primaryTranscriptAnnotation

Warren Anderson, Mete Civelek, Michael Guertin

2019-08-05

This vignette describes basic usage of the version 1 primaryTranscriptAnnotation package. The package is currently available at https://github.com/WarrenDavidAnderson/genomicsRpackage/tree/master/primaryTranscriptAnnotation.

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 analyses of early adipogenesis.

For these analyses, we used bigWig format files derived from aggregating the aligned read data across 7 timepoints during the initial phase of 3T3-L1 cell differentiation into adipocytes. We do not describe how the bigWig files were obtained for our analyses, though links for our analysis scripts included below. For analyses such as those indicated below, we recommend aggregating the data from all of the conditions under investigation (e.g., all time points, control and disease, etc).

The user should be mindful to organize data objects as shown in this vignette.

Annotating nascent transcripts from run-on data is 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 balance between flexibility and simplicity. The general pipeline illustrated in section 8 contains several instances of recommendations for manual analysis and refinements of the annotations based in inspection of various performance metrics and quality checks. Such a process is recommended. However, a more streamlined approach can be taken as described in section 5. We have included several of our analysis data frames in the package so that the user can replicate our use of tha package in a streamlined manner.

2. Dependencies

The primaryTranscriptAnnotation R package has the following dependencies that should be loaded when running the functions described in this vignette. Here are web resources for package install:

dplyr: https://www.r-project.org/nosvn/pandoc/dplyr.html

pracma: https://cran.r-project.org/web/packages/pracma/index.html

bigWig: https://github.com/andrelmartins/bigWig

RColorBrewer: https://cran.r-project.org/web/packages/RColorBrewer/index.html

NMF: https://cran.r-project.org/web/packages/NMF/index.html

devtools: https://cran.r-project.org/web/packages/devtools/index.html

In particular, the bigWig package can be installed as follows:

```r
# install the package
library(devtools)
install_github("andrelmartins/bigWig", subdir="bigWig")
```

3. Installation and loading the package

The devtools library is required for installation directly from github (see section 2).

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

The following libraries should be loaded:

```
library(NMF)
library(dplyr)
library(bigWig)
library(pracma)
library(RColorBrewer)
library(primaryTranscriptAnnotation)
```

4. Acquiring and loading data

We parsed the data in gencode.firstExon and gencode.transcript from a comprehensive GENCODE annotation (https://www.gencodegenes.org/) using UNIX shell scripts. Here we first get the coordinates for every 'exon 1' and then we get the coordinates for documented transcripts.

```
# get gencode comprehensive annotations
# https://www.gencodegenes.org/mouse_releases/current.html
# https://www.gencodegenes.org/data_format.html
# https://www.gencodegenes.org/gencode_biotypes.html
wget ftp://ftp.ebi.ac.uk/pub/databases/gencode/Gencode_mouse/release_M22/gencode.vM22.annotation.gtf.gz
gunzip gencode.vM22.annotation.gtf.gz
```

```
# get the first exons for protein coding genes
grep 'transcript_type "protein_coding"' gencode.vM22.annotation.gtf | \
awk '{if($3=="exon"){print $0}}' | \
grep -w "exon_number 1" | \
cut -f1,4,5,7,9 | tr ";" "\t" | \
awk '{for(i=5;i<=NF;i++){if($i~/^gene_name/){a=$(i+1)}} print $1,$2,$3,a,"na",$4}' | \
tr " " "\t" | tr -d '"' > gencode.mm10.firstExon.bed

# get all transcripts for protein coding genes
grep 'transcript_type "protein_coding"' gencode.vM22.annotation.gtf | \
awk '{if($3=="transcript"){print $0}} ' | \
cut -f1,4,5,7,9 | tr ";" "\t" | \
awk '{for(i=5;i<=NF;i++){if($i~/^gene_name/){a=$(i+1)}} print $1,$2,$3,a,"na",$4}' | \
tr " " "\t" | tr -d '"' > gencode.mm10.transcript.bed
```

We downloaded that file corresponding to chrom.sizes (R object available with the package) directly from the UCSC browser website (https://genome.ucsc.edu/cgi-bin/hgGateway) using command line code. For reference, human data can be acquired similarly.

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

```
# get hg38 chromosome sizes
wget https://hgdownload-test.gi.ucsc.edu/goldenPath/hg38/bigZips/hg38.chrom.sizes
```

These files can be processed in R as follows.

```
# import data for first exons, annotate, and remove duplicate transcripts
fname = "gencode.mm10.firstExon.bed"
dat0 = read.table(fname,header=F,stringsAsFactors=F)
names(dat0) = c('chr', 'start', 'end', 'gene', 'xy', 'strand')
dat0 = unique(dat0)
gencode.firstExon = dat0

# import data for all transcripts, annotate, and remove duplicate transcripts
fname = "gencode.mm10.transcript.bed"
dat0 = read.table(fname,header=F,stringsAsFactors=F)
names(dat0) = c('chr', 'start', 'end', 'gene', 'xy', 'strand')
dat0 = unique(dat0)
gencode.transcript = dat0

# chromosome sizes
chrom.sizes = read.table("mm10.chrom.sizes",stringsAsFactors=F,header=F)
names(chrom.sizes) = c("chr","size")
```

Details regarding the processing of the PRO-seq read data can be found on github: https://github.com/WarrenDavidAnderson/genomeAnalysis/tree/master/PROseq. After plus and minus strand bigwigs have been generated, the data can be merged as follows using UCSC tools (http://hgdownload.soe.ucsc.edu/admin/exe/). This shell code is provided here for reference. However, the resulting merged bigWig files can be downloaded as described below.

```
# chrom sizes file
chromsizes=mm10.chrom.sizes

# merge plus
plusfiles=$(ls *plus*)
bigWigMerge ${plusfiles} preadip_plus_merged.bedGraph
bedGraphToBigWig preadip_plus_merged.bedGraph ${chromsizes} preadip_plus_merged.bigWig

# merge minus
minusfiles=$(ls *minus*)
bigWigMerge ${minusfiles} preadip_minus_merged.bedGraph
bedGraphToBigWig preadip_minus_merged.bedGraph ${chromsizes} preadip_minus_merged.bigWig
```

The package is based on our analyses of PRO-seq read data in the bigWig format. The data are stored online through CyVerse: https://user.cyverse.org. These data can be acquired as follows:

```
# download bigWig data
wget https://de.cyverse.org/dl/d/068D0BC4-89F1-4A55-AE27-694A2679AD6E/preadip_minus_merged.bigWig
wget https://de.cyverse.org/dl/d/026727C7-36EF-435E-9AA1-7A6F9E79C514/preadip_plus_merged.bigWig
```

The PRO-seq data used for this analysis consist of plus and minus bigWig files with aligned 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 = "preadip_plus_merged.bigWig"
minus.file = "preadip_minus_merged.bigWig"
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 for inferring primary transcript annotations. Further details that can be considered for primary transcript annotation are described below (see section 8). Start by loading key libraries and the PRO-seq data.

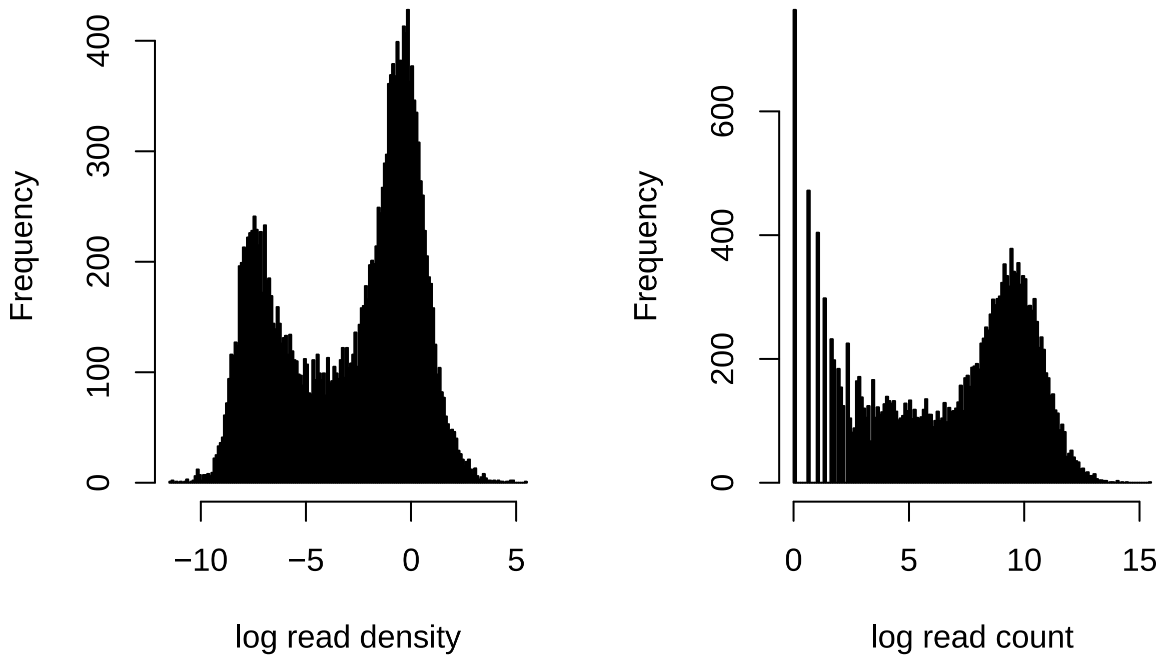
```
library(NMF)
library(dplyr)
library(bigWig)
library(pracma)
library(RColorBrewer)
library(primaryTranscriptAnnotation)

plus.file = "preadip_plus_merged.bigWig"
minus.file = "preadip_minus_merged.bigWig"
bw.plus = load.bigWig(plus.file)
bw.minus = load.bigWig(minus.file)
```

We obtain comprehensive gene annotations using get.largest.interval() and we count reads for individual transcripts using read.count.transcript(). We define the largest intervals by taking the minimal start coordinates and maximal end coordinates from the bed-format annotation for each gene. We use read.count.transcript() to determine the transcript with the highest read count for each gene, and the function returns that read count along with the associated read density.

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

# get read counts and densities for each annotated gene
transcript.reads = read.count.transcript(bed=gencode.transcript,
                                         bw.plus=bw.plus, bw.minus=bw.minus)
```

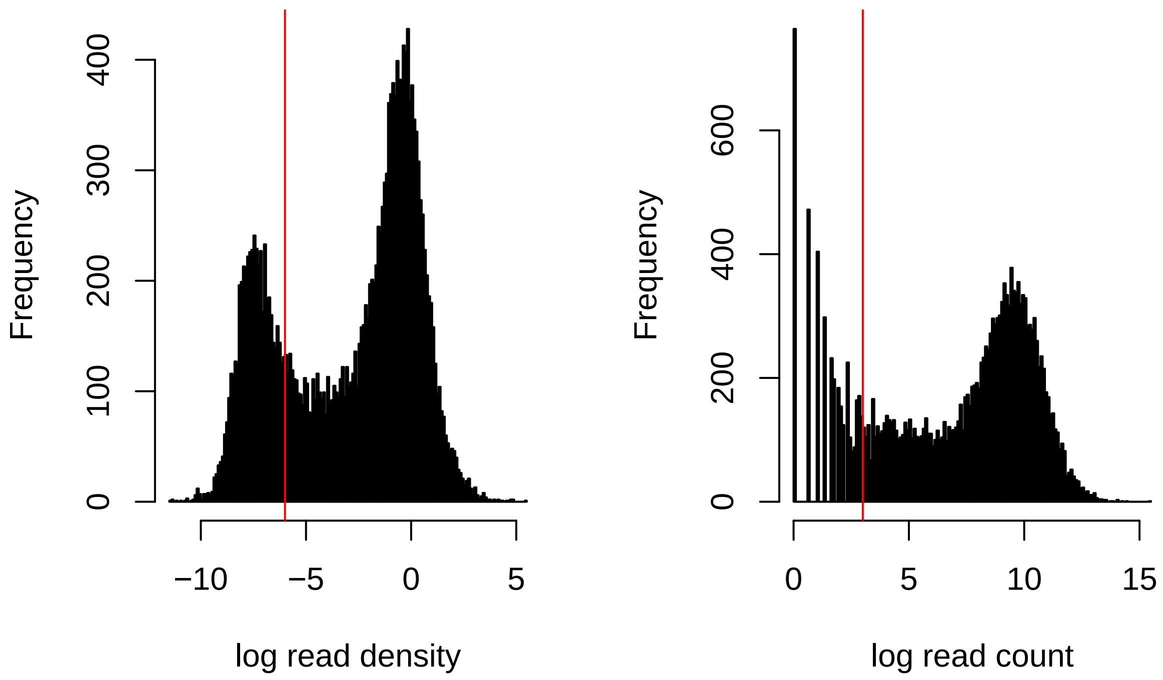
The data in transcript.reads can be used to filter out genes with considerably low or absent expression. One can then examine the count and density distributions and choose thresholds below which expression is considered negligible.

```
# evaluate count and density distributions
par(mfrow=c(1,2))
hist(log(transcript.reads$density), breaks=200,
     col="black",xlab="log read density",main="")
hist(log(transcript.reads$counts), breaks=200,
     col="black",xlab="log read count",main="")
```



Based on visual analysis of the distributions, the user can choose thresholds (e.g., log(counts)<3 or counts<20). Then the designated genes can be filtered out.

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



```r
# remove "unexpressed" genes
unexp = names(transcript.reads$counts)[ind.cut]
largest.interval.expr.bed = largest.interval.bed[!(largest.interval.bed$gene %in% unexp),]
```

Next a transcription start site (TSS) can be identified for each gene using get.TSS(). The bp.range determines the interval following the exon 1 start (or end for the minus strand) in which reads corresponding to pause sites can be identified. TSSs are taken as the closest exon1 starts relative to the identified pause sites.

```r
# select the TSS for each gene and incorporate these TSSs
# into the largest interval coordinates
bp.range = c(20,120)
cnt.thresh = 5
bed.out = largest.interval.expr.bed
bed.in = gencode.firstExon[gencode.firstExon$gene %in% bed.out$gene,]
TSS.gene = get.TSS(bed.in=bed.in, bed.out=bed.out,
                   bw.plus=bw.plus, bw.minus=bw.minus,
                   bp.range=bp.range, cnt.thresh=cnt.thresh)

TSS.gene = TSS.gene$bed
```

After identifying TSSs, it is important to address gene overlaps (i.e., multiple genes occupying the same coordinates) before performing transcription termination site (TTS) identification. The functions gene.overlaps() and remove.overlaps() can identify and remove overlaps, respectively. For this analysis, we recommend first removing poorly annotated genes (e.g., those with “Gm” or “Rik” identifiers in the mouse genome annotation). Overlaps are removed by retaining the gene with the most highly expressed transcript for each overlap case.

```r
# pre-filter to remover poorly annotated genes
TSS.gene = TSS.gene[-grep("Gm",TSS.gene$gene),]
TSS.gene = TSS.gene[-grep("Rik",TSS.gene$gene),]
TSS.gene = TSS.gene[-grep("AC",TSS.gene$gene),]
TSS.gene = TSS.gene[-grep("AW",TSS.gene$gene),]

# remove overlaps by selecting the gene with the highest density
overlap.data = gene.overlaps( bed = TSS.gene )
filtered.id.overlaps = remove.overlaps(bed=TSS.gene,
                                       overlaps=overlap.data$cases,
                                       transcripts=gencode.transcript,
                                       bw.plus=bw.plus, bw.minus=bw.minus, by="den")
```

Subsequent to filtering the data for overlaps, TTS coordinates can be identified by finding TTS search regions with get.end.intervals() and identifying the TTSs with get.TTS(). The get.TTS() function generates a bed formatted output with both inferred TSSs and inferred TTSs. More details on parameterizing these functions are described in section 8.

```r
# get intervals for TTS evaluation
add.to.end = 100000
fraction.end = 0.2
dist.from.start = 50
bed.for.tts.eval = get.end.intervals(bed=filtered.id.overlaps,
                                     add.to.end=add.to.end,
                                     fraction.end=fraction.end,
                                     dist.from.start=dist.from.start)

# 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 = 50000
frac.max = 1
frac.min = 0.3
inferred.coords = get.TTS(bed=bed.for.tts.eval, tss=filtered.id.overlaps,
                          bw.plus=bw.plus, bw.minus=bw.minus,
                          bp.bin=bp.bin, add.to.end=add.to.end,
                          pk.thresh=pk.thresh, knot.thresh=knot.thresh,
                          cnt.thresh=cnt.thresh, tau.dist=tau.dist,
                          frac.max=frac.max, frac.min=frac.min, knot.div=knot.div)

coords = inferred.coords$bed
```

6. Identification and annotation of de novo transcriptional units

The following analyses were completed to obtain annotated transcriptional unit (TU) coordinates. Either section 5 or section 8 should be completed prior to this section. However, if the user simply want to skip to this section, all necessary data are included along with the package (i.e., coords.filt, hmm.ann.overlap). We used the R/bioconductor package groHMM: https://bioconductor.org/packages/release/bioc/html/groHMM.html to implement de novo TU identification. This method has been 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 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: https://github.com/WarrenDavidAnderson/genomeAnalysis/tree/master/groHMMcode.

Given an optimized set of TUs generated by groHMM, the next step is to the match these TUs to previously identified gene annotations. We include inferred.coords, hmm.bed within the package in case the user wants to skip directly to this section. We also include hmm.ann.overlap in case the user wants to skip the following section. This analysis requires initial processing steps that require BEDtools https://bedtools.readthedocs.io/en/latest/. This first step of this analysis is to intersect the TU set with the gene annotations to identify gene/TU overlaps.

```r
# note that this frame should be an input to get.tu.gene.coords() below
bed.tss.tts = coords.filt$bed

# 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 characteristics of how the TUs relate to annotated genes using tu.gene.overlaps(). Our analysis identified 83,639 TUs (79,555 of these did not overlap with genes, 4,084 overlapped with genes). We found that 12,529 genes overlapped with TUs. Of the TUs that overlapped with genes, 1,978 TUs overlapped with individual genes and 2,106 overlapped with more than one gene.

```
tg.overlaps = tu.gene.overlaps(hmm.ann.overlap=hmm.ann.overlap)
tg.overlaps$n.tus # 83639
#> [1] 83639
tg.overlaps$n.gene.overlap.tu # 12529
#> [1] 12529
tg.overlaps$n.tu.no.gene.overlap # 79555
#> [1] 79555
tg.overlaps$n.tu.overlap.gene # 4084
#> [1] 4084
tg.overlaps$n.tu.overlap.single.genes # 1978
#> [1] 1978
tg.overlaps$n.tu.overlap.multiple.genes # 2106
#> [1] 2106
```

We use the get.tu.gene.coords() function to integrate TUs and inferred gene coordinates and establish a final annotation. Extensive details underlying this analysis can be found in the function documentation and associated manuscript. In brief, we distinguished cases of TUs overlapping single genes and TUs overlapping multiple genes.

For single gene/TU overlaps, the annotation is implemented based on a reference case in which the entire gene is within the boundaries of a TU. All other single overlap configurations are treated according to the logic for the reference scenario following a modification of the TU boundaries. 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 boundary 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. The analysis of single gene overlaps accommodated treatment of cases where a single gene overlapped with multiple TUs (dissociation error). As a consequence, this analysis can generate new TUs that 'stitch together' the original TUs with intervening gene annotations.

For 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. 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 boundaries to the TSSs and TTSs. The TSS and TTS proximity 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).

We use assess.final.coords() to evaluate the TU/gene matches. Our analysis revealed zero cases in which the final gene coordinate TSSs did not match the input TSSs (a consequence of tss.thresh = 0), and 107 cases in which the TTSs were not identical.

```
# get the final annotation
tss.thresh = 0
delta.tss = 50
delta.tts = 1000
res = get.tu.gene.coords(hmm.ann.overlap=hmm.ann.overlap,
                         gene.frame=bed.tss.tts,
                         tss.thresh=tss.thresh,
                         delta.tss=delta.tss,
                         delta.tts=delta.tts)

# evaluate the final annotation
metrics = assess.final.coords(bed1=bed.tss.tts, bed2=res)
metrics$dtss.uneq # 0
#> [1] 0
metrics$dtts.uneq # 107
#> [1] 107
```

As described in the Supplementary Materials of our manuscript, we apply arbitrary identifiers to TUs that do not directly overlap with genes. We output the data as follows.

```
# output single overlap data
out0 = res %>% filter(class==1 | class==2 | class==3 | class==4)
out = cbind(out0[,1:4], c(500), out0$strand) %>% as.data.frame(stringsAsFactors=F)
write.table(out,"split0.bed",quote=F,sep="\t",col.names=F,row.names=F)

# output multiple overlap data
out0 = res %>% filter(class==5 | class==6 | class==7 | class==8)
out = cbind(out0[,1:4], c(500), out0$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
out0 = res %>% filter(class=="0")
out = cbind(out0[,1:4], c(500), out0$strand) %>% 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 = cbind(hmm.bed[,1:4], 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(res,"TU_20190731.bed",quote=F,sep="\t",col.names=F,row.names=F)
```

7. Command line code for creating annotations to be visualized in the UCSC genome browser

Here we use the following UNIX code modify the outputs from above for adding custom tracks for visualization with the UCSC genome browser.

```
############################################################################
# inferred coords
############################################################################

# sort file
sort -k1,1 -k2,2n inferredcoords.bed > inferred_sorted.bed

# add file header
```

```
touch temp.txt
echo "track name=inferred description="inferred" color=120,120,120" >> temp.txt
cat temp.txt inferred_sorted.bed > inferred.bed

# remove excess
rm inferred_sorted.bed temp.txt

################################################################################
# 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
```

The hmm-based annotations can be visualized using a stable link: https://genome.ucsc.edu/s/warrena%40virginia.edu/primaryTranscriptAnnotation_20190801. Identified HMM-based TUs are shown in black, unannotated regions of identified TUs are shown in green. Genes and TU segments from single TU/gene overlaps are shown in red, Genes and TU segments from multiple overlaps are shown in blue, and inferred coordinates are shown in gray.

8. Data-driven gene annotation

(a) Comprehensive annotation and removal of unexpressed genes

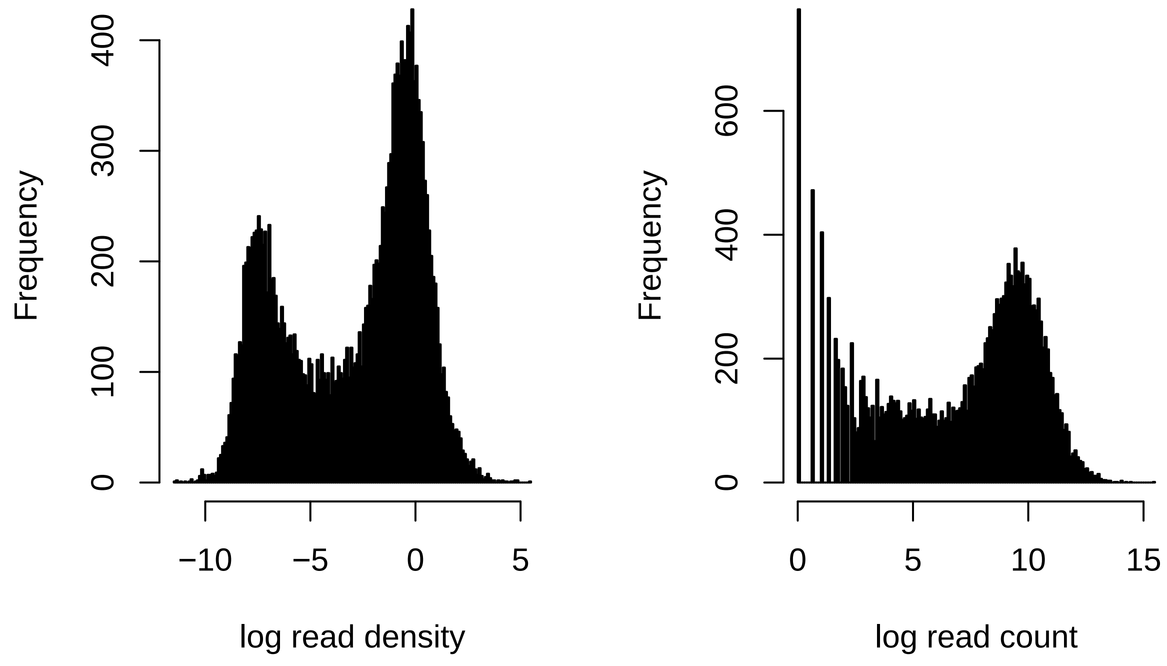
This section is a more elaborate version of the transcript coordinate identification process described i section 5. BigWig data and conventional gene annotations should be acquired before running this section. Here we describe the initial procedures for defining gene boundaries with the use of both existing annotations and PRO-seq data. First we perform some simple processing steps. We obtain comprehensive gene annotations using get.largest.interval() and we identify transcripts with maximal reads for each gene using read.count.transcript(). We obtain the largest intervals by taking the minimal start coordinates and maximal end coordinates from the bed-format annotation. We use read.count.transcript() to determine the transcript with the highest read count for each gene, and the function returns that read count along with the associated read density.

```
library(NMF)
library(dplyr)
library(bigWig)
library(pracma)
library(RColorBrewer)
library(primaryTranscriptAnnotation)

# load bigWigs
plus.file = "preadip_plus_merged.bigWig"
minus.file = "preadip_minus_merged.bigWig"
bw.plus = load.bigWig(plus.file)
bw.minus = load.bigWig(minus.file)

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

```
# get read counts and densities in annotated gene
transcript.reads = read.count.transcript(bed=gencode.transcript,
                                         bw.plus=bw.plus, bw.minus=bw.minus)
```

The data in transcript.reads can be used to filter out genes with considerably low or absent expression. One can then examine the count and density distributions and choose thresholds below which expression is considered negligible.

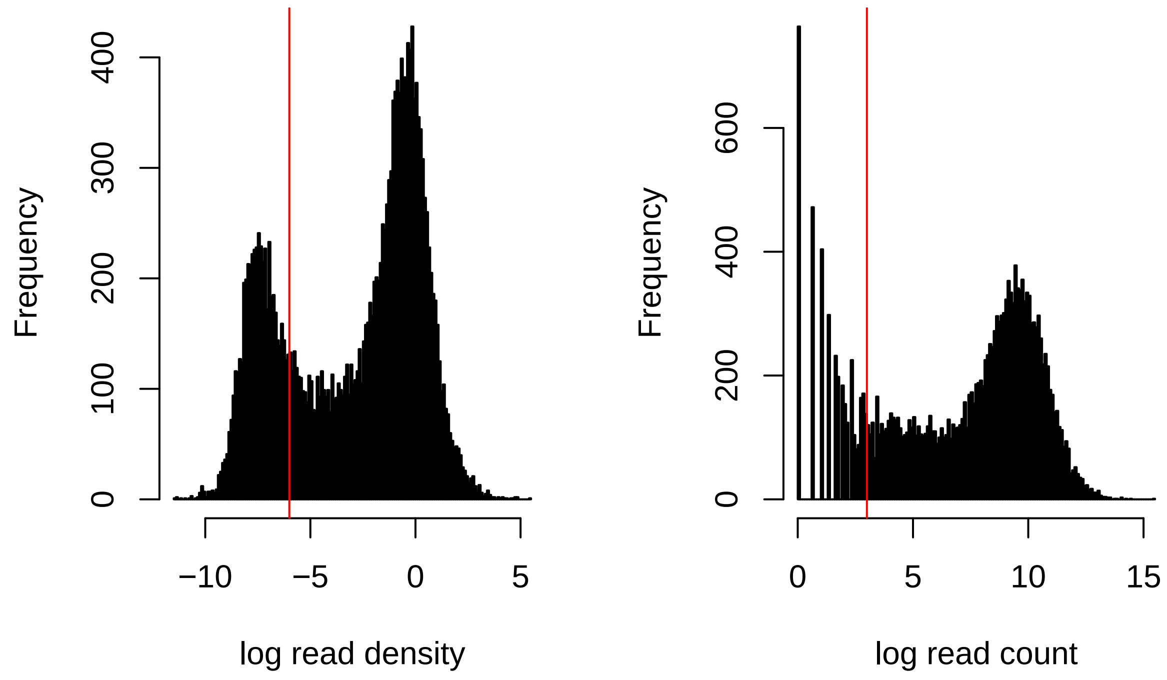
```
# evaluate count and density distributions
par(mfrow=c(1,2))
hist(log(transcript.reads$density), breaks=200,
     col="black",xlab="log read density",main="")
hist(log(transcript.reads$counts), breaks=200,
     col="black",xlab="log read count",main="")
```



Based on visual analysis of the distributions, the user can choose thresholds (e.g., log(counts)<3 or counts<20). Then the designated genes can be filtered out.

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

```
par(mfrow=c(1,2))
hist(log(transcript.reads$density), breaks=200,
     col="black",xlab="log read density",main="")
abline(v=den.cut, col="red")
hist(log(transcript.reads$counts), breaks=200,
     col="black",xlab="log read count",main="")
abline(v=cnt.cut, col="red")
```



```
# remove "unexpressed" genes
unexp = names(transcript.reads$counts)[ind.cut]
largest.interval.expr.bed = largest.interval.bed[
  !(largest.interval.bed$gene %in% unexp),]
```

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

A transcription start site (TSS) can be identified for each gene using get.TSS(). The bp.range parameter determines the interval following the exon 1 start (or end for the minus strand) in which reads corresponding to pause sites can be identified. The default behavior of this function entails removing genes with all exon 1 counts < cnt.thresh (low.read=FALSE), because in such a case, the data needed for TSS inference are absent. To keep the upstream-most exon 1 as the TSS for such cases, indicate low.read=TRUE.

We implement this analysis by setting a range adjacent to 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 (hence, bp.range = c(20,120)). We determine the first exon with the maximal density out of all first exons for a given gene, and we define the TSS as the start of the exon 1 with highest density.

```r
# select the TSS for each gene and incorporate these TSSs
# into the largest interval coordinates
# note that the default bevahior of this function entails removing genes
# with all exon 1 counts < cnt.thresh (low.read=FALSE)
# to keep the upstream-most exon 1 as the TSS for such cases, indicate low.read=TRUE
bp.range = c(20,120)
cnt.thresh = 5
bed.out = largest.interval.expr.bed
bed.in = gencode.firstExon[gencode.firstExon$gene %in% bed.out$gene,]
TSS.gene = get.TSS(bed.in=bed.in, bed.out=bed.out,
                   bw.plus=bw.plus, bw.minus=bw.minus,
                   bp.range=bp.range, cnt.thresh=cnt.thresh)

TSS.gene = TSS.gene$bed
```

Here the user can evaluate the performance of the TSS inference process. We specify a region of 1 kb, centered on the TSS, for this analysis (i.e., 500 bp on either side, window = 1000). We obtain read counts in 10 bp bins that span this window (bp.bin = 10). The eval.tss() function sorts the genes based on the bin with the maximal reads and scales the read data to the interval (0,1) for visualization. We compute the distances between the TSS and the bin with the max reads within the specified window. The eval.tss() function can be used to get reads around TSSs and compute distances from read peaks to TSSs. We performed this analysis for both the identified coordinates and the largest interval gene coordinates (min start, max end). We plotted heatmaps sorted based on the max bin counts 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 extended for the largest interval gene annotations (data not shown).

```r
# parameters and analysis
window = 1000
bp.bin = 10
bed1 = TSS.gene
bed2 = largest.interval.expr.bed
bed2 = bed2[bed2$gene %in% bed1$gene,]
tss.eval = eval.tss(bed1=bed1, bed2=bed2,
                    bw.plus=bw.plus, bw.minus=bw.minus,
                    window=window, bp.bin=bp.bin, fname="TSSres.pdf")

# plot examples
bk = seq(-window/2, window/2, bp.bin)
hist(tss.eval$tss.dists.inf$dist,main="inferred",
     xlab="dist from TSS to read max (bp)",
     breaks=bk,col="black")
plot(rowSums(tss.eval$tss.dists.inf$raw), tss.eval$tss.dists.inf$dist,
     xlab="window read depth",
     ylab="dist from TSS to read max (bp)")

# alternatively, the data can be plotted based on the organization of
# the TSS distances ranked for the largest interval genes
id.inds = sapply(names(tss.eval$tss.dists.lng$dist),function(x){
  which(names(tss.eval$tss.dists.inf$dist)==x)
}) %>% unlist
pdf("TSS_heatmap_li.pdf", onefile=FALSE)
par(mfrow=c(2,2))
```

```
aheatmap(tss.eval$tss.dists.inf$scaled[id.inds,],Colv=NA,Rowv=NA,
         color=brewer.pal(9,"Greys"),main="inferred")
aheatmap(tss.eval$tss.dists.lng$scaled,Colv=NA,Rowv=NA,
         color=brewer.pal(9,"Greys"),main="largest")
dev.off()
```

(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 using get.dups(). The duplicate cases can be written out for manual analysis. The instances of overlapping genes can then be evaluated by outputting the duplicate data and visualizing the read data using the UCSC genome browser https://genome.ucsc.edu/.

```
# identify duplicates
dups = get.dups(bed = TSS.gene)
max(dups$cases) # 17 cases
#> [1] 17
head(dups)
#>       chr    start      end    gene xy strand cases
#> 11   chr1 88134882 88218997 Ugt1a6a na      +     1
#> 12   chr1 88166012 88218997  Ugt1a5 na      +     1
#> 13   chr1 88200601 88218997  Ugt1a2 na      +     1
#> 14   chr1 88211959 88218997  Ugt1a1 na      +     1
#> 15 chr10 80602880 80615783  Scamp4 na      +     2
#> 16 chr10 80606021 80615783 Gm49322 na      +     2

# output duplicate data for manual curration
# note that some genes are found to be adjacent, these will be addressed below
fname = "duplicateCoords_20190724.txt"
write.table(dups,fname,col.names=T,row.names=F,sep="\t",quote=F)
```

Our analysis of duplicates revealed 17 cases in which multiple genes overlap. Based on our manual analysis, we either removed genes or marked cases in which pairs of genes appeared to be adjacent. The later were set aside for subsequent analysis. For cases in which multiple genes had overlapping annotations, we removed the least well-documented 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. We include the results of our manual analysis within this package (remove.genes.id, fix.genes.id). These results will be used below.

```
# output duplicate data for manual curration
# note that some genes are found to be adjacent, these will be addressed below
fname = "duplicateCoords_20190724.txt"
write.table(dups,fname,col.names=T,row.names=F,sep="\t",quote=F)

# input results, includen in the package
# remove.genes.id = read.table("duplicateRemove_remove_20190724.csv",
#                              header=T,sep=",",stringsAsFactors=F)
# fix.genes.id = read.table("duplicateRemove_keepAdjacent_20190724.csv",
#                              header=T,sep=",",stringsAsFactors=F)
head(remove.genes.id)
#>    remove
#> 1  Ugt1a5
#> 2  Ugt1a2
#> 3  Ugt1a1
#> 4 Gm49322
#> 5  Trim17
#> 6 Gm43951
head(fix.genes.id)
#>   upstream downstream
#> 1     Aars     Exosc6
#> 2    Glod4     Gemin4
```

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 these genes in fix.genes.id.

```
# remove genes based on manual analysis
# note that genes from fix.genes.id will be addressed below
genes.remove1 = c(remove.genes.id$remove, fix.genes.id$upstream, fix.genes.id$downstream)
TSS.gene.filtered1 = TSS.gene[!(TSS.gene$gene %in% genes.remove1),]

# verify the absence of identical coordinate overlaps (output NULL)
get.dups(bed = TSS.gene.filtered1)
#> NULL
```

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 boundaries 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.

```
# run overlap analysis
overlap.data = gene.overlaps( bed = TSS.gene.filtered1 )
has.start.inside = overlap.data$has.start.inside
is.a.start.inside = overlap.data$is.a.start.inside
head(has.start.inside)
#>       chr     start       end    gene
#> 1963 chr6 128362967 128376146   Foxm1
#> 415  chr7 101394368 101412586   Arap1
#> 830  chr7   5020553   5060785 Ccdc106
#> 2183 chr7   5015496   5033097 Gm44973
#> 2187 chr7 105640577 105655748 Gm45799
#> 2188 chr7 101410886 101512819 Gm45837
#>                                           xy strand
#> 1963          Tex52 starts inside this gene      +
#> 415         Gm45837 starts inside this gene      +
#> 830   Zfp580,Zfp865 starts inside this gene      +
#> 2183 Ccdc106,Zfp865 starts inside this gene      +
#> 2187          Dnhd1 starts inside this gene      +
#> 2188          Pde2a starts inside this gene      +
head(is.a.start.inside)
```

```
#>        chr     start       end    gene                           xy
#> 5462  chr6 128375456 128385144   Tex52   this gene starts inside Foxm1
#> 2188  chr7 101410886 101512819 Gm45837   this gene starts inside Arap1
#> 6289  chr7   5051538   5053723  Zfp580 this gene starts inside Ccdc106
#> 6344  chr7   5020570   5036768  Zfp865 this gene starts inside Ccdc106
#> 830   chr7   5020553   5060785 Ccdc106 this gene starts inside Gm44973
#> 63441 chr7   5020570   5036768  Zfp865 this gene starts inside Gm44973
#>       strand
#> 5462       +
#> 2188       +
#> 6289       +
#> 6344       +
#> 830        +
#> 63441      +
head(overlap.data$cases)
#>       chr     start       end    gene xy strand cases
#> 1963 chr6 128362967 128376146   Foxm1 na      +     1
#> 5462 chr6 128375456 128385144   Tex52 na      +     1
#> 415  chr7 101394368 101412586   Arap1 na      +     2
#> 2188 chr7 101410886 101512819 Gm45837 na      +     2
#> 3916 chr7 101421691 101512827   Pde2a na      +     2
#> 830  chr7   5020553   5060785 Ccdc106 na      +     3
```

There were typically poorly annotated mouse genes involved in overlaps that we eliminated, as shown below, and then re-evaluated the remaining data for overlaps.

```
# document numbers and filter overlap data to remove specific gene annotations
overlap.genes = overlap.data$cases$gene %>% unique
length(grep("AC",overlap.genes)) # 3
#> [1] 3
length(grep("AL",overlap.genes)) # 3
#> [1] 3
length(grep("Gm",overlap.genes)) # 86
#> [1] 86
length(grep("Rik",overlap.genes)) # 11
#> [1] 11
genes.remove2 = overlap.genes[grep("AC",overlap.genes)]
genes.remove2 = c(genes.remove2, overlap.genes[grep("AL",overlap.genes)])
genes.remove2 = c(genes.remove2, overlap.genes[grep("Gm",overlap.genes)])
genes.remove2 = c(genes.remove2, overlap.genes[grep("Rik",overlap.genes)])
```

We eliminated any 'large' genes that enclosed the initiation of two or more genes using inside.starts().

```
# identify genes with multiple starts inside (i.e., 'big' genes)
# set these genes for removal
mult.inside.starts = inside.starts(vec = is.a.start.inside$xy)
length(mult.inside.starts) # 29
#> [1] 29
genes.remove2 = c(genes.remove2, mult.inside.starts) %>% unique
```

After removing overlap genes as just described, we manually analyzed the remaining 85 overlap cases.

```
# remove filtered genes and re-run overlap analysis
in.dat = TSS.gene.filtered1[!(TSS.gene.filtered1$gene %in% genes.remove2),]
overlap.data = gene.overlaps( bed = in.dat )
has.start.inside = overlap.data$has.start.inside
is.a.start.inside = overlap.data$is.a.start.inside
case.dat = overlap.data$cases
length(unique(case.dat$cases)) # 85
#> [1] 85

# output data for manual curration
fname = "nonid_overlaps_20190724.txt"
write.table(case.dat,fname,col.names=T,row.names=F,sep="\t",quote=F)
```

We carried out our manual analysis by visualizing 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 overlapping genes, we removed genes for the following reasons (n cases): the smaller gene did not show distinguishable expression (n=36), no mitigating information was available and the annotations were nearly identical (arbitrary decision to remove one of two genes, n=10), the gene that was either less common or had no documented role in adipose biology (n=4), the gene was relatively large and did not appear to be expressed (n=2), or both genes in the pair were removed because there was an apparent absence of expression (n=8). In addition to the genes removed, we flagged 25 gene pairs in which both adjacent genes appeared to be expressed. These 25 gene pairs are examined further below. We include the results of our manual analysis in the package (see remove.genes.ov and fix.genes.ov). We initially remove all genes that were flagged through our manual analysis and verify the absence of overlaps in the filtered data.

```
# remove genes based on manual curation
# note that only one overlap can be retained per case, otherwise,
# the coordinates for two adjacent genes can be addressed
# note that the 'fix.gene' or fix genes are addressed below
# fname = "nonid_overlaps_remove_20190726.csv"
# remove.genes.ov = read.table(fname,header=T,sep=",",stringsAsFactors=F)
# fname = "nonid_overlaps_revise_20190726.csv"
# fix.genes.ov = read.table(fname,header=T,sep=",",stringsAsFactors=F)

head(remove.genes.ov)
#>    remove                           decision
#> 1   Nup62 arbitrary for matching annotations
#> 2    Prnp arbitrary for matching annotations
#> 3   Cers1 arbitrary for matching annotations
#> 4 Pcdhgb8 arbitrary for matching annotations
#> 5   Haus3 arbitrary for matching annotations
#> 6  Adam1a arbitrary for matching annotations
head(fix.genes.ov)
#>   upstream downstream
#> 1    Foxm1     Tex52
#> 2   Xntrpc     Trpc2
#> 3    Bbof1   Rnf113a2
#> 4    Hif1a    Snapc1
#> 5    Pfdn5       Myg1
#> 6    Psma2   AW209491

genes.remove2 = c(genes.remove2, remove.genes.ov$remove,
                  fix.genes.ov$upstream, fix.genes.ov$downstream)
```

```
TSS.gene.filtered2 = TSS.gene.filtered1[
  !(TSS.gene.filtered1$gene %in% genes.remove2),]

# verify the absence of overlaps
overlap.data = gene.overlaps( bed = TSS.gene.filtered2 )
overlap.data$cases # NULL
#> [1] chr    start  end    gene   xy     strand cases
#> <0 rows> (or 0-length row.names)
```

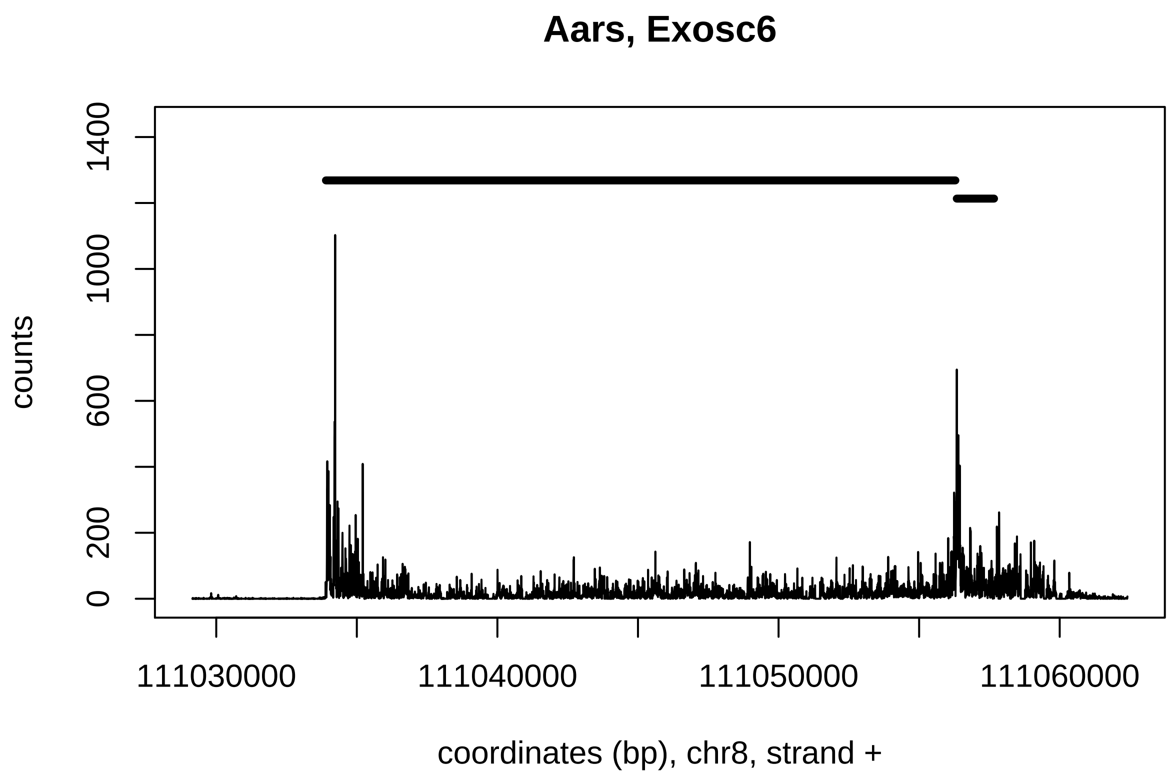
For gene annotation overlaps that are identified as adjacent genes, the respective TSSs and TTS can be identified using adjacent.gene.coords(). We considered the adjacent gene pairs with overlaps that were identified based on manual curation. We empirically define TSSs for these genes by binning the 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 bin size and apply the constraint that the identified 'pause' peaks must be a given distance apart. The search region includes a specified distance upstream of the upstream-most gene. For the spline fits, we set the number of knots to the number of bins divided by specified setting. We identify the TSSs as the exon 1 coordinates closest to the pasue site peaks. We identify the TTS of the upstream gene as an interval from the TSS of the downstream gene. The input TTS for the downstream gene is retained.

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 aforementioned parameters. Plots of read counts, spline fits, and TSS locations are printed to an output file for visual inspection.

```
# get the coordinates for adjacent gene pairs
fix.genes = rbind(fix.genes.id, fix.genes.ov)
bp.bin = 5
knot.div = 40
shift.up = 100
delta.tss = 50
diff.tss = 1000
dist.from.start = 50
adjacent.coords = adjacent.gene.coords(fix.genes=fix.genes, bed.long=TSS.gene,
                                 exon1=gencode.firstExon,
                                 bw.plus=bw.plus, bw.minus=bw.minus,
                                 knot.div=knot.div, bp.bin=bp.bin,
                                 shift.up=shift.up, delta.tss=delta.tss,
                                 dist.from.start=dist.from.start,
                                 diff.tss=diff.tss, fname="adjacentSplines.pdf")
```

It is possible to visualize the inferred coordinates for the adjacent gene pairs using adjacent.coords.plot().

```
# visualize coordinates for a specific pair
adjacent.coords.plot(adjacent.coords=adjacent.coords,
                     pair=fix.genes[1,],
                     bw.plus=bw.plus,
                     bw.minus=bw.minus)
```



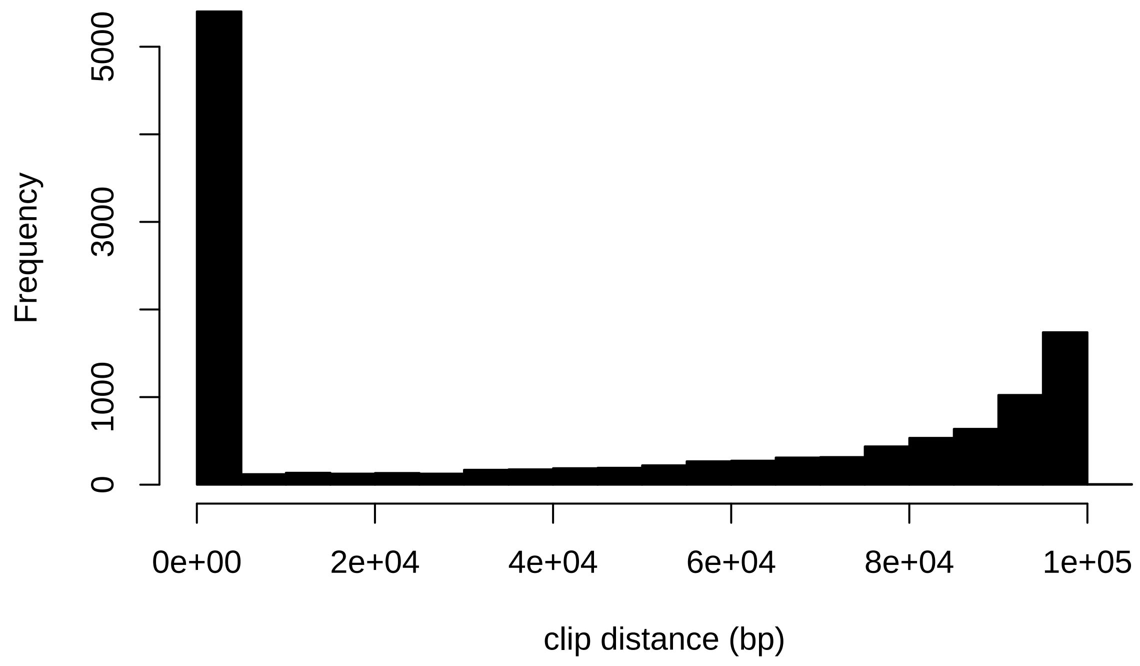
To complete the overlap analysis, we apply the coordinates of the adjacent gene pairs, which were previously removed, back to the data set.

```r
# aggregate downstream adjacent genes with main data
TSS.gene.filtered3 = rbind(TSS.gene.filtered2, adjacent.coords)

# verify the absence of overlaps
overlap.data = gene.overlaps( bed = TSS.gene.filtered3 )
overlap.data$cases # NULL
#> [1] chr    start  end    gene   xy     strand cases
#> <0 rows> (or 0-length row.names)
```

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

The next task is to empirically identify TTSs, which are operationally 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. To capture the end of transcription, it is critical to examine regions beyond annotated gene boundaries. 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 20% of the largest gene annotation (fraction.end). 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 boundaries. 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. First we use get.end.intervals() to define TTS search regions and plot the distribution of clip distances.

```
# get intervals for TTS evaluation
add.to.end = 100000
fraction.end = 0.2
dist.from.start = 50
bed.for.tts.eval = get.end.intervals(bed=TSS.gene.filtered3,
                                     add.to.end=add.to.end,
                                     fraction.end=fraction.end,
                                     dist.from.start=dist.from.start)

# distribution of clip distances
hist(bed.for.tts.eval$xy,xlab="clip distance (bp)",col="black",main="")
```



The distribution shows a peak at zero and a peak at 100 kb. The zero peak corresponds to genes for which a downstream expressed gene is not observed within 100 kb. The 100 kb peak corresponds to genes for which the most downstream annotated gene ends are in close proximity to downstream genes.

Given regions within which to search for TTSs, we operationally defined the TTSs by binning the gene end regions, counting reads within the bins, fitting smooth spline curves to the bin counts, and detecting points at which the curves peak and then 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 in 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. 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. 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 5% 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 for detecting curve peaks 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 peak search 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 peak. 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 peak search 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 30% of the TTS search 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 50 kb distance constant (tau.dist). We set the search region to the maximal number of bases clipped, which was slightly above 100 kb due to the occasionally close proximity of adjacent genes. 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()).

TTSs can be identified as follows. This analysis may take around a half hour (32 Gb RAM, 2.80GHz CPU).

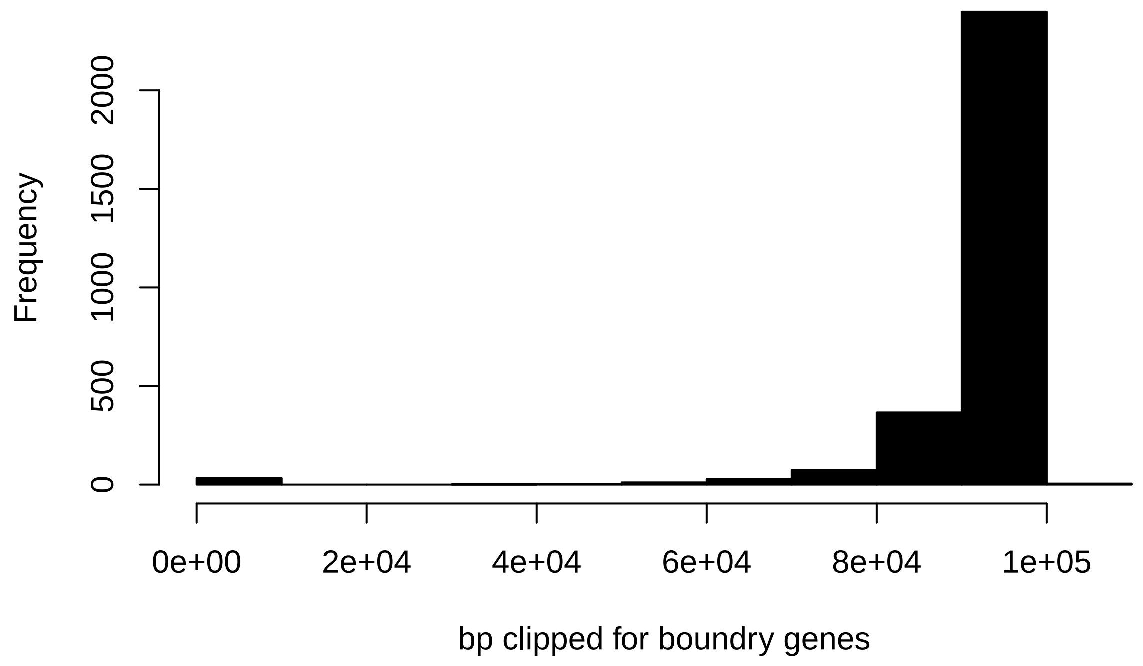
```
# identify gene ends
add.to.end = max(bed.for.tts.eval$xy)
knot.div = 40
pk.thresh = 0.05
bp.bin = 50
knot.thresh = 5
cnt.thresh = 5
tau.dist = 50000
frac.max = 1
frac.min = 0.3
inferred.coords = get.TTS(bed=bed.for.tts.eval, tss=TSS.gene.filtered3,
                          bw.plus=bw.plus, bw.minus=bw.minus,
                          bp.bin=bp.bin, add.to.end=add.to.end,
                          pk.thresh=pk.thresh, knot.thresh=knot.thresh,
                          cnt.thresh=cnt.thresh, tau.dist=tau.dist,
                          frac.max=frac.max, frac.min=frac.min, knot.div=knot.div)
```

For convenience, we include inferred.coords in the package data.

```
coords = inferred.coords$bed
```

Next we compute some basic metrics to evaluate the performance of the analysis. Our analysis revealed that 23% of the gene ends matched the gene end boundaries (TTS.boundary.match()). We assessed whether TTSs identified at the search boundary had clipped boundaries (TTS.boundary.clip()). We found that 98% of the TTSs at search boundaries were clipped. In the figure below, we show the distribution of clip distances for genes with TTSs identified at the boundaries of the TTS 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.

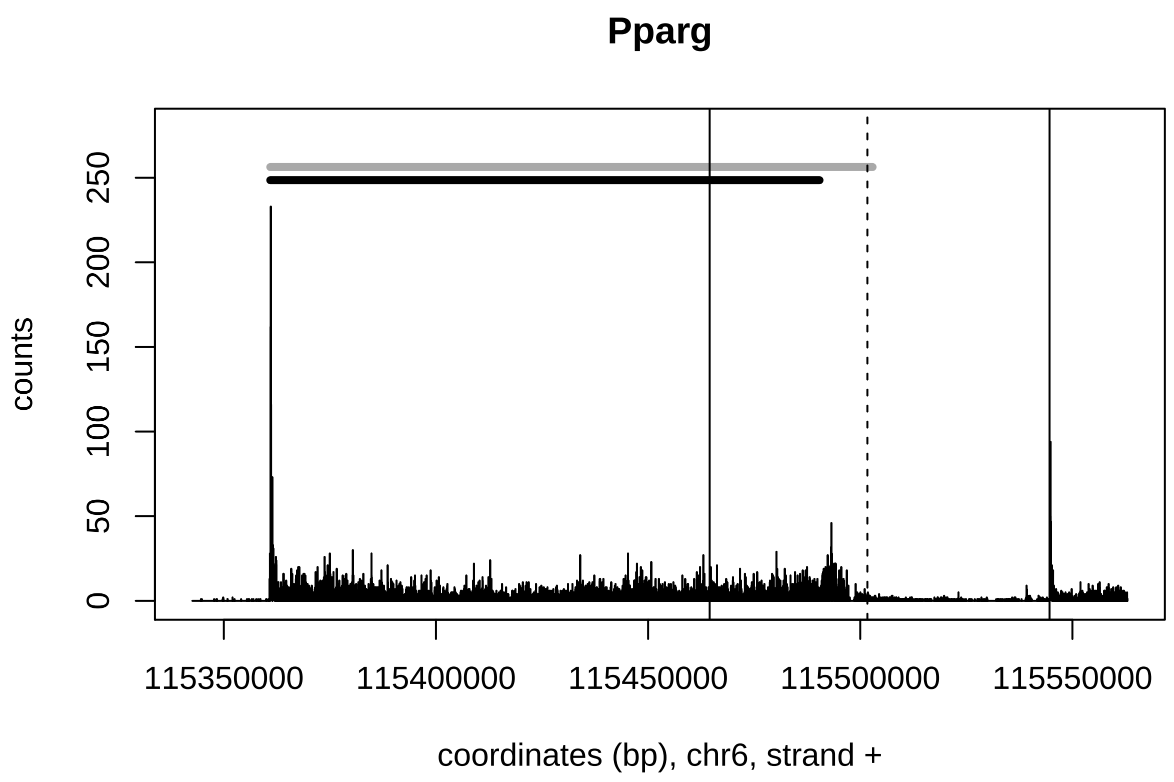
```
# check for the percentage of identified TTSs that match the search region boundry
TTS.boundary.match(coords=coords, bed.for.tts.eval=bed.for.tts.eval) # 0.2330275
#> [1] 0.2330275

# look at whether TTSs identified at the search boundry had clipped boundries
frac.bound.clip = TTS.boundary.clip(coords=coords, bed.for.tts.eval=bed.for.tts.eval)
frac.bound.clip$frac.clip # 0.9887025
#> [1] 0.9887025
frac.bound.clip$frac.noclip # 0.0112975
#> [1] 0.0112975

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



Finally, there are plotting functions for evaluating transcript coordinate inference performance. The tts.plot() function shows read data along with the TTS search region (solid vertical lines) and the spline peak search region (vertical dashed line). The inferred gene coordinates are indicated above the read data (gray) along with the largest interval annotation (black).

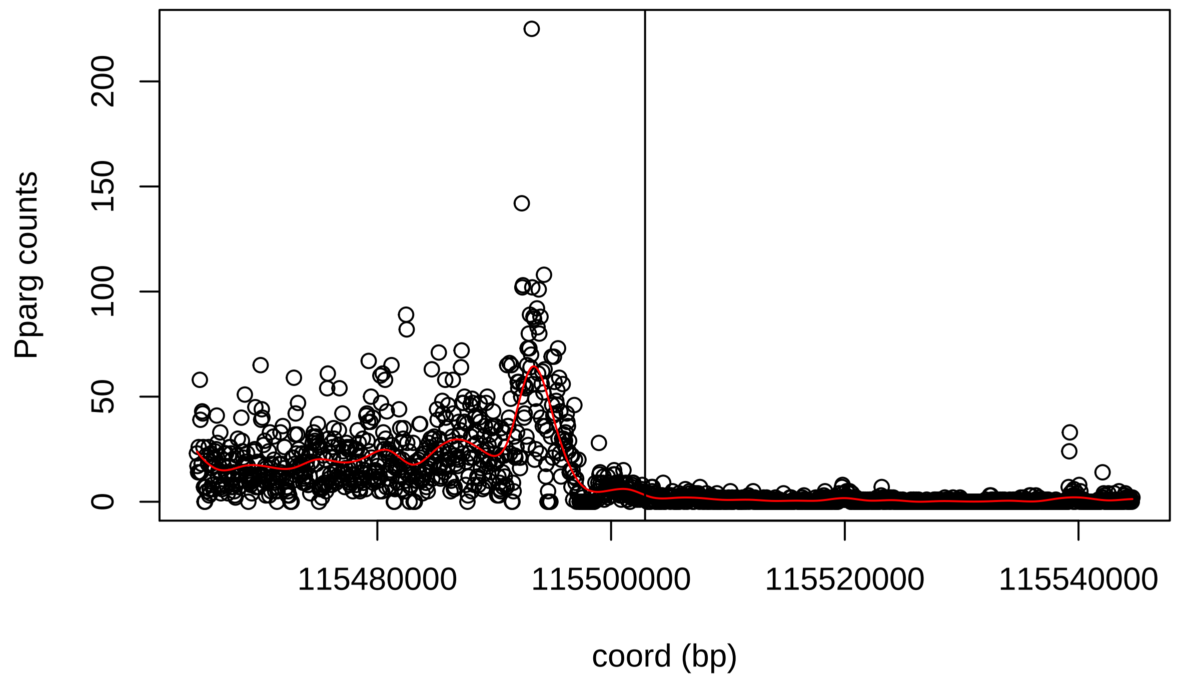
```
# plot new coord id
gene.end = bed.for.tts.eval
```

```
long.gene = largest.interval.bed
tts.plot(coords=coords, gene.end=gene.end, long.gene=long.gene,
         gene = "Pparg", xper=0.1, yper=0.2,
         bw.plus=bw.plus, bw.minus=bw.minus, bp.bin=5,
         frac.min=frac.min, frac.max=frac.max,
         add.to.end=add.to.end, tau.dist=tau.dist)
```



It is also possible to directly visualize how the spline fit (red) relates to the inferred TTS (vertical line).

```
#### plot tts curve
gene.end.plot(bed=bed.for.tts.eval, gene="Pparg",
              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 formatting and output

We use the information in chrom.sizes to filter the inferred coordinates and then we write the data to disk for subsequent visualization in the UCSC browser (see section 7 for details on formatting these data for the UCSC browser).

```r
# filter for chrom sizes - if any ends are greater than the species chromosome size,
# reduce the end to the size limit
# any negative starts will be set to 0
coords.filt = chrom.size.filter(coords=coords, chrom.sizes=chrom.sizes)
coords.filt$log
head(coords.filt$bed)

# process output, remove mitochondrial coordinates, and write out data
out = coords.filt$bed %>% select(chr,start,end,gene)
out = cbind(out, c(500), coords.filt$bed$strand) %>% as.data.frame(stringsAsFactors=F)
out = out[-which(out$chr == "chrM"),]
write.table(out,"inferredcoords.bed",quote=F,sep="\t",col.names=F,row.names=F)
```

The inferred and largest interval annotations can be visualized in parallel via a saved session with the UCSC genome browser https://genome.ucsc.edu/s/warrena%40virginia.edu/primaryTA_inf%2Dlng_20190801. Above the read data, largest interval annotations are illustrated with horizontal black bars, while inferred annotations are illustrated with gray.