documentation
?get.largest.interval
?read.count.end

# get intervals for furthest TSS and TTS +/- interval
bed.long = get.largest.interval(bed=gencode.bed)

#####################################################################################
## filter annotations to remove unexpressed genes
#####################################################################################

# get read counts and densities at the end of each annotated gene
fraction.end = 0.1
add.to.end = 0
end.reads = read.count.end(bed=bed.long, bw.plus=bw.plus, bw.minus=bw.minus,
                           fraction.end=fraction.end, add.to.end=add.to.end)

# specify which genes to cut based on low expression
den.cut = -6
cnt.cut = 3
ind.cut.den = which(log(end.reads$density) < den.cut)
ind.cut.cnt = which(log(end.reads$counts) < cnt.cut)
ind.cut = union(ind.cut.den, ind.cut.cnt)
par(mfrow=c(1,2))
hist(log(end.reads$density), breaks=200, col="black",xlab="log read density",main="")
abline(v=den.cut, col="red")
hist(log(end.reads$counts), breaks=200, col="black",xlab="log read count",main="")
abline(v=cnt.cut, col="red")
```





```
# remove "unexpressed" genes
unexp = names(end.reads$counts)[ind.cut]
bed.long.filtered = bed.long[!(bed.long$gene %in% unexp),]
```

(b) Data-driven inference of transcription start sites (TSSs)

Here we describe the methods for inferring a single TSS for each gene. We implement this analysis by setting a range around each annotated gene start within which we search for a region of peak read density. Such regions of peak read density exist at 'pause sites' that typically occur 20-80 bp from the TSS. Within each region around an annotated gene start, we translate a sliding window throughout to find the sub-region with maximal density. We determine the window with the maximal density out of all regions considered and defined the TSS as the upstream boundry of this window, with an additional shift to account for the position of the pause site relative to the TSS.

For our analysis, we set our search regions to 1.2 kb so that we inspected 600 bp on either side of each annotated gene start from the comprehensive annotation (bp.range = 1200). We counted reads in 50 bp bins where the sliding window was moved in 10 bp increments (bp.bin = 50, bp.delta = 10). Finally, once the bin with maximal density was obtained, we set the opporationally defined TSS to 50 bp upstream from the upstream boundry of the bin (delta.tss = 50). In this context, 'upstream' refers to smaller 'start' coordinates for the plus strand and larger 'end' cooordinates for the minus strand.

Note this step takes a long time and the user can perform a 'time test' as indicated in comments.

```
#########################################################################################
## get a single TSS for each gene
#########################################################################################

# select the TSS for each gene
bp.range = 1200
bp.delta = 10
bp.bin = 50
delta.tss = 50
cnt.thresh = 5
TSS.gene.partial = get.TSS(bed=gencode.bed[1:5000,], bw.plus=bw.plus,
                   bw.minus=bw.minus, bp.range=bp.range, bp.delta=bp.delta,
                   bp.bin=bp.bin, delta.tss=delta.tss, cnt.thresh=cnt.thresh)

# time test
# test = dat0[1:1000,]
# start_time <- Sys.time()
# TSS.gene = get.TSS(bed=test, bw.plus=bw.plus, bw.minus=bw.minus,
# bp.range=bp.range, bp.delta=bp.delta, bp.bin=bp.bin, delta.tss=delta.tss)
# end_time <- Sys.time()
```

The analysis implemented by get.TSS() can take several hours, so for illustrative purposed we only run it for 5000 lines of the gene annotation file. We include the results for the complete analysis in this package (TSS.gene).

For this analysis, if the number of counts in the bin with maximal counts was less than the number specified by the parameter cnt.thresh, we flag the TSS with a 0 as an indicator of the absence of gene expression. We set cnt.thresh = 5 and removed such genes.

After identifying the TSSs, we apply the TSS coordinates to the existing annotations using apply.TSS.coords():

```
########################################################################################
## add proper TSS to the expression-filtered long annotations
########################################################################################

# note that genes with less counts than cnt.thresh in the max count region
# will be assigned a TSS of 0; we quantify and remove such genes here
in.tss.remove = which(TSS.gene == 0)
tss.remove = names(TSS.gene)[in.tss.remove]
bed.long.filtered = bed.long.filtered[!(bed.long.filtered$gene %in% tss.remove),]

# apply TSS coordinates to the expression-filtered long gene annotations
bed.long.filtered.tss = apply.TSS.coords(bed=bed.long.filtered, tss=TSS.gene)
```

Next we re-identify TSSs for problematic cases. We observed some cases in which the bed start was larger than the bed end coordinate. For these cases, we reset the range around each annotated gene start to 100 bp and moved the sliding window in 5 bp increments. Re-defining the problematic TSSs according to this analysis remediated the end < start coordinate problem for all but four cases (out of 111). In these four cases, the TSSs were flagged for removal based on low expression and the corresponding genes were removed from analysis.

```
########################################################################################
## Adjust problematic TSSs
########################################################################################

# check for problems (end > start) and remove such cases
# save gene list for re-defining TSS
ind.problem = which(bed.long.filtered.tss$end < bed.long.filtered.tss$start)
genes.redefine = bed.long.filtered.tss[ind.problem,] %>% select(gene) %>%
  as.matrix %>% as.character
bed.long.filtered.tss = bed.long.filtered.tss[-ind.problem,]

# re-define TSS for abberant genes with a smaller window
# be sure to use the same delta.tss value as above
# genes that do not meet the count threshold are removed
bed.redefine = gencode.bed[gencode.bed$gene %in% genes.redefine,]
bp.range = 100
bp.bin = 50
bp.delta = 5
delta.tss = 50
cnt.thresh = 5
TSS.gene.redefine = get.TSS(bed=bed.redefine, bw.plus=bw.plus, bw.minus=bw.minus,
                            bp.range=bp.range, bp.bin=bp.bin, bp.delta=bp.delta,
                            delta.tss=delta.tss, cnt.thresh=cnt.thresh)
```

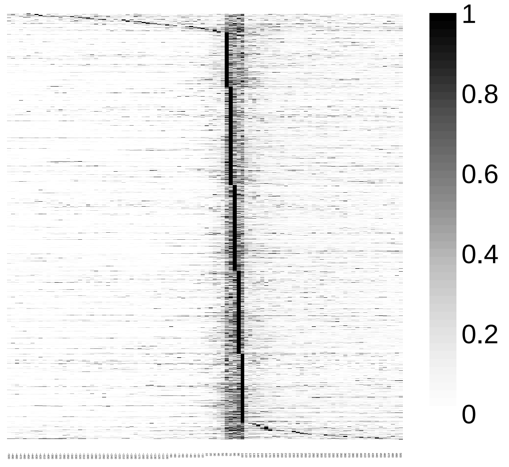
```r
bed.redefine0 = bed.long[bed.long$gene %in% genes.redefine,]
bed.redefine = apply.TSS.coords(bed=bed.redefine0, tss=TSS.gene.redefine)
rem.minus = which(bed.redefine$strand=="-" & bed.redefine$end==0)
rem.plus = which(bed.redefine$strand=="+" & bed.redefine$start==0)
bed.redefine = bed.redefine[-c(rem.minus, rem.plus),]
ind.problem = which(bed.redefine$end < bed.redefine$start)
length( ind.problem ) # no more problem cases
#> [1] 0
bed.long.filtered.tss = rbind(bed.long.filtered.tss, bed.redefine)

# update the TSS annotation
tss.inds = sapply(names(TSS.gene.redefine),function(x){which( names(TSS.gene) == x )})
TSS.gene[tss.inds] = TSS.gene.redefine
```

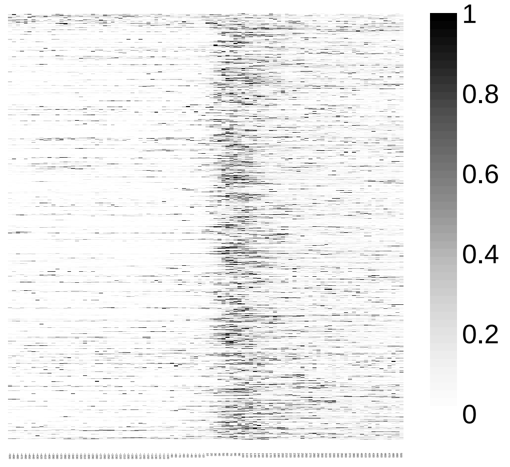
Here the user can evaluate the performance of TSS inference. We specified a region of 1 kb, centered on the TSS, for this analysis (i.e., 500 bp on either side, window). We obtained read counts in 10 bp bins that span this window (bp.bin). We sorted the genes based on the bin with the maximal reads and we scale the data to the interval (0,1) for visualization. We computed the distances between the TSS and the bin with the max reads within the specified window. We used the TSS.count.dist() function to get reads around TSSs and compute distances from read peaks to TSSs.

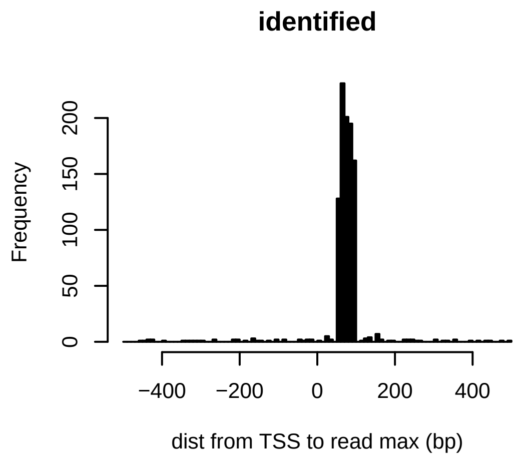
We performed this analysis for both the identified coordinates and the long gene coordinates. We plotted heatmaps sorted based on the max gene bins for both data sets and we show histograms for distances from the TSS to the max bin. The data show that the majority of the identified TSSs are close proximity to the region of peak reads. The distances were more extended for the reference gene annotations.

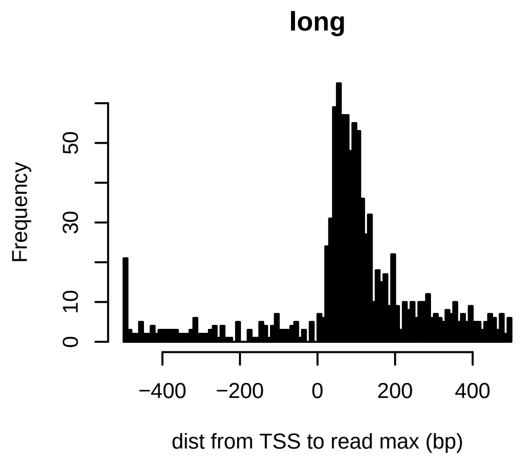
```r
# subsample data
new.bed = bed.tss.tts[c(1:500,11500:12000),]
old.bed = bed.long[bed.long$gene %in% new.bed$gene,]

# parameters
window = 1000
bp.bin = 10

# look at read distribution around identified TSSs
tss.dists = TSS.count.dist(bed=new.bed, bw.plus=bw.plus, bw.minus=bw.minus,
                           window=window, bp.bin=bp.bin)



# look at read distribution around 'long gene' annotation TSSs
tss.dists.lng = TSS.count.dist(bed=old.bed, bw.plus=bw.plus, bw.minus=bw.minus,
                               window=window, bp.bin=bp.bin)

# plot data based on organization of the identified TSSs
library(NMF)
library(RColorBrewer)
id.inds = sapply(names(tss.dists$dist),function(x){which(names(tss.dists.lng$dist)==x)})
bk = seq(-window/2, window/2, bp.bin)
par(mfrow=c(3,2))
aheatmap(tss.dists$scaled,Colv=NA,Rowv=NA,color=brewer.pal(9,"Greys"))
aheatmap(tss.dists.lng$scaled[id.inds,],Colv=NA,Rowv=NA,color=brewer.pal(9,"Greys"))
hist(tss.dists$dist,main="identified",xlab="dist from TSS to read max (bp)",
     breaks=bk,col="black")
hist(tss.dists.lng$dist,main="long",xlab="dist from TSS to read max (bp)",
     breaks=bk,col="black")
```

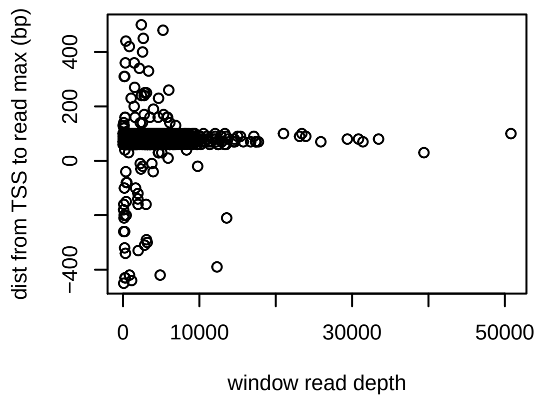
```
plot(rowSums(tss.dists$raw), tss.dists$dist, xlab="window read depth",
     ylab="dist from TSS to read max (bp)")
plot(rowSums(tss.dists.lng$raw), tss.dists.lng$dist, xlab="window read depth",
     ylab="dist from TSS to read max (bp)")
```

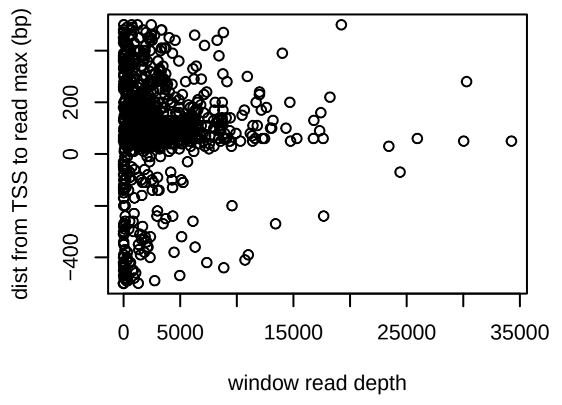












(c) Filtering gene annotations for overlapping genes

Because the comprehensive gene annotations contain instances in which genes overlap, we perform a combination of programatic and manual filtering steps described below. First, we consider cases in which there are multiple identical bed starts and/or bed ends for particular genes. We identify such cases and manually inspect the results using the UCSC genome browser: link. For identified overlaps, we either removed genes or marked cases in which pairs of genes appeared to be adjacent (fix.genes.id). The later were set aside for subsequent analysis. For cases in which multiple genes had overlapping annotations, we attempted to remove the least dominant annotation(s), annotations that did not appear to be expressed, or small genes that could not be distinguished within broader annotations. For such 'small genes', the broad annotations correspond to a broad range of reads so that the smaller annotation could not be unambiguously associated with a distinct transcript. This analysis revealed 141 genes with at least one matching bed start or end. Ninty one of these genes were removed from all subsequent analyses and nine pairs of genes were set aside for analyses of adjacent gene pairs (see below).

We include the results of our manual analysis within this package (remove.genes.id, fix.genes.id, remove.genes.ov, fix.genes.ov). These results will be used below.

First we identify start/end duplicate cases. In our original analysis, we manually visualized these gene in the UCSC browser and we generated the results in remove.genes.id and fix.genes.id used here. We remove the genes from remove.genes.id and we subsequently address the genes in fix.genes.id.

```
########################################################################################
## check for genes with identical annotations, manually curate
########################################################################################

# identical start coordinates
dup.start = bed.long.filtered.tss[(duplicated(bed.long.filtered.tss[,1:2]) |
                                     duplicated(bed.long.filtered.tss[,1:2],
                                                fromLast=TRUE)),]
dup.start = dup.start[order(dup.start$chr,dup.start$start,dup.start$end),]

# identical end coordinates
dup.end = bed.long.filtered.tss[(duplicated(bed.long.filtered.tss[,c(1,3)]) |
                                   duplicated(bed.long.filtered.tss[,c(1,3)],
                                              fromLast=TRUE)),]
dup.end = dup.end[order(dup.end$chr,dup.end$start,dup.end$end),]

# remove genes based on manual analysis
genes.remove1 = remove.genes.id$remove
genes.remove1 = c(genes.remove1, fix.genes.id$upstream)
genes.remove1 = c(genes.remove1, fix.genes.id$downstream)
bed.long.filtered2.tss = bed.long.filtered.tss[
  !(bed.long.filtered.tss$gene %in% genes.remove1),]

# verify the absence of identical coordinates
# both of these frames are and should be empty
dup.start = bed.long.filtered2.tss[(duplicated(bed.long.filtered2.tss[,1:2]) |
                                      duplicated(bed.long.filtered2.tss[,1:2],
                                                 fromLast=TRUE)),]
dup.end = bed.long.filtered2.tss[(duplicated(bed.long.filtered2.tss[,c(1,3)]) |
                                    duplicated(bed.long.filtered2.tss[,c(1,3)],
                                               fromLast=TRUE)),]
```

Next we address overlaps that do not involve identical coordinates using gene.overlaps(). Note that this function separates the documentation of genes found within other genes (is.a.start.inside) and genes with boundries that encompass other genes (has.start.inside). We implement the overlap analysis to determine genes in which other genes were found, as well as genes that were found within other genes. There were a total of 582 unique overlap genes. We removed genes that contained “AC”, “RP”, or “Gm” in their identifiers. These were typically transcript variants, processed pseudogenes, predicted genes, or otherwise poorly annotated genes. One hundred fifty five such genes were flagged for removal, of which 144 were “RP” genes (e.g., RP23-461O23.5), 10 were “AC” genes (e.g., AC155937.10) and one was a “Gm” gene (Gm11084). We eliminated any ’large’ genes that enclosed the initiation of two or more genes. We removed 44 such ’large gene’ cases. After removing 170 genes, there were 91 remaining cases of overlapping gene pairs that we analyzed manually (see remove.genes.ov and fix.genes.ov).

```
#########################################################################################
## address overlaps
#########################################################################################

# run overlap analysis
overlap.data = gene.overlaps( bed = bed.long.filtered2.tss )
has.start.inside = overlap.data$has.start.inside
is.a.start.inside = overlap.data$is.a.start.inside

# filter overlap data to remove non-coding genes
over.genes = c(has.start.inside$gene, is.a.start.inside$gene) %>% unique
genes.remove2 = over.genes[grep("AC",over.genes)]
genes.remove2 = c(genes.remove2, over.genes[grep("RP",over.genes)])
genes.remove2 = c(genes.remove2, over.genes[grep("Gm",over.genes)])

# identify genes with multiple starts inside (i.e., 'large' genes)
# and set these genes for removal
vec = is.a.start.inside$xy
mult.inside.starts = vec[duplicated(vec) | duplicated(vec, fromLast=TRUE)] %>% unique
mult.inside.starts = sapply(mult.inside.starts,function(x){
  strsplit(x,"inside ")[[1]][2]}) %>% unname
genes.remove2 = c(genes.remove2, mult.inside.starts) %>% unique

# remove filtered genes and re-run overlap analysis
overlap.data = gene.overlaps( bed = bed.long.filtered2.tss[
  !(bed.long.filtered2.tss$gene %in% genes.remove2),] )
has.start.inside = overlap.data$has.start.inside
is.a.start.inside = overlap.data$is.a.start.inside

# remove genes based on manual curation
# note that the 'noremove' or fix genes are addressed below
genes.remove2 = c(genes.remove2, remove.genes.ov$remove)
genes.remove2 = c(genes.remove2, fix.genes.ov$upstream)
genes.remove2 = c(genes.remove2, fix.genes.ov$downstream) %>% unique
bed.long.filtered3.tss = bed.long.filtered2.tss[
  !(bed.long.filtered2.tss$gene %in% genes.remove2),]

# verify the absence of overlaps
overlap.data = gene.overlaps( bed = bed.long.filtered3.tss )
has.start.inside = overlap.data$has.start.inside
is.a.start.inside = overlap.data$is.a.start.inside
```

NULL dimensions for has.start.inside and is.a.start.inside confirms that there are not overlaps in the gene set.

```
dim(has.start.inside)
#> NULL
dim(is.a.start.inside)
#> NULL
```

Our manual analysis was caried out by visualising the overlapping gene pairs in the UCSC genome browser and searching for the genes that had nearly identical annotations using Google. For our manual analysis of overlaping gene pairs, we removed genes for the following reasons (n cases): the smaller gene did not show distinguishable expression (n=23), there was low overlap between annotations and one did not appear to be expressed (n=21), no mitigating information was available and the annotations were nearly identical (arbitrary decision to remove one of two genes, n=13), the gene that was either less common or had no documented role in adipose biology (n=9), the gene was relatively large and did not appear to be expressed (n=7), or both genes in the pair were removed because there was an apparent absence of expression (n=6). In addition to the genes removed, we flagged 16 gene pairs in which both adjacent genes appeared to be expressed. These 16 gene pairs are examined further below. For gene annotation overlaps that are identified as adjacent genes, the respective TSSs and TTS can be identified using adjacent.gene.tss() and adjacent.gene.tts().

We considered the adjacent gene pairs with overlaps that were identified based on manual curation. We empirically defined TSSs for these genes by binning the region spanned by both genes, fitting smooth spline curves to the binned read counts, and identifying the two largest peaks separated by a specified distance. We set a 10 bp bin size (bp.bin) and applied the constraint that the identified peaks must be >2 kb apart (diff.tss). We also modified the search region to include 100 bp upstream of the upstream-most gene (shift.up). As described above for TSS identification, we shifted the identified peaks upstream by 50 bp as the operational definition of the TSS (delta.tss). For the spline fits, we set the number of knots to the number of bins divided by 40 (knot.div). The analysis results were very robust to variations in the afformentioned parameters. Plots of read counts, spline fits, and TSS locations are printed to an output file for visual inspection.

```
########################################################################################
## address adjacent gene pairs that were manually identified
########################################################################################

# get the start sites for adjacent gene pairs
fix.genes = rbind(fix.genes.id, fix.genes.ov)
bp.bin = 10
knot.div = 40
shift.up = 100
delta.tss = 50
diff.tss = 2000
TSS.adjacent = adjacent.gene.tss(fix.genes=fix.genes, bed.long=bed.long,
                                 bw.plus=bw.plus, bw.minus=bw.minus,
                                 knot.div=knot.div, bp.bin=bp.bin,
                                 shift.up=shift.up, delta.tss=delta.tss,
                                 diff.tss=diff.tss, fname="adjacentTSS.pdf")

# update TSS information
tss.inds = sapply(TSS.adjacent$gene,function(x){which(names(TSS.gene)==x)})
TSS.gene[tss.inds] = TSS.adjacent$TSS

# get the end sites for adjacent upstream genes, integrate with TSSs
dist.from.start  = 50
adjacent.tts.up = adjacent.gene.tts(fix.genes=fix.genes, bed.long=bed.long,
                                    TSS.adjacent=TSS.adjacent,
                                    dist.from.start=dist.from.start)
tss = TSS.adjacent$TSS
names(tss) = TSS.adjacent$gene
```

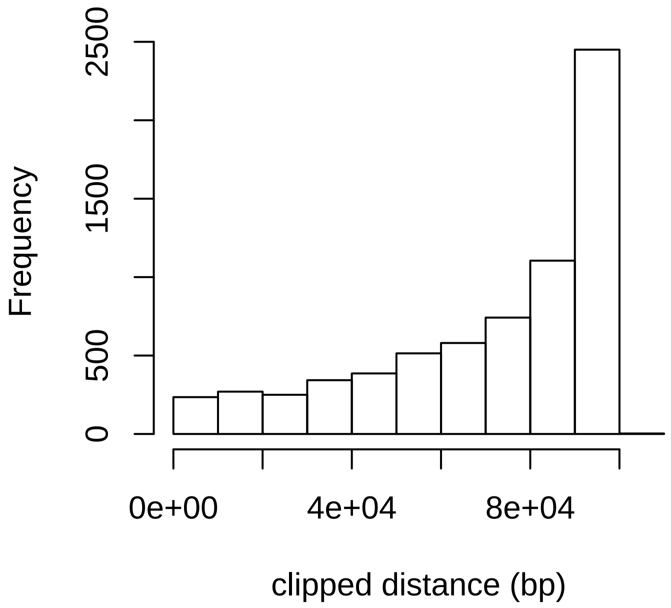
```r
adjacent.tts.up = apply.TSS.coords(bed=adjacent.tts.up, tss=tss)

# aggregate downstream adjacent genes with main data
adjacent.tts.down = bed.long[bed.long$gene %in% fix.genes$downstream,]
adjacent.tts.down = apply.TSS.coords(bed=adjacent.tts.down, tss=TSS.gene)
bed.long.filtered4.tss = rbind(bed.long.filtered3.tss, adjacent.tts.down,
                               adjacent.tts.up)
```

(d) Data-driven inference of transcription termination sites (TTSs)

The next task is to empirically identify TTSs, which are opperationally defined based on the following analysis. First we define regions at the end of genes to examine read counts. Note that transcription frequently extends beyond the poly-A site of a gene. So to capture the end of transcription, it is critical to examine regions beyond annotated gene boundries. For our analysis, we look for evidence of transcriptional termination in regions extending from a 3' subset of the gene to a selected number of base pairs downstream of the most distal annotated gene end. We initiated the search region with the last 10% of the largest gene annotation (fraction.end; see get.largest.interval()). We extended the search region up to 100 kb past the annotated gene end (add.to.end). We varied this parameter and discovered that for much lower values, gene ends were often identified at the boundries. Hence, we selected the large value of 100 kb in the interest of being as inclusive as possible. Further details related to this aspect of the analysis are described below. We also applied the constraint that a TTS could not be identified closer than 50 bp to the previously identified TSS of a downstream gene (dist.from.start). In general, this analysis incorporated the constraint that a gene end region identified could not cross the TSS of a downstream gene, thereby preventing gene overlaps on a given strand. Thus, we clipped the amount of bases added on to the gene end as necessary to avoid overlaps.

Here we demonstrate usage of the get.end.intervals() function. We evaluated whether there were any problems with the regions specified for TTS evaluation and we observed one case in which a minus strand gene had a boundry < 0, so we reset this bed start coordinate (i.e., gene end) to zero.

```r
#######################################################################################
## isolate gene ends for TTS estimation
#######################################################################################

# get intervals for TTS evaluation
add.to.end = 100000
fraction.end=0.1
dist.from.start=50
bed.for.tts.eval = get.end.intervals(bed=bed.long.filtered4.tss,
                                     add.to.end=add.to.end,
                                     fraction.end=fraction.end,
                                     dist.from.start=dist.from.start)

# check how many gene ranges were clipped to avoid overlaps
# set xy column to zero for unclipped genes
clipped = bed.for.tts.eval %>% filter(xy != "na")
hist(clipped$xy %>% as.numeric, main="", xlab = "clipped distance (bp)")
```



```
frac.clipped = nrow(clipped) / nrow(bed.for.tts.eval)
bed.for.tts.eval$xy[bed.for.tts.eval$xy == "na"] = 0
bed.for.tts.eval$xy = as.numeric(bed.for.tts.eval$xy)

# confirm the absence of overlaps
full.tts.eval = apply.TSS.coords(bed=bed.for.tts.eval, tss=TSS.gene)
overlap.data = gene.overlaps( bed = full.tts.eval )
has.start.inside = overlap.data$has.start.inside
is.a.start.inside = overlap.data$is.a.start.inside

# check for problems and modify accordingly
# ind.problem1 = which(full.tts.eval$end < full.tts.eval$start)
# ind.problem2 = which(full.tts.eval$end < 0)
ind.problem3 = which(full.tts.eval$start < 0)
length( ind.problem3 ) # 1 case, minus strand
#> [1] 1
bed.for.tts.eval$start[which(bed.for.tts.eval$gene==
                               full.tts.eval$gene[ind.problem3])] = 0
```

Given regions within which to search for TTSs, we opperationally defined the TTSs by binning the gene end regions, counting reads within the bins, fitting smooth spline curves to the bin counts, and de- tecting points at which the curves decay towards zero. For this analysis, we used a bin size of 50 bp (bp.bin) and we set the number of knots for the spline fits to the number of bins in the gene end search region divided by 40 (knot.div). Increasing this parameter results and a smoother curve. We applied the constraint that there must be at least 5 bins (250 bp) in the gene end interval (cnt.thresh), otherwise this TTS analysis was not applied (8 cases, 3 plus and 5 minus, revisited below). Similarly, if the number of knots identified was < 5 (i.e., <20 bins, <1000 bp; knot.thresh), then we set the number of knots to five (1,781 cases, 873 minus and 908 plus). For this analysis, we specified a sub-region at the beginning of the gene end region and identified the maximal peak from the spline fit. Then we identified the point at which the spline fit decayed to 2% of the peak level (pk.thresh). Reducing this parameter increases the sensitivity of the analysis for detecting the termination of transcription.

We selected a sub-region of the gene end because we set very long gene ends (i.e., up to 100 kb past the conventional annotation) and we did not want our identification of TTSs to be contaminated by distal enhancers or divergent transcripts. We reasoned that the sub-region should be largest for genes with the greatest numbers of clipped bases (around 100 kb), because such cases occur when the conventional gene ends are proximal to identified TSSs, and we should include these entire regions for analysis of the TTS. Similarly, we reasoned that for genes with substantially less clipped bases, and correspondingly larger gene end regions with greater potential for observing enhancers or divergent transcripts, the sub-regions should be smaller sections of the upstream-most gene end region. We used an exponential model to define the sub-regions.

We constrained the model to use 20% of the gene end regions with zero clipped bases (frac.min) and 100% of the end regions for genes with 100 kb of clipped bases (frac.max). The exponential decay was defined by a 10 kb distance constant (tau.dist). Therefore, for a gene with 100,000 - 10 = 90,000 kb of clipped bases, we set the peak search sub-region to 37% of the range between 0.2 and 1: 0.2 + exp(-1) * (1 - 0.2) = 0.49. That is, for a gene with 90,000 kb of clipped bases, we would search for the max spline peak within 49% of the region extending from 10% of the annotated gene end (fraction.end in get.end.intervals) and 10 kb past the annotated gene end. We set the search region to the maximal number of bases clipped, which was slightly above 100 kb due to the occaisionally close proximity of adjacent genes (add.to.end = 100,039). This was for the technical reason that we subtracted the number of clipped based from add.to.end to produce the argument to the exponential function defining the fraction of the gene end for evaluation, and this argument should not be less than zero (see get.TTS()).

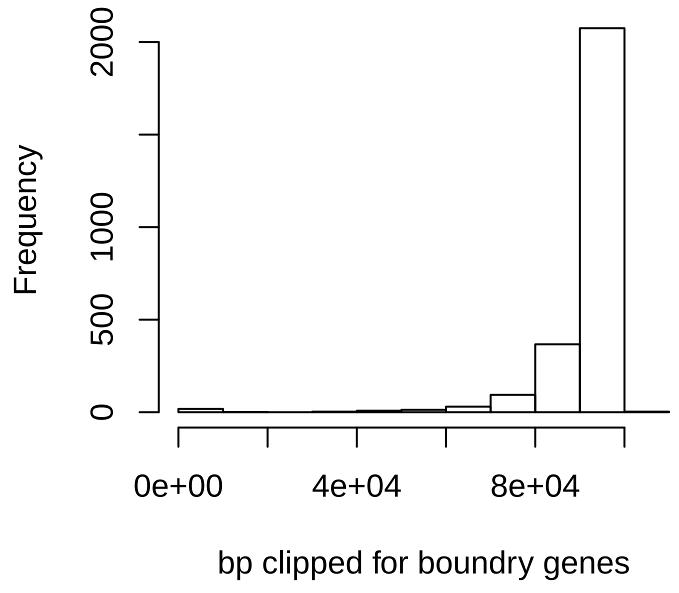
Now to identify the TTSs, use a subset of the data so it runs fast. The full data is included in the package (gene.ends).

```
#######################################################################################
## perform TTS estimation
#######################################################################################

# identify gene ends
add.to.end = max(bed.for.tts.eval$xy)
knot.div = 40
pk.thresh = 0.05
bp.bin = 50
knot.thresh = 5
cnt.thresh = 5
tau.dist = 10000
frac.max = 1
frac.min = 0.2
gene.ends.partial = get.gene.end(bed=bed.for.tts.eval[c(1:200,12000:12200),],
                         bw.plus=bw.plus, bw.minus=bw.minus,
                         bp.bin=bp.bin, add.to.end=add.to.end, knot.div=knot.div,
                         pk.thresh=pk.thresh, knot.thresh=knot.thresh,
                         cnt.thresh=cnt.thresh, tau.dist=tau.dist,
                         frac.max=frac.max, frac.min=frac.min)
```

Next we compute some basic metrics to evaluate the performance of the analysis. Our analysis revealed that 24% of the gene ends matched the gene end boundries. We assessed whether TTSs identified at the search boundry had clipped boundries. In the figure below, we show the distribution of clip distances for genes with TTSs identified at the boundries of the search regions from get.end.intervals(). These data indicate that the majority of the genes with boundry TTSs had large numbers of clipped bases, as should be the case. Fourty two genes had boundry TTSs with zero clipped basses. We manually analyzed this gene set using the UCSC genome browser. From this analysis, we removed 19 genes due to low read densities and apparent absence of expression, we reset TTSs for six genes to the comprehensive 'long gene' annotations, and we retained the identified TTSs for 17 genes. Here are the codes for plotting the clip distance distribution and adjusting the TTSs based on the manual analysis of 42 genes.

As mentioned above, there were eight cases in which TTSs could not be identified based on the constraints of our analysis. We set the TTSs of these gene to the long gene annotations.

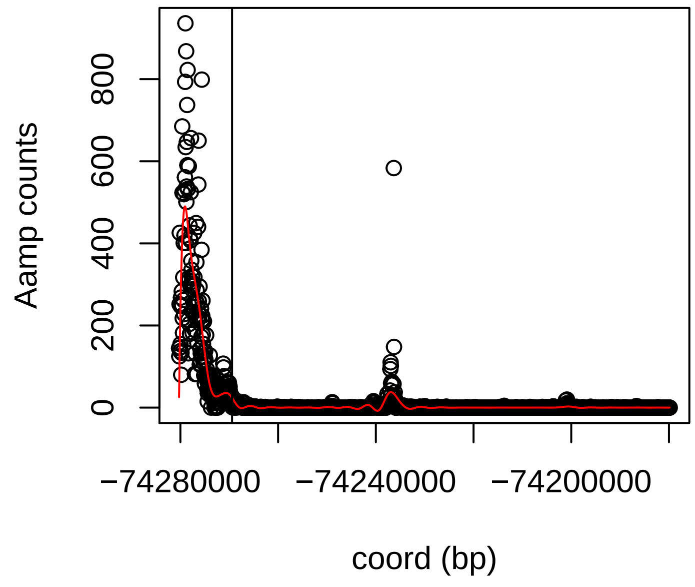
```r
# get metrics
minus.lowcount = length(gene.ends$minus.lowcount) # 5
plus.lowcount = length(gene.ends$plus.lowcount) # 3
minus.knotmod = length(gene.ends$minus.knotmod)
plus.knotmod = length(gene.ends$plus.knotmod)
ends = gene.ends$bed

# check for the percentage of identified TTSs that match the gene end boundry
gene.ends.plus = ends %>% filter(strand=="+") %>% select(end) %>%
  data.matrix %>% as.numeric
gene.ends.minus = ends %>% filter(strand=="-") %>% select(start) %>%
  data.matrix %>% as.numeric
tts.eval.plus = bed.for.tts.eval %>% filter(strand=="+") %>%
  select(end) %>% data.matrix %>% as.numeric
tts.eval.minus = bed.for.tts.eval %>% filter(strand=="-") %>%
  select(start) %>% data.matrix %>% as.numeric
ind.plus.match = which(gene.ends.plus==tts.eval.plus)
ind.minus.match = which(gene.ends.minus==tts.eval.minus)
pos = length(ind.plus.match)
neg = length(ind.minus.match)
frac.match = (pos+neg)/nrow(bed.for.tts.eval)

# look at whether TTSs identified at the search boundry had clipped boundries
clip.tts.plus = merge(ends,bed.for.tts.eval,by="gene") %>%
  filter(strand.x=="+",end.x==end.y)
clip.tts.minus = merge(ends,bed.for.tts.eval,by="gene") %>%
  filter(strand.x=="-",start.x==start.y)
clip.tts = rbind(clip.tts.plus, clip.tts.minus)
clip.tts0 = clip.tts %>% filter(xy.x==0)
n.boundry.genes = nrow(clip.tts0)
frac.bound.noclip = length(which(clip.tts$xy.x == 0)) / nrow(clip.tts)
frac.bound.clip = length(which(clip.tts$xy.x > 0)) / nrow(clip.tts)

# plot didtribution of clip distances for genes with boundary TTSs
hist(clip.tts$xy.x,main="",xlab="bp clipped for boundry genes")
```



```
# note that the preceding analysis sets unidentifiable TTSs to zero
# here we reset such cases to the 'long gene' ends
no.tts.minus = ends %>% filter(strand=="-", start==0) %>% select(gene)
no.tts.plus = ends %>% filter(strand=="+", end==0) %>% select(gene)
ind.end.minus = sapply(no.tts.minus$gene,function(x){which(ends$gene==x)})
ind.end.plus = sapply(no.tts.plus$gene,function(x){which(ends$gene==x)})
ind.lng.minus = sapply(no.tts.minus$gene,function(x){which(bed.long$gene==x)})
ind.lng.plus = sapply(no.tts.plus$gene,function(x){which(bed.long$gene==x)})
ends$start[ind.end.minus] = bed.long$start[ind.lng.minus]
ends$end[ind.end.plus] = bed.long$end[ind.lng.plus]

# add TSS data
bed.tss.tts = apply.TSS.coords(bed=ends, tss=TSS.gene)
```

The user may wish to visualize the performance to the TTS identification analysis for individual genes. Here is a way to accomplish this using gene.end.plot():

```
gene.end.plot(bed=bed.for.tts.eval, gene="Aamp",
              bw.plus=bw.plus, bw.minus=bw.minus,
              bp.bin=bp.bin, add.to.end=add.to.end, knot.div=knot.div,
              pk.thresh=pk.thresh, knot.thresh=knot.thresh,
              cnt.thresh=cnt.thresh, tau.dist=tau.dist,
              frac.max=frac.max, frac.min=frac.min)
```



(e) Data formating and output for upload to the UCSC browser

We now perform some final checks and processing steps. First we verify that none of our identified gene annotations span beyond the documented chromosome sizes. We also remove chrM from the output. Then we write the bed annotation to file.

```
#########################################################################################
## process results and get output for browser
#########################################################################################

# filter for chrom sizes
```

```
log.fix = c()
for(ii in 1:length(unique(bed.tss.tts$chr))){
  ind.chr = which(bed.tss.tts$chr == unique(bed.tss.tts$chr)[ii])
  ind.fix = which(bed.tss.tts$end[ind.chr] >
                    chrom.sizes$size[chrom.sizes$chr == unique(bed.tss.tts$chr)[ii]])
  if(length(ind.fix) > 0){
    bed.tss.tts$end[ind.chr][ind.fix] =
      chrom.sizes$size[chrom.sizes$chr == unique(bed.tss.tts$chr)[ii]]
    log.fix = c(log.fix, rep(unique(bed.tss.tts$chr)[ii],length(ind.fix)))
  }
}

# process output for browser visualization
out = bed.tss.tts %>% select(chr,start,end,gene)
out = cbind(out, c(500), bed.tss.tts$strand) %>% as.data.frame(stringsAsFactorf=F)

# remove mitochondrial coordinates
indM = which(out$chr == "chrM")
out = out[-indM,]

write.table(out,"tss_tts0.bed",quote=F,sep="\t",col.names=F,row.names=F)
```

Before we can upload our annotations to the browser, the following formatting commands are entered in the command line. Once complete, the file end_tts.bed can be uploaded to the UCSC genome browser.

```
# sort file
sort -k1,1 -k2,2n tss_tts0.bed > tss_tts_sorted.bed

# add file header
touch temp.txt
echo "track name=merge description="long" color=120,120,120" >> temp.txt
cat temp.txt tss_tts_sorted.bed > tsstts.bed

# remove excess
rm tss_tts0.bed tss_tts_sorted.bed temp.txt
```

6. Identification and annotation of de novo transcriptional units

The following analyses were completed to obtain annotated transcriptional unit (TU) coordinates. We used the R/bioconductor package groHMM link to implement de novo TU identification. This method has ben shown to outperform complementary methods in terms of sensitivity and specificity. Because groHMM supports parameter tuning for performance optimization, we varied key analysis parameters according to directions in the package documentation. We varied the log probability of a transition from the transcribed to the untranscribed state and the variance of the read counts in the untranscribed state (LtProbB and UTS, respectively). For this analysis, we used the gene annotations inferred previously (Section 5) to evaluate the performance of each set of TUs identified from a given HMM parameterization. The code for this analysis can also be found on github (ADD LINK).

(a) Annotating TUs based on identified gene coordinates

Given an optimized set of TUs generated by groHMM, the next step was the match these TUs to our previously identified gene annotations. We include hmm.bed and bed.tss.tts within the package in case the user wants to skip directly to this section. This analysis requires initial processing steps that require BEDtools (link). This first step of this analysis is to intersect the TU set with the gene annotations (using BEDTools) to identify gene/TU overlaps.

```
###########################################################################################
## intersect transcriptional units with gene annotations
###########################################################################################

# directory for applying bed intersect from inside R
bed.bin = "/media/wa3j/Seagate2/Documents/software/bedtools2/bin/"

# write bed files
write.table(hmm.bed,"hmm.bed",sep="\t",quote=F,col.names=F,row.names=F)
write.table(bed.tss.tts,"ann.bed",sep="\t",quote=F,col.names=F,row.names=F)

# sort the bed files
command1=paste('sort -k1,1 -k2,2n', 'hmm.bed', '> hmm.sorted.bed')
command2=paste('sort -k1,1 -k2,2n', 'ann.bed', '> ann.sorted.bed')
system(command1)
system(command2)

# create the command string and call the command using system()
comm = paste0(bed.bin,"intersectBed -s -wao -a hmm.sorted.bed -b ",
              "ann.sorted.bed > hmm_ann_intersect.bed")
system(comm)

# import intersect results
hmm.ann.overlap = read.table("hmm_ann_intersect.bed",header=F,stringsAsFactors=F)
names(hmm.ann.overlap) = c("hmm.chr", "hmm.start", "hmm.end",
                           "hmm.xy", "hmm.gene", "hmm.strand",
                           "ann.chr", "ann.start", "ann.end",
                           "ann.gene", "ann.xy", "ann.strand",
                           "overlap")
system("rm hmm_ann_intersect.bed hmm.sorted.bed ann.sorted.bed hmm.bed ann.bed")
```

Based on these overlaps, we identified basic charactistics of how the TUs relate to annotated genes. Our gene coordinate identification analysis generated a set of 12,247 genes, 12,242 (99.6%) of which were shown to overlap with identified TUs. In total, we identified 90,629 TUs. Out of all TUs, 86,425 (95.4%) did not overlap with annotated genes. Of the 4,204 TUs that overlapped with annotated genes, 2,142 unique TUs overlaped with multiple genes, with a total of 10,338 overlaps observed, and there were 2,062 TUs that overlapped with a single gene (2,142 + 2,062 = 4,204).

```
###########################################################################################
## identify basic characteristics of the hmm TUs
###########################################################################################

# genes overlapping with TUs
length(unique(hmm.ann.overlap$ann.gene))-1 # 12242
#> [1] 12242

# all TUs
nrow(unique(hmm.ann.overlap[,c(1:3,6)])) # 90629
#> [1] 90629

# identify TUs with no overlaps at all
```

```
isolated.tu = hmm.ann.overlap %>% filter(overlap == 0)
nrow(unique(isolated.tu[,c(1:3,6)])) # 86425
#> [1] 86425

# identify all TUs overlapping genes
overlap.tu = hmm.ann.overlap %>% filter(overlap != 0)
nrow(unique(overlap.tu[,c(1:3,6)])) # 4204
#> [1] 4204

# TUs with multi gene overlaps
dup.hmm.rows = overlap.tu[duplicated(overlap.tu[,c(1:3,6)]) |
                            duplicated(overlap.tu[,c(1:3,6)], fromLast=TRUE),]
nrow(dup.hmm.rows) # 10338
#> [1] 10338
nrow(unique(dup.hmm.rows[,c(1:3,6)])) # 2142
#> [1] 2142

# identify TUs overlapping single genes
sing.hmm.rows = overlap.tu[!duplicated(overlap.tu[,c(1:3,6)]) &
                             !duplicated(overlap.tu[,c(1:3,6)], fromLast=TRUE),]
nrow(unique(sing.hmm.rows[,c(1:3,6)])) # 2062
#> [1] 2062
nrow(sing.hmm.rows) # 2062
#> [1] 2062
```

(b) Single gene overlaps

The next task is to match the identified TUs with annotated genes. For this analysis, we separately considered TUs that overlap with single genes and TUs that overlap with multiple genes. For all single gene/TU overlaps, the annotation is implemented based on a reference case in which the entire gene is within the boundries of a TU. All other single overlap configurations are treated according to the logic for the reference scenario following a modification of the TU boundries. This analysis evaluates whether the beginning of a TU falls within tss.thresh bases of the gene annotation. If this criteria is not met, the initial portion of the TU, upstream of the gene annotation, is assigned an arbitrary identifier and the identified TSS defines the start of a TU with an identifier matching the respective gene name. Similarly, if the annotated gene end is more than delta.tts from the boundry of the TU, then we set the end of the TU named by the respective overlapping gene to the identified TTS and we identified the upstream region of the TU with an arbitrary label. We also set the constraint that maximal distance between an upstream gene end and the downstream gene start is defined by delta.tss. In particular, we defines all TUs in a manner that takes the union of identify TUs and gene annotations, except in cases in which annotated genes did not overlap any TUs (<1%). Here is the functions that implemented the single gene/TU overlap analysis.

We implemented the analysis as follows with tss.thresh = 200 bp, delta.tts = 1 kb, and delta.tss = 50 bp. We started with 2,062 unique TUs that overlapped with 1,977 unique genes. Hence, this analysis accomodated treatment of cases where a single gene overlapped with multiple TUs (dissociation error). As a consequence, this analysis generated new TUs that 'stitched together' the original TUs with intervening gene annotations. This analysis showed that there were 54 genes for which dissociation errors were observed. A total of 5,813 TUs were annotated from the single gene overlap analysis.

```
########################################################################################
## annotation for TUs with single gene overlaps
########################################################################################

# assign identifiers for single overlaps
```

```
nrow(sing.hmm.rows) # 2062
#> [1] 2062
nrow(unique(sing.hmm.rows[,1:3])) # 2062
#> [1] 2062
length(unique(sing.hmm.rows$ann.gene)) # 1977
#> [1] 1977
overlaps = sing.hmm.rows
names(overlaps)[1:6] = c("infr.chr","infr.start","infr.end",
                         "infr.gene","infr.xy","infr.strand")
tss.thresh = 200
delta.tss = 50
delta.tts = 1000
class1234 = single.overlaps(overlaps=overlaps, tss.thresh=tss.thresh,
                            delta.tss=delta.tss, delta.tts=delta.tts)
new.ann.sing = class1234$bed

# note. dissociation errors are reflected in duplicate entries
#       for single overlap genes
disso.tu = new.ann.sing[duplicated(new.ann.sing[,c(1,2,3,4,6)]) |
                          duplicated(new.ann.sing[,c(1,2,3,4,6)], fromLast=TRUE),]
disso.tu  = disso.tu[with(disso.tu, order(chr, start, end)),]
nrow(disso.tu) # 138
#> [1] 138
diss.genes = unique(c(disso.tu$id))
length(diss.genes) # 54
#> [1] 54
new.ann.sing = new.ann.sing[!duplicated(new.ann.sing[,
                            c('chr','start','end','id','strand')]),]
new.ann.sing = new.ann.sing[!duplicated(new.ann.sing[,c('id')]),]
nrow(new.ann.sing) # 5813
#> [1] 5813
```

(c) Multiple gene overlaps

Next we address cases with multiple genes overlapping individual TUs. As for single overlaps, we considered a reference case in which multiple annotated genes are observed within a single TU, and we analyze all other configurations according to the logic specified for the reference case. For the upstream-most and downstream-most overlaps, we apply analyses comparable to that for the initial scenario in which we use either the existing gene annotation or introduced an arbitrary identifier depending on the proximities of the boundries to the TSSs and TTSs. The TSS and TTS promity thresholds are defined by tss.thresh and delta.tts, respectively.

We implemented the analysis as described below (tss.thresh = 200, delta.tts = 1 kb, and delta.tss = 50 bp). For this analysis, we started with a total of 10,311 genes in 2,142 TUs. As with the treatment of cases in which single TUs overlapped with single genes, we observed cases where genes overlapped with multiple TUs (each of which overlapped with multiple genes). Such cases indicate dissociation error combined with merge errors. For our analysis, as above, this simply led to multiple identical entries for particular genes (27 genes, 54 total duplicated entries). This analysis generated 19,727 distinct TUs, including TUs that were identified based on the overlapping genes as well as those regions of the original TUs that did not overlap genes.

```
#######################################################################################
## annotation for TUs with multi gene overlaps
#######################################################################################
```

```
dup.full = dup.hmm.rows
names(dup.full)[1:6] = c("infr.chr","infr.start","infr.end",
                         "infr.gene","infr.xy","infr.strand")
nrow(dup.full[,1:3] %>% unique) # 2142 TUs
#> [1] 2142
nrow(dup.full %>% select(ann.gene) %>% unique) # 10311 genes
#> [1] 10311
tss.thresh = 200
delta.tss = 50
delta.tts = 1000
class5678 = multi.overlaps(overlaps=dup.full, tss.thresh=tss.thresh,
                           delta.tss=delta.tss, delta.tts=delta.tts)
new.ann.mult = class5678$bed

# identify overlaps
nrow(new.ann.mult) # 19754
#> [1] 19754
nrow(unique(new.ann.mult[,1:3])) # 19727
#> [1] 19727
disso.tu = new.ann.mult[duplicated(new.ann.mult[,1:4]) |
                          duplicated(new.ann.mult[,1:4], fromLast=TRUE),]
disso.tu  = disso.tu[with(disso.tu, order(chr, start, end)),]
nrow(disso.tu) # 54
#> [1] 54
length(unique(disso.tu$id)) # 27
#> [1] 27
new.ann.mult = new.ann.mult[!duplicated(new.ann.mult[,
                                                     c('chr','start','end','id','strand')]),]
new.ann.mult = new.ann.mult[!duplicated(new.ann.mult[,c('id')]),]
nrow(new.ann.mult) # 19727
#> [1] 19727
```

(d) Output annotations for visualization in the UCSC genome browser and downstream analysis

The final step in our identification of TUs for downstream analysis is to visualize the resulting TUs in the UCSC genome browser. First we output coordinate data from R. We output bed format data for single overlaps, multiple overlaps, HMM TUs without gene overlaps, and all original HMM TUs for comprehensive comparison. We note that our analysis examined overlaps of genes with TUs. A given TU could overlap with genes A, B, and C, while gene C could overlap with a distinct TU that overlaped with no other genes. This type of scenario leads to gene C being annotated twice, within the analyses for multiple and single TU overlaps. We remove such rare cases of multiple annotations here (n=46).

```
#######################################################################################
## output data for downstream analysis and loading into the browser
#######################################################################################

nrow(new.ann.mult) + nrow(new.ann.sing)
#> [1] 25540
length(unique(c(new.ann.sing$id, new.ann.mult$id)))
#> [1] 25494

# aggregate annotated and unannotated TUs
```

```r
out.overlap = rbind(new.ann.sing, new.ann.mult)
out.nooverlap = hmm.ann.overlap[which(hmm.ann.overlap$overlap == 0),1:6]
out.nooverlap[,4] = paste0("unann_",c(1:nrow(out.nooverlap)))
names(out.nooverlap) = names(out.overlap)
out.all = rbind(out.overlap, out.nooverlap)
dim(out.all) # 111965
#> [1] 111965      6
out.all = out.all[!duplicated(out.all$id),]
dim(out.all) # 111919
#> [1] 111919      6

# verify the absence of duplicates
which(duplicated(out.all)==TRUE)
#> integer(0)
length(unique(out.all$id)) == nrow(out.all)
#> [1] TRUE
out.all[duplicated(out.all$id)==TRUE,]
#> [1] chr    start  end    id     class  strand
#> <0 rows> (or 0-length row.names)

# output data for loading into the browser
out = new.ann.sing[,1:4]
out = cbind(out, c(500), new.ann.sing$strand) %>% as.data.frame(stringsAsFactors=F)
write.table(out,"split0.bed",quote=F,sep="\t",col.names=F,row.names=F)
out = new.ann.mult[,1:4]
out = cbind(out, c(500), new.ann.mult$strand) %>% as.data.frame(stringsAsFactors=F)
write.table(out,"merge0.bed",quote=F,sep="\t",col.names=F,row.names=F)

# output no-overlap hmm annotations for the browser
ind = which(hmm.ann.overlap$overlap == 0)
out = hmm.ann.overlap[ind,1:4]
out = cbind(out, c(500), hmm.ann.overlap$hmm.strand[ind]) %>% as.data.frame(stringsAsFactors=F)
write.table(out,"unann0.bed",quote=F,sep="\t",col.names=F,row.names=F)

# output original hmm annotations for the browser
out = hmm.bed[,1:4]
out = cbind(out, c(500), hmm.bed$strand) %>% as.data.frame(stringsAsFactors=F)
write.table(unique(out),"hmm0.bed",quote=F,sep="\t",col.names=F,row.names=F)

# output TU data
write.table(unique(out.all),"TU_20181107.bed",quote=F,sep="\t",col.names=F,row.names=F)
```

(e) Command line code for creating annotations to be visualized in the UCSC genome browser

We process the bed files as follows in the command line then we upload the files to the browser session with the read data and gene annotations.

```
############################################################################
# merge data
############################################################################

# sort file
sort -k1,1 -k2,2n merge0.bed > merge_sorted.bed
```

```
# add file header
touch temp.txt
echo "track name=merge description="merge" color=0,0,255" >> temp.txt
cat temp.txt merge_sorted.bed > merge.bed

# remove excess
rm merge_sorted.bed temp.txt

##########################################################################
# split data
##########################################################################

# sort file
sort -k1,1 -k2,2n split0.bed > split_sorted.bed

# add file header
touch temp.txt
echo "track name=split description="split" color=255,0,0" >> temp.txt
cat temp.txt split_sorted.bed > split.bed

# remove excess
rm split_sorted.bed temp.txt


##########################################################################
# orig hmm tus
##########################################################################

# sort file
sort -k1,1 -k2,2n hmm0.bed > hmm_sorted.bed

# add file header
touch temp.txt
echo "track name=hmm description="hmm" color=0,0,0" >> temp.txt
cat temp.txt hmm_sorted.bed > hmm.bed

# remove excess
rm hmm_sorted.bed temp.txt

##########################################################################
# orig hmm tus with no gene overlaps
##########################################################################

# sort file
sort -k1,1 -k2,2n unann0.bed > unann_sorted.bed

# add file header
touch temp.txt
echo "track name=unann description="unann" color=0,255,50" >> temp.txt
cat temp.txt unann_sorted.bed > unann.bed

# remove excess
rm unann_sorted.bed temp.txt
```

Here is a stable UCSC browser session with annotations for chr1 and chr6: link.