bigWig

Luther Vucic [1]***, André Luis Matins*** [2]***, and Michael J. Guertin*** [3]

[1]James Madison University, Harrisonburg, Virginia
[2]Cornell University, Ithaca, New York
[3]University of Virginia, Charlottesville, Virginia

9 August 2020

Abstract

Querying of *bigWig* files in *R*

Package

bigWig 0.2.9

Contents

1 Prerequisites

The R *bigWig* libraries require an *R* version of >= 2.12.0.

2 Introduction

The *bigWig* package efficiently queries *bigWig* files over genomic intervals. The functions provide several counting variations, including over a region or step-wise. The functions can incorporate a mappability file, which determines areas of the genome that are not mappable at a specified K-mer and excludes them from calculations. Graphing functions are used to display data. The following definitions are used throughout the vignette:

- **Genomic interval** is the basic unit that all of these functions and is a segment of a genome file. It is defined by listing the chromosome [`chrom=`], starting index number [`start=`] and the ending index number [`end=`].
  - Example: `chrom ='chr1', start = 23000, end = 24000`
- **query** refers to the return of count metrics (raw, average, etc.) within genomic intervals. The terms **probe** and **bp** are used in conjunction with **query** to specify how *bigWig* values are treated.
  - **probe** refers to each bigWig entry that spans an interval.
  - **bp** or **base pair** is an individually indexed genomic position. In terms of counting, any *bp* function treats the value associated with each nucleotide position within a *bigWig* interval separately.
- **region** contains one or more genomic intervals, and at minimum include [`chrom=`], [`start=`], and [`end=`] values, with an optional [`strand=`] argument.
- **bed** and **bed6** are *R* data frames containing multiple genomic intervals. Only columns 1-3 are considered for *bed* operations, and column 6 is additionally passed for *bed6* operations–all other columns are ignored. See UCSC's description of BED file format. UCSC Genome
- **step** refers to dividing the genomic interval into equally sized sub-intervals. Note if the genomic interval is not a multiple of the step, an error will result.

3 Getting started

3.1 Installation

Since bigWig is not yet available on *bioconductor*, we can not use the basic `installation.packages('bigWig')`. Below are installation instructions from GitHub and locally stored source files.

3.1.1 From Github

The most up to date version of the `bigWig` pkg is located at bigWig. Using `devtools`, you can download and install `bigWig` from github directly.

```
#install devtools if necessary
install.packages("devtools")
library('devtools')
#location of bigWig package and subfolder
pkgLoc='andrelmartins/bigWig'
```

```
subFld='bigWig'
devtools::install_github(pkgLoc, subdir=subFld)
```

3.1.2 From local directory

Use the following commands to build from the source files.

```
setwd('bigWig-master')
system('R CMD INSTALL bigWig')
```

4 Usage

After installation load the *bigWig* package:

```
library(bigWig)
```

4.1 bigWig utilities

These a functions that load, unload, query and print the information that is in each bigWig.

4.1.1 bigWig format

bigWig files are genetic sequence fragments stored as indexed binary format. These files are not readily readable by humans, but the format allows for large continuous data to be stored compactly and accessed quickly.

4.1.2 load.bigWig

```
load.bigWig(filename, udcDir = NULL)
```

- arguments
  - `filename` [required] is a string, which is either the the local file directory or URL.
  - `udcDir` is a string which is the location for storing cached copies of remote files locally, while in use. These are destroyed when you unload the bigWig. If left as the default `udcDir = NULL`, then it uses `/tmp/udcCache`.

`load.bigWig` creates a `bigWig` class object in `R`. This object contains relevant information about the bigWig file and serves as a pointer to the underlying C object of the entire bigWig file. The only parameter required for this is a string of the location and filename. `udcDir` is only used if you want to keep the downloaded bigWig file locally if `filename` is a URL.

```
#load bigWig into variable bw
setwd('./bigWig')

bw=load.bigWig('../inst/extdata/bp.bigWig')
```

The all of the attributes of the object can be accessed using `atrributes` and each individual can be accessed via `$`

```
# list all attributes
attributes(bw)
## $handle_ptr
## <pointer: 0x60000105fb10>
##
## $names
##  [1] "version"          "isCompressed"     "isSwapped"
##  [4] "primaryDataSize"  "primaryIndexSize" "zoomLevels"
##  [7] "chroms"           "chromSizes"       "basesCovered"
## [10] "mean"             "min"              "max"
## [13] "std"
```

```
##
## $class
## [1] "bigWig"

#access individual attribute
bw$basesCovered
## [1] 15
```

4.1.3 print.bigWig

`print.bigWig(bw)` is used to print all of the attributes contained within the object.

```
print.bigWig(bw)
## bigWig
##  version: 4
##  isCompressed: yes
##  isSwapped: no
##  primaryDataSize: 90
##  primaryIndexSize: 6,204
##  zoomLevels: 2
##  chromCount: 1
##      chr1 248956422
##  basesCovered: 15
##  mean:  2.333333
##  min:  1
##  max:  4
##  std:  1.290994
```

- arguments
  - `bw` is the pointer of the underlying `C` object created in `load.bigWig`

4.1.4 unload.bigWig

```
unload.bigWig(bw)
```

- arguments
  - `bw` is the pointer of the underlying `C` object created in `load.bigWig`

Use `unload.bigWig(bw)` to destroy the C object and remove it from memory. This does not clear the `R` object. To do that use `rm()` or `remove()`

```
#destroy C object
unload.bigWig(bw)
ls()
##  [1] "bed"                "bed.step"           "bed6"
##  [4] "bedTSS"             "bedTSSwindow"       "bigwig.map"
##  [7] "bw"                 "bw.bp"              "bw.probes"
## [10] "bw.probes.Q"        "bw.splitprobes"     "bwMap"
## [13] "bwMap.left"         "bwMap.right"        "bwMinus"
## [16] "bwPlus"             "mat2"               "minusPRO"
## [19] "plusPRO"            "sizes.bed"          "step.bp.bw.probes"
```

```
## [22] "tss.matrix"           "tss.matrix.notStrand" "x"
## [25] "y"
#remove variable in R
remove(bw)
ls()
##  [1] "bed"                  "bed.step"             "bed6"
##  [4] "bedTSS"               "bedTSSwindow"         "bigwig.map"
##  [7] "bw.bp"                "bw.probes"            "bw.probes.Q"
## [10] "bw.splitprobes"       "bwMap"                "bwMap.left"
## [13] "bwMap.right"          "bwMinus"              "bwPlus"
## [16] "mat2"                 "minusPRO"             "plusPRO"
## [19] "sizes.bed"            "step.bp.bw.probes"    "tss.matrix"
## [22] "tss.matrix.notStrand" "x"                    "y"
```

4.2 query.bigWig

To demonstate the calculations performed by the `*Query.bigWig` functions we generated three *bigWig* files that have the same information at each position in the genome, but the files are structured differently (Figure 1).

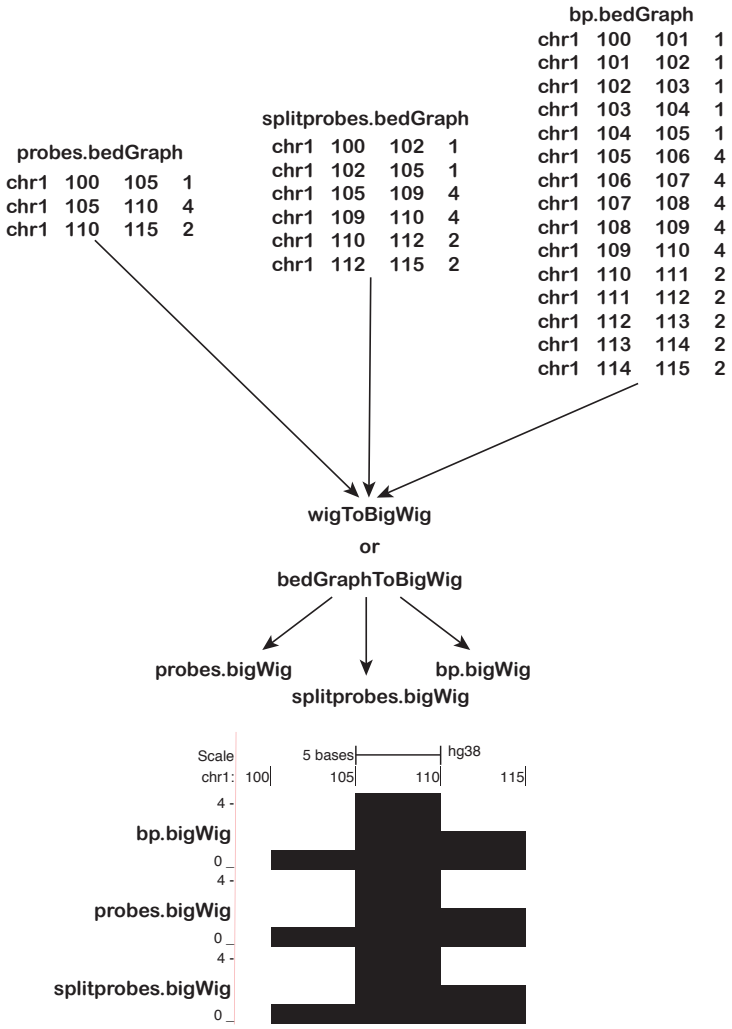

Figure 1: Structured bigWig files
Three bigWig files with identical values at each position are structured differently to later highlight the differences between *Query.bigWig functions.

```
query.bigWig(bw, chrom, start, end, clip = TRUE)
```

- arguments
  - `bw` is the pointer of the underlying `C` object created in `load.bigWig`
  - `chrom` is a string representing the chromosome to which the query interval belongs
  - `start` is an integer value defining the start of the query interval
  - `end` is an integer value defining the end of the query interval
  - `clip` is a logical value; if TRUE bigWig regions are clipped to the query interval.

4.2.1 bigWig file structure

`query.bigWig` allows you to search the *bigWig* files using chromosome string (`chrom='chr1'`) and genomic window (`start=1, end = 12000`), both are integers and end is inclusive meaning it searches up to and including `end`. The query results are printed to the command line. Note how the output of query reflects the original structure of the *bigWig* file (Figure 1). Each row that is output from a `query.bigWig` call is a genomic interval that is referred to as a `probe` in the relevant functions.

```
# load the three bigWigs
bw.bp = load.bigWig('../inst/extdata/bp.bigWig')
bw.probes = load.bigWig('../inst/extdata/probes.bigWig')
bw.splitprobes = load.bigWig('../inst/extdata/splitprobes.bigWig')

#note differences in the bigWig structures
query.bigWig(bw.probes, 'chr1', 100, 115)
##      start end value
## [1,]   100 105     1
## [2,]   105 110     4
## [3,]   110 115     2
query.bigWig(bw.splitprobes, 'chr1', 100, 115)
##      start end value
## [1,]   100 102     1
## [2,]   102 105     1
## [3,]   105 109     4
## [4,]   109 110     4
## [5,]   110 112     2
## [6,]   112 115     2
query.bigWig(bw.bp, chrom='chr1',start=100, end=115)
##       start end value
##  [1,]   100 101     1
##  [2,]   101 102     1
##  [3,]   102 103     1
##  [4,]   103 104     1
##  [5,]   104 105     1
##  [6,]   105 106     4
##  [7,]   106 107     4
##  [8,]   107 108     4
##  [9,]   108 109     4
## [10,]   109 110     4
## [11,]   110 111     2
## [12,]   111 112     2
## [13,]   112 113     2
## [14,]   113 114     2
## [15,]   114 115     2
```

The default behavior is to clip the bigWig intervals to the queried regions. The `bw.probes` variable and underlying *bigWig* structure can be used to highlight the `clip=` option.

```
query.bigWig(bw.probes, 'chr1', 104, 111, clip=FALSE)
##      start end value
## [1,]   100 105     1
## [2,]   105 110     4
```

```
## [3,]   110 115     2
query.bigWig(bw.probes, 'chr1', 104, 111, clip=TRUE)
##      start end value
## [1,]   104 105     1
## [2,]   105 110     4
## [3,]   110 111     2
```

The query can be set as a variable for storage.

```
bw.probes.Q = query.bigWig(bw.probes, 'chr1', 100, 115)
```

Access the array as an indexed array; the following returns the first row.

```
bw.probes.Q[1,]
## start   end value
##   100   105     1
```

Standard [X,Y] indexing returns the specified row and column.

```
bw.probes.Q[1,2]
## end
## 105
```

The genomic coordinate variable strings are keywords that can be used to access the respective columns.

```
bw.probes.Q[1,'start']
## start
##   100
```

4.3 bpQuery and probeQuery

This section outlines `*.bpQuery.bigWig` and `*.probeQuery.bigWig` functions to highlight their diffrences and commonalities. Both functions can incorporate `bwMap` files to account for the mappability of each position in the genome. `bwMap` files come from the calc_Mappability functions and will be discussed later on. They specify genomic interval mappability based on the sequence being repeated in the genome.

4.3.1 region query

FOR MANY FUNCTIONS THE PRESENCE OF STRAND IS CONFUSING.

The *bp* and *probe* query functions takes a region defined by `chrom`, `start` and `end` and returns the result of the operation on the counts.

```
region.bpQuery.bigWig(bw, chrom, start, end, strand = NA
                      op = "sum", abs.value = FALSE,
                      bwMap = NULL, gap.value = 0)

region.probeQuery.bigWig(bw, chrom, start, end,
                      op = "wavg", abs.value = FALSE,
                      gap.value = NA)
```

- arguments
  - `bw` is the pointer of the underlying `C` object created in `load.bigWig`
  - `chrom` is a string representing the chromosome to which the query interval belongs
  - `start` is an integer value defining the start of the query interval
  - `end` is an integer value defining the end of the query interval
  - `strand` + or - character indicating the strand of the supplied coordinates (bpQuery only)
  - `op` is a string representing the operation to perform on the interval.
    - `sum` adds all the counts
    - `avg` averages the counts
    - `min` finds the minimum value
    - `max` finds the maximum value
    - `wavg` weighted average of the values–only pertains to probeQuery
  - `abs.value` is a logical argument which determines if the absolute value of the input is performed before the `op`.
  - `gap.value` is an integer value that replaces areas that have no overlaps
  - `bwMap` a bigWig file of coordinates that cannot be uniquely mapped. Note that the sequence read length of the original FASTQ file should determine the k-mer mappability for this file

All `bpQuery` functions are insensitive to the structure of the original *bigWig* file, because each base position is evaluated separately. However, `probeQuery` functions consider each genomic interval as a separate entity, or *probe*, and evalates them separately. The following `region.probeQuery.bigWig` evaluations highlight the different outputs that result from differentially structured *bigWig* files that have identical values at each genomic position (see Figure 1). Note that the output for each command is the sum of the `value` column output from the first code chunk in Section 4.2.1.

```
region.probeQuery.bigWig(bw.probes, 'chr1', 100, 115, op = 'sum')
## [1] 7
region.probeQuery.bigWig(bw.splitprobes, 'chr1', 100, 115, op = 'sum')
## [1] 14
region.probeQuery.bigWig(bw.bp, 'chr1', 100, 115, op = 'sum')
## [1] 35
```

In contrast, the `region.bpQuery.bigWig` function considers each base position within each genomic interval input separately. These *bigWig* files have identical values at each position, so the calculations are identical.

```
region.bpQuery.bigWig(bw.probes, 'chr1', 100, 115, op = 'sum')
## [1] 35
region.bpQuery.bigWig(bw.splitprobes, 'chr1', 100, 115, op = 'sum')
## [1] 35
region.bpQuery.bigWig(bw.bp, 'chr1', 100, 115, op = 'sum')
## [1] 35
```

4.3.2 operations (op)

4.3.2.1 sum `op='sum'` As noted in Section 4.3.1, the `op='sum'` argument adds all the values of each probe or bp position in the specified genomic interval.

4.3.2.2 maximum `op='max'` Return the maximum value of the interval:

```
region.bpQuery.bigWig(bw.probes,'chr1', 100, 115, op='max')
## [1] 4
region.probeQuery.bigWig(bw.probes,'chr1', 100, 115, op='max')
## [1] 4
```

4.3.2.3 minimum `op='min'`

Return the minimum value of the interval:

```
region.bpQuery.bigWig(bw.probes, 'chr1', 100, 115, op='min')
## [1] 1
region.probeQuery.bigWig(bw.probes, 'chr1', 100, 115, op='min')
## [1] 1
```

4.3.2.4 average `op='avg'`

Return the average of the values of the interval: `bpQuery` uses the range of the query window as the denominator when calculating `avg`, while `probeQuery` will use the number of *bigWig* entries in the query region as the denominator.

```
region.bpQuery.bigWig(bw.splitprobes,'chr1', 100, 115, op='avg')
## [1] 2.333333
region.probeQuery.bigWig(bw.splitprobes,'chr1', 100, 112, op='avg')
## [1] 2.4
```

Notice the difference in the return of the average when there are no values at genomic position. The `bpQuery` counts the number of base pairs to use as the denominator of the average, but `probeQuery` uses the number of genomic intervals as the denominator.

```
region.bpQuery.bigWig(bw.probes,'chr1', 85, 115, op='avg')
## [1] 1.166667
region.probeQuery.bigWig(bw.probes,'chr1', 85, 115, op='avg')
## [1] 2.333333
```

4.3.2.5 weighted average `op='wavg'`

For `probe` functions, the average value can be weighted by the size of the genomic intervals. the `wavg` operation multiplies the values by the interval size before computing the average, therefore the average of the probes is weighted by their size. The *splitprobe* variable contains two genomic intervals that are distinct sizes and values, recall that chr1:102-105 is a genomic interval with the value *1* and chr1:105-109 has the value *4*. The `avg` operation weights these equally with a result of 2.5, as determined by: $(1 + 4)/2$. However, the `wavg` operation applies more weight to the wider genomic interval; each value is multiplied by the interval size and their sum is divided by the sum of the interval sizes: $((1*3) + (4*4)) / (3 + 4)$.

```
region.probeQuery.bigWig(bw.splitprobes,'chr1', 102, 109, op='avg')
## [1] 2.5
region.probeQuery.bigWig(bw.splitprobes,'chr1', 102, 109, op='wavg')
## [1] 2.714286
```

If a probe extends beyond the query interval, the probe will get truncated and the weight is the truncated size. In the example, the third probe is truncated from 5 to 1, so it is weighted one fifth of the first two probes that also span 5 bases, the value is calculated as follows: $((1*5) + (4*5) + (2*1)) / (5 + 5 + 1)$.

```
region.probeQuery.bigWig(bw.probes,'chr1', 100, 111, op='wavg')
## [1] 2.454545
```

4.3.2.6 abs.value = FALSE

If *bigWig* files contain negative values, the `abs.value=TRUE` option can be invoked to convert the output to absolute values.

4.3.2.7 gap.value

`gap.value` determines how the function handles instances where there is no data returned.

Notice that if you were to query chr1:80-90, that there would be no return.

```
query.bigWig(bw.probes,'chr1', 80, 90)
## NULL
```

Running `region.probeQuery.bigWig` on that genomic interval returns an NA (note `gap.value=NA` is the default for `probeQuery` functions) for all of the operations.The functionality is identiocal for `bpQuery.bigWig` operations, but the default is `gap.value=0`.

```
region.probeQuery.bigWig(bw.probes, 'chr1', 80, 90, op = 'sum')
## [1] NA
region.bpQuery.bigWig(bw.probes, 'chr1', 80, 90, op='sum')
## [1] 0
```

By adding `gap.value = 1` or any numeric value, the value is assigned to each query interval that has no intersecting probes. For both `probeQuery` and `bpQuery.bigWig` operations, the non-overlapping intervals that are assigned the `gap.value` are calculated as if the *bigWig* file had a single probe spanning the query interval coordinates with the associated `gap.value`.

```
region.probeQuery.bigWig(bw.probes, 'chr1', 80, 90, op = 'sum', gap.value=1)
## [1] 1
region.bpQuery.bigWig(bw.probes, 'chr1', 80, 90, op = 'avg', gap.value=100)
## [1] 100
```

4.4 BED utilities

These functions are used to manipulate *BED* files.

4.4.1 BED format

A standard three column *BED* file is a tab delimited file that consists of the chromosome name, the starting, and ending positions of the genomic interval. A BED6 file contains all of the BED columns plus 3 more: name, score, and strand. Only the strand column is considered for `bed6` functions described here. Strand defines whether the BED track interval refers to the + or - stand of DNA. More information can be found on UCSC website. BED files are saved with a .bed extension. The `bigWig` package operates on bed-formatted files that are loaded as data.frames into *R*.

4.4.2 Load BED file

Load a BED file is to use R's `read.table` function, which converts the tab delimited file into an *R* data.frame. First set file location to a variable like `bedloc` and read in the file. The `header` argument refers whether the columns are named in the first row. BED files don't

usually include headers so we can set `header=FALSE`. If track information or miscellaneous information lines. If there are lines prior ot the coordinate information, use the `skip=` argument to skip the number of lines that precede the genomic intervals.

```
bed=read.table('../inst/extdata/testBED1.bed',
             header=FALSE, sep='\t', stringsAsFactors=FALSE)
bed
##     V1  V2  V3
## 1 chr1 101 104
## 2 chr1 105 107
## 3 chr1 107 110
## 4 chr1 112 115
```

To create a *BED6* file, you need to define the following columns: `chrom`, `start`, `end`, `name`, `score` and `strand`. `bigWig` functions don't use `name` and `score`. In the following example we used place holders 'na' for `name` and 1 for `score`.

```
bed6=read.table('../inst/extdata/testBED1_strand.bed',
             header=FALSE, sep='\t', stringsAsFactors=FALSE)
bed6
##     V1  V2  V3 V4 V5 V6
## 1 chr1 101 104 na  1  +
## 2 chr1 101 104 na  1  -
## 3 chr1 105 107 na  1  +
## 4 chr1 107 110 na  1  +
## 5 chr1 112 115 na  1  -
```

4.4.3 BED transformations

These 3 functions take an original BED file and transform each rows start and end columns. The functions differ by the anchor point of the window. These functions are strand specific. For a *BED6* file, `threeprime.bed` and `fiveprime.bed` refers to upstream and downstream are relative to the strand information. The primary usage of theese transformations is to generate constant windows anchored on a genomic feature, such as a transcription start site or sequence motif.

```
center.bed(bed, upstreamWindow, downstreamWindow)

fiveprime.bed(bed, upstreamWindow, downstreamWindow)

threeprime.bed(bed, upstreamWindow, downstreamWindow)
```

- Arguments
  - `bed` is the input BED data.frame.
  - `upstreamWindow` is an integer number of bases to include upstream of the anchor point.
  - `downstreamWindow` is an integer number of bases to include downstream of the anchor point.

Anchor Point

The anchor point is different for each function.

`center.bed` uses the center of the original window. The difference between `end` and `start` is taken and divided by 2. If the difference is odd, you are left with a $X.5$, which is rounded down to X. The anchor point is the `start` + X.

`fiveprime.bed` uses the `start` as the anchor point for BED files and BED6 entires with a + in the sixth strand column. The `end` is the anchor for BED6 entires with a - in the strand column

`threeprime.bed` uses the `end` as the anchor point for BED files and BED6 entires with a + in the sixth strand column. The `start` is the anchor for BED6 entires with a - in the strand column

New Window

The new window is calculated by using the anchor point, `upstreamWindow` and `downstreamWindow`.

The new `start` is anchor point - `upstreamWindow`.

The new `end` is the anchor point + 1 + `downstreamWindow`.

Using the previously loaded bed6 file, we can test a few different scenarios.

```
bed6
##     V1  V2  V3 V4 V5 V6
## 1 chr1 101 104 na  1  +
## 2 chr1 101 104 na  1  -
## 3 chr1 105 107 na  1  +
## 4 chr1 107 110 na  1  +
## 5 chr1 112 115 na  1  -
```

Using the `center.bed` function and `upstreamWindow = 0` and `downstreamWindow = 0`, you can see the anchor point.

```
center.bed(bed6, upstreamWindow = 0, downstreamWindow = 0)
##     V1  V2  V3 V4 V5 V6
## 1 chr1 102 103 na  1  +
## 2 chr1 102 103 na  1  -
## 3 chr1 106 107 na  1  +
## 4 chr1 108 109 na  1  +
## 5 chr1 113 114 na  1  -
```

Equal and Positive Windows

Often when we query signal *bigWig* tracks around transcription factor binding sites or sequence motifs, the `upstreamWindow` and `downstreamWindow` are equal and positive.

```
center.bed(bed, upstreamWindow = 5, downstreamWindow = 5)
##     V1  V2  V3
## 1 chr1  97 108
## 2 chr1 101 112
## 3 chr1 103 114
## 4 chr1 108 119
```

The `start` values are all anchor point - 5 and the `end` values are all anchor point + 1 + 5.

fiveprime and threeprime

By setting `upstreamWindow = 0` and `downstreamWindow = 0`, you can see that the difference between `start` and `end` have no influence on the anchor point. The functions `fiveprime.bed` and `threeprime.bed` anchor the window as indicated by the strand. These transformations are most typically used when transforming gene annotation files to query *bigWig* files relative to features such as transcription start sites.

Changing the `strand` affects the anchor point:

- If `strand = '+'` while using `fiveprime.bed`
  - anchor point = original `start`
  - `start` = anchor point - `upstreamWindow`
  - `end` = anchor point + 1 + `downstreamWindow`
- If `strand = '-'` while using `fiveprime.bed`
  - anchor point = original `end`
  - `start` = anchor point - `downstreamWindow`
  - `end` = anchor point + 1 + `upstreamWindow`
- If `strand = '+'` while using `threeprime.bed`
  - anchor point = original `end`
  - `start` = anchor point - 1 - `upstreamWindow`
  - `end` = anchor point + `downstreamWindow`
- If `strand = '-'` while using `threeprime.bed`
  - anchor point = original `start`
  - `start` = anchor point - `downstreamWindow`
  - `end` = anchor point + 1 + `upstreamWindow`

```
fiveprime.bed(bed, upstreamWindow = 0, downstreamWindow = 0)
##     V1  V2  V3
## 1 chr1 101 102
## 2 chr1 105 106
## 3 chr1 107 108
## 4 chr1 112 113
threeprime.bed(bed, upstreamWindow = 0, downstreamWindow = 0)
##     V1  V2  V3
## 1 chr1 103 104
## 2 chr1 106 107
## 3 chr1 109 110
## 4 chr1 114 115
```

`fiveprime.bed` uses the 5' end or `start` as the anchor point, while `threeprime.bed` uses 3' or `end` for the anchor point.

Calculating the new window varies slightly. While `fiveprime.bed` follows `center.bed` by

- `start` = anchor point - `upstreamWindow`
- `end` = anchor point + 1 + `downstreamWindow`

`threeprime.bed` calculates the window by

- `start` = anchor point - 1 - `upstreamWindow`
- `end` = anchor point + `downstreamWindow`

Both of these function operate like `center.bed` other than the initial anchor point.

```
fiveprime.bed(bed, upstreamWindow = 1, downstreamWindow = 5)
##     V1  V2  V3
## 1 chr1 100 107
```

```
## 2 chr1 104 111
## 3 chr1 106 113
## 4 chr1 111 118
threeprime.bed(bed, upstreamWindow = 1, downstreamWindow = 5)
##     V1  V2  V3
## 1 chr1 102 109
## 2 chr1 105 112
## 3 chr1 108 115
## 4 chr1 113 120
```

If using a BED file without a `strand` column, `fiveprime.bed` and `threeprime.bed` assume that the `start` is the 5' end of the sequence.

```
fiveprime.bed(bed6, upstreamWindow=4, downstreamWindow=2)
##     V1  V2  V3 V4 V5 V6
## 1 chr1  97 104 na  1  +
## 2 chr1 102 109 na  1  -
## 3 chr1 101 108 na  1  +
## 4 chr1 103 110 na  1  +
## 5 chr1 113 120 na  1  -
threeprime.bed(bed6, upstreamWindow=4, downstreamWindow=2)
##     V1  V2  V3 V4 V5 V6
## 1 chr1  99 106 na  1  +
## 2 chr1  99 106 na  1  -
## 3 chr1 102 109 na  1  +
## 4 chr1 105 112 na  1  +
## 5 chr1 110 117 na  1  -
```

4.4.4 downstream, upstream

These two functions transform the BED file by taking the corresponding anchor point and the window.

```
downstream.bed(bed, downstreamWindow)

upstream.bed(bed, upstreamWindow)
```

- Arguments
  - bed the input BED data.frame.
  - upstreamWindow integer number of bases to include upstream of the anchor point.
  - downstreamWindow integer number of bases to include downstream of the anchor point.

`downstream.bed` uses the original `start` point [5'] as the anchor point.

- `start` = anchor point
- `end` = anchor point + downstreamWindow

`upstream.bed` uses the original `end` point [3'] as the anchor point.

- `start` = anchor point - upstreamWindow
- `end` = anchor point

```
downstream.bed(bed6,5)
##     V1  V2  V3 V4 V5 V6
## 1 chr1 101 106 na  1  +
## 2 chr1  99 104 na  1  -
## 3 chr1 105 110 na  1  +
## 4 chr1 107 112 na  1  +
## 5 chr1 110 115 na  1  -
upstream.bed(bed6,5)
##     V1  V2  V3 V4 V5 V6
## 1 chr1  96 101 na  1  +
## 2 chr1 104 109 na  1  -
## 3 chr1 100 105 na  1  +
## 4 chr1 102 107 na  1  +
## 5 chr1 115 120 na  1  -
```

For all transformations, negative numbers are allowed for `downstreamWindow` and `upstreamWindow`, but beware that downstream operations will fail if the start coordinate is greater than the end coordinate.

4.4.5 foreach

`foreach.bed` is a way to quickly apply a function across all rows of a bed file. This function is similar to running `apply`, but faster since it's tailored to the structure of a BED file and it avoids a lot of boilerplate code.

```
foreach.bed(bed, func, envir = parent.frame())
```

- Arguments
  - `bed` is a dataframe structured like a bed file with columns for `chrom`, `start` and `end`
  - `func` is the function to apply to each entry in `bed`. Function must have four arguments: `index`, `chrom`, `start`, `end` and `strand`. Index will be a one-based integer corresponding to the current BED line. Chrom is a character string with the chromosome name. Start and end are the coordinates for the current entry (remember that BED files are zero-based left-open intervals). Strand is a character string with the entry's strand ('+' or '-') or NA if the *bed* has less than 6 columns
  - Environment where the function is evaluated. Default value is `parent.frame()` which corresponds to the environment where the `foreach.bed` was called, giving access (through «-) to the local variables.

A simple example is to calculate the size of each window.

```
sizes.bed <- function(bed) {
  N = dim(bed)[1]
  sizes = vector(mode="integer", length=N)

foreach.bed(bed, function(i, chrom, start, end, strand) {
    sizes[i] <<- end - start
  })

  return(sizes)
}
```

```
sizes.bed(bed)
## [1] 3 2 3 3
```

Everything is wrapped into a function `sizes.bed`.

`N` returns the length of the `bed` file.

`sizes` creates a vector of length N of zeros. This will be used in the `foreach.bed` function as the return.

Then the `foreach.bed` function is called. The `bed` file is passed in, as well as the `function`.

`func` iterates through all *i*'s calculating the window size, `end - start`, and setting the corresponding place in the vector, `sizes[i]`, equal to the window size.

`sizes.bed` then returns the vector sizes. The result is a vector of length `N` of window sizes.

Obviously,`sizes = bed[,3] - bed[,2]` is much faster, but the function can be designed as complicated as necessary.

4.5 bed and bed6 region

bed.region

```
bed.region.bpQuery.bigWig(bw, bed, strand = NA,
                          op = "sum", abs.value = FALSE,
                          gap.value = 0, bwMap = NULL)
bed.region.probeQuery.bigWig(bw, bed, op = "wavg",
                          abs.value = FALSE, gap.value = NA)
```

- arguments
  - `bw` is the pointer of the underlying `C` object created in `load.bigWig`
  - `bed` is a dataframe structured like a bed file with columns for `chrom`, `start` and `end`
  - `strand` + or - character indicating the strand of the supplied coordinates (bpQuery only)
  - `op` is a string representing the operation to perform on the interval.
    - `sum` adds all the counts
    - `avg` averages the counts
    - `min` finds the minimum value
    - `max` finds the maximum value
    - `wavg` weighted average of the values–only pertains to probeQuery
  - `abs.value` is a logical argument which determines if the absolute value of the input is performed before the `op`.
  - `gap.value` is an integer value that replaces areas that have no overlaps

This function is an extension of `region.bpQuery.bigWig` and operates identically, except the the region `chrom`, `start`, and `end` is defined by the corresponding column of a *BED data.frame*. The output is a vector that is equal in length to the number of rows of the BED *data.frame*. The vectors's index corresponds to the *BED data.frame* row index.

```
# note: If you leave out op='', it will default to op='sum'
bed.region.bpQuery.bigWig(bw.bp, bed6)
## [1]  3  3  8 12  6
```

```
bed.region.probeQuery.bigWig(bw.bp, bed6)
## [1] 1 1 4 4 2
```

bed6.region

```
bed6.region.bpQuery.bigWig(bw.plus, bw.minus, bed6,
                op = "sum", abs.value = FALSE, gap.value = 0, bwMap = NULL)

bed6.region.probeQuery.bigWig(bw.plus, bw.minus, bed6,
                op = "wavg", abs.value = FALSE, gap.value = NA)
```

- arguments
  - `bw.plus` is the `R` pointer created in `load.bigWig` and refers to the plus strand
  - `bw.minus` is the `R` pointer created in `load.bigWig` and refers to the minus strand
  - `bed6` is a BED6 style data.frame that specifies a `strand` value in column 6
  - `op` is a string representing the operation to perform on the interval.
    - `sum` adds all the counts
    - `avg` averages the counts
    - `min` finds the minimum value
    - `max` finds the maximum value
    - `wavg` weighted average of the values–only pertains to probeQuery
  - `abs.value` is a logical argument which determines if the absolute value of the input is performed before the `op`.
  - `gap.value` is an integer value that replaces areas that have no overlaps
  - `with.attributes` is a logical argument that determines if the results are returned annotated with their source components and/or step size.

Using the `bw.plus` and `bw.minus` strands, we can evaluate a `bed6.region` function. First, refer to the query for each strand as a reference.

```
bwPlus=load.bigWig('../inst/extdata/bpPlus.bigWig')
bwMinus=load.bigWig('../inst/extdata/bpMinus.bigWig')
```

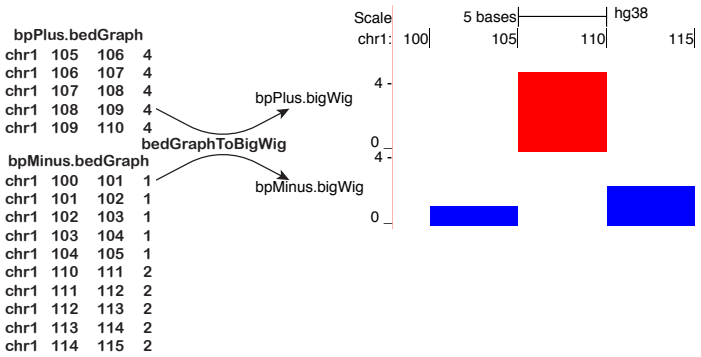

Figure 2: Stranded bigWig files
Two bigWig files contain values that are strand-specific to highlight the bed6 functions.

```
query.bigWig(bwMinus, chrom='chr1', start=100, end=115)
##       start end value
##  [1,]   100 101     1
```

```
##  [2,]   101 102     1
##  [3,]   102 103     1
##  [4,]   103 104     1
##  [5,]   104 105     1
##  [6,]   110 111     2
##  [7,]   111 112     2
##  [8,]   112 113     2
##  [9,]   113 114     2
## [10,]   114 115     2
query.bigWig(bwPlus, chrom='chr1', start=100, end=115)
##      start end value
## [1,]   105 106     4
## [2,]   106 107     4
## [3,]   107 108     4
## [4,]   108 109     4
## [5,]   109 110     4
```

Note that each interval is specified by the strand in column 6 of the *BED* data.frame, and each respective stranded *bigWig* file is queried for an interval.

```
#bed6 structure
bed6
##     V1  V2  V3 V4 V5 V6
## 1 chr1 101 104 na  1  +
## 2 chr1 101 104 na  1  -
## 3 chr1 105 107 na  1  +
## 4 chr1 107 110 na  1  +
## 5 chr1 112 115 na  1  -
bed6.region.bpQuery.bigWig(bwPlus, bwMinus, bed6)
## [1]  0  3  8 12  6
bed6.region.probeQuery.bigWig(bwPlus, bwMinus, bed6)
## [1] NA  1  4  4  2
```

4.6 Step through a region

The following functions operate over defined steps and is described by `step=` argument. In a given region [`start=1` and `end=10`] and a `step=5`, the function will create intervals of 5. In this example, it will run on [`start=1`, `end=5`] and [`start=6`, `end=10`]. The `probeQuery` and `bpQuery` functions coupled with `step` have the same behavior as previously described, but they operate on each step interval.

```
step.bpQuery.bigWig(bw, chrom, start, end, step,
                    strand = NA, op = "sum", abs.value = FALSE, gap.value = 0,
                    bwMap = NULL, with.attributes = TRUE)

step.probeQuery.bigWig(bw, chrom, start, end, step,
                    op = "wavg", abs.value = FALSE, gap.value = NA,
                    with.attributes = TRUE)
```

- arguments
  - `bw` is the pointer of the underlying `C` object created in `load.bigWig`

- `chrom` is a string representing the chromosome to which the query interval belongs
- `start` is an integer value defining the start of the query interval
- `end` is an integer value defining the end of the query interval
- `op` is a string representing the operation to perform on the interval.
  - `sum` adds all the counts
  - `avg` averages the counts
  - `min` finds the minimum value
  - `max` finds the maximum value
  - `wavg` weighted average of the values–only pertains to probeQuery
- `step` is the step size in base pairs
- `abs.value` is a logical argument which determines if the absolute value of the input is performed before the `op`.
- `gap.value` is an integer value that replaces areas that have no overlaps
- `with.attributes` is a logical argument that determines if the results are returned annotated with their source coordiantes and step size.

The *step* functions run through the range provide breaking it up into equal size steps as defined by `step =`. The length of the range [`end-start`] has to be a multiple of the step. For example if `start=1` and `end=21`, the length of the range is 20. This allows for `step = [1,2,4,5,10,20]`. The return is the value of the operation over that step. So if `step =2` and `op = 'min'`, then the return would be 10 minimum values. If `step = 5` and `op = 'max'`, the return will be a 4 element array of the maximum value in the step. The `avg` operation of `step.bpQuery` and `step.probeQuery` is identical to Section 4.3.2.4, in that the denominator is the step size (i.e. the width of each query region) for `step.bpQuery` and the number of intervals present in the *bigWig* in each step interval for`step.probeQuery`. For `step.probeQuery` function, the `wavg` operation handles step windows that span probes as previously described in Section 4.3.2.5. By invoking `with.attributes`, the chromosome, start, end, and step and can be extracted.

Below we *step query* over the 15 bp interval `start=100`, `end=115` and a `step=5`.

```
step.probeQuery.bigWig(bw.probes, 'chr1', 100, 115, op = 'sum', step=5)
## [1] 1 4 2
## attr(,"chrom")
## [1] "chr1"
## attr(,"start")
## [1] 100
## attr(,"end")
## [1] 115
## attr(,"step")
## [1] 5
step.bpQuery.bigWig(bw.probes, 'chr1', 100, 115, op = 'sum', step=5)
## [1]  5 20 10
## attr(,"chrom")
## [1] "chr1"
## attr(,"start")
## [1] 100
## attr(,"end")
## [1] 115
## attr(,"step")
## [1] 5

step.bp.bw.probes = step.bpQuery.bigWig(bw.probes, 'chr1', 100, 115,
```

```
        op = 'sum', step=5)
        attributes(step.bp.bw.probes)$step
        ## [1] 5
```

bed.step

The `bed.step` function operates like the `bed.region` function, but the intervals are specified in the *BED* data.frame.

```
        bed.step.bpQuery.bigWig(bw, bed, step,
                          strand = NA, op = "sum", abs.value = FALSE, gap.value = 0,
                          bwMap = NULL, with.attributes = TRUE, as.matrix = FALSE)

        bed.step.probeQuery.bigWig(bw, bed, step,
                          op = "wavg", abs.value = FALSE, gap.value = NA,
                          bwMap = NULL, with.attributes = TRUE, as.matrix = FALSE)
```

- arguments
  - `bw` is the pointer of the underlying `C` object created in `load.bigWig`
  - `bed` is a dataframe structured like a bed file with columns for `chrom`, `start` and `end`
  - `op` is a string representing the operation to perform on the interval.
    - `sum` adds all the counts
    - `avg` averages the counts
    - `min` finds the minimum value
    - `max` finds the maximum value
    - `wavg` weighted average of the values–only pertains to probeQuery
  - `abs.value` is a logical argument which determines if the absolute value of the input is performed before the `op`.
  - `gap.value` is an integer value that replaces areas that have no overlaps
  - `with.attributes` is a logical argument that determines if the results are returned annotated with their source components and/or step size.
  - `as.matrix` is a logical argument that will return the results in a matrix format. The *BED* data.frame has to be a fixed window for all entries.

These functions are an extension of `step.bpQuery.bigWig` and `step.probeQuery.bigWig` and they operate identically, except the the region `chrom`, `start`, and `end` is defined by the corresponding column of a *BED data.frame*. The regions in the *BED data.frame* need to be exact multiples of the step. The regions defined within the *BED* file do not need to be the same size, unless `as.matrix = TRUE`. The output is a list of vectors, one per query *BED* interval. An additional argument option is to return the data as a matrix. The matrix argument will only return a matrix if the interval widths in the *BED* data.frame are identical for each row. This is often useful when querying a fixed window around the center of a genomic feature, such as a transcription factor binding site or transcription start site.

```
        #generate a bed
        bed.step=data.frame('chr1',100,106)
        bed.step[2,] = c('chr1',109,115)
        colnames(bed.step)=c('chrom', 'start', 'end')

        bed.step.bpQuery.bigWig(bw.splitprobes, bed.step, step = 2)
```

```
## [[1]]
## [1] 2 2 5
## attr(,"chrom")
## [1] "chr1"
## attr(,"start")
## [1] 100
## attr(,"end")
## [1] 106
## attr(,"step")
## [1] 2
##
## [[2]]
## [1] 6 4 4
## attr(,"chrom")
## [1] "chr1"
## attr(,"start")
## [1] 109
## attr(,"end")
## [1] 115
## attr(,"step")
## [1] 2
bed.step.bpQuery.bigWig(bw.splitprobes, bed.step, step = 2,
                        as.matrix=TRUE)
##      [,1] [,2] [,3]
## [1,]    2    2    5
## [2,]    6    4    4
## attr(,"step")
## [1] 2
```

bed6.step

The `bed6.step` function operates like the `bed6.region` function, but the intervals and `strand` information are specified in the *BED* data.frame. The `follow.strand` argument is introduced for the `bed6` step function. `follow.strand` reverses the direction of `-` strand output. This is commonly set to TRUE when the specific genomic feature in the *bed6* file has inherent strandedness. It is useful to know how the counts relate to the orientation of the feature, such as a sequence motif or transcription start site.

```
bed6.step.bpQuery.bigWig(bw.plus, bw.minus, bed6, step,
                        op = "sum", abs.value = FALSE, gap.value = 0,
                        bwMap = NULL, with.attributes = TRUE,
                        as.matrix = FALSE, follow.strand = FALSE)

bed6.step.probeQuery.bigWig(bw.plus, bw.minus, bed6, step,
                        op = "wavg", abs.value = FALSE, gap.value = NA,
                        with.attributes = TRUE, as.matrix = FALSE,
                        follow.strand = FALSE)
```

- arguments
  - `bw.plus` is the `R` pointer created in `load.bigWig` and refers to the plus strand
  - `bw.minus` is the `R` pointer created in `load.bigWig` and refers to the minus strand
  - `chrom` is a string referring to what chromosome is referenced

- `start` is an integer value designation the starting position
- `end` is an integer value designation the ending position
- `op` is a string representing the operation to perform on the step.
  - `sum` adds all the counts
  - `avg` averages the counts
  - `min` finds the smallest count
  - `max` finds the largest count
- `abs.value` is a logical argument which determines if the absolute value of the input is performed before the `op`.
- `gap.value` is an integer value that replaces areas that have no overlaps
- `with.attributes` is a logical argument that determines if the results are returned annotated with their source components and/or step size.
- `as.matrix` is a logical argument that will return the results in a matrix format. The *BED* data.frame has to be a fixed window for all entries.
- `follow.strand` is a logical value; if TRUE, in 'BED' type queries, the result is a matrix, otherwise it's a list of vectors, one per query 'BED' entry.

To highlight the `as.matrix` and `follow.strand` functionality, we sum the signal of PRO-seq *bigWig* files around transcription start sites. The `as.matrix` option returns a matrix and each row corresponds to the respective row of the *bed6* data.frame and each column is a window the size of the step (10) that spans the 500 base pair interval.

```
plusPRO=load.bigWig('../inst/extdata/plusPRO.bigWig')
minusPRO=load.bigWig('../inst/extdata/minusPRO.bigWig')
bedTSS=read.table(gzfile("../inst/extdata/TSS.bed.gz"))

bedTSSwindow=fiveprime.bed(bedTSS, upstreamWindow = 249,
                  downstreamWindow = 250)

tss.matrix = bed6.step.bpQuery.bigWig(plusPRO, minusPRO, bedTSSwindow,
                  step = 10, as.matrix=TRUE, follow.strand=TRUE)
colnames(tss.matrix) = seq(-245, 254, by = 10)
rownames(tss.matrix) = bedTSSwindow[,5]
#filter out genes that have no signal
tss.matrix=tss.matrix[rowSums(tss.matrix) != 0,]
head(tss.matrix)
##            -245 -235 -225 -215 -205 -195 -185 -175 -165 -155 -145 -135 -125
## WASH7P      0.3    0    0    0    0    0  0.0    0  0.0    0    0    0    0
## MIR6859-1   0.0    0    0    0    0    0  0.0    0  0.0    0    0    0    0
## FO538757.1  0.0    0    0    0    0    0  0.0    0  0.3    0    0    0    0
## MIR6859-2   0.0    0    0    0    0    0  0.0    0  0.0    0    0    0    0
## MTND1P23    0.0    0    0    0    0    0  0.0    0  0.0    0    0    0    0
## MTND2P28    0.0    0    0    0    0    0  1.1    0  3.6    0    0    0    0
##            -115 -105 -95 -85 -75 -65 -55 -45 -35 -25 -15 -5 5 15 25  35 45
## WASH7P        0    0   0   0   0   0   0   0   0 0.0   0  0 0  0  0 0.0  0
## MIR6859-1     0    0   0   0   0   0   0   0   0 0.0   0  0 0  0  0 0.0  0
## FO538757.1    0    0   0   0   0   0   0   0   0 0.3   0  0 0  0  0 0.0  0
## MIR6859-2     0    0   0   0   0   0   0   0   0 0.0   0  0 0  0  0 0.6  0
## MTND1P23      0    0   0   0   0   0   0   0   0 0.0   0  0 0  0  0 0.0  0
## MTND2P28      0    0   0   0   0   0   0   0   0 0.0   0  0 0  0  0 0.0  0
##            55  65 75 85 95 105 115 125 135 145 155 165 175 185 195 205 215
## WASH7P      0 0.0  0  0  0   0   0 0.0 0.0   0   0   0   0 0.0 0.0   0   0
```

```
## MIR6859-1   0 0.3  0  0  0   0   0 0.0 0.0   0   0   0   0 0.0 0.0   0   0
## FO538757.1  0 0.0  0  0  0   0   0 0.0 0.0   0   0   0   0 0.0 0.0   0   0
## MIR6859-2   0 0.0  0  0  0   0   0 0.0 0.0   0   0   0   0 0.0 0.0   0   0
## MTND1P23    0 0.0  0  0  0   0   0 1.1 5.1   0   0   0   0 2.8 2.2   0   0
## MTND2P28    0 0.0  0  0  0   0   0 0.0 0.0   0   0   0   0 0.0 0.0   0   0
##            225 235 245
## WASH7P       0   0   0
## MIR6859-1    0   0   0
## FO538757.1   0   0   0
## MIR6859-2    0   0   0
## MTND1P23     0   0   0
## MTND2P28     0   0   0
```

The `follow.strand` argument orients the *bed6* intervals so that downstream and upstream are relative to the `strand` in column 6. Using PRO-seq data below with `follow.strand=TRUE`, we observe a RNA Polymerase paused peak just downstream of the TSS.

```
plot(seq(-249, 250, by = 10),colMeans(tss.matrix),
                  xlab = 'stranded TSS position',
                  ylab = 'bigWig intensity', type = "l")
```

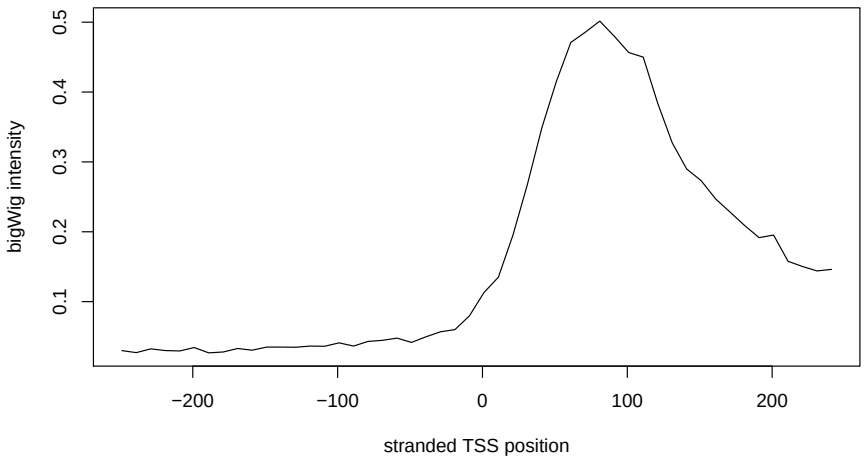

Figure 3: Feature-stranded composite profile
The RNA Polymerase peak results from the accumulation of reads that originate from the coding strand orientation. Therefore, it makes sense to orient the reads relative to the TSS position.

However, if `follow.strand` is not invoked, the - and + strand signals accumulate upstream and downstream, respectively, of the central position.

```
tss.matrix.notStrand = bed6.step.bpQuery.bigWig(plusPRO, minusPRO,
                  bedTSSwindow, step = 10, as.matrix=TRUE,
                  follow.strand=FALSE)
tss.matrix.notStrand = tss.matrix.notStrand[rowSums(tss.matrix.notStrand) != 0,]
```

```
plot(seq(-249, 250, by = 10),colMeans(tss.matrix.notStrand),
                   xlab = 'absolute TSS position',
                   ylab = 'bigWig intensity', type = "l")
```

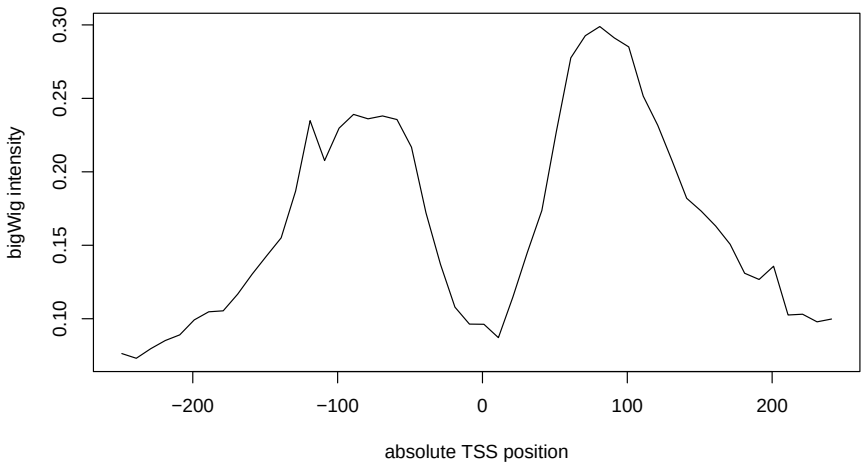

Figure 4: Absolute-stranded composite profile
Since a comparable number of genes are oriented in the plus and minus orientatino relative to teh reference genome, the peak downstream of the TSS is split if absolute genomic coordinate is followed.

4.7 Mappability

When counting reads within intervals, it can be inappropriate to include a count of zero if the genomic coordinate is not uniquely mappable at a given read length. By convention, a mappability *bigWig* indicates mappable positions as 0 and unmappable as 1. Raw data in underlying *bigWig* file marks a position as unmappable if the read that starts at that position is unmappable in the plus strand (obviously dependent on read length).

```
load.bwMap(filename, read.len, read.left.edge,
                          threshold.fraction = 0, udcDir=NULL)
```

- arguments
  - `filename` is a character string giving the name of the file to load. It can be a valid URL.
  - `read.len` is a integer number representing the length (in base pairs) of k-mers (or sequence read length) for which the mappability file was constructed.
  - `read.left.edge` is a logical value indicating if a read is represented by it's left-most edge (5' position) or it's right-most edge (3' position).
  - `threshold.fraction` is a numeric value indicating the maximum fraction of un-mappable bases in a query region for that region to still be considered mappable (Default = 0).
  - `udcDir` is a character string giving the name of the folder to use as a local cache when accessing remote files. Set to NULL to use the default location (/tmp/udcCache).

```
bigwig.map=load.bwMap('../inst/extdata/bwMap.bigWig', read.len = 30,
                      read.left.edge=FALSE, threshold.fraction = 0.19)
#structure of the bigWig:
#chr1   105 106 1
#chr1   107 108 1
#chr1   112 113 1


#y=load.bigWig('bwMap.bigWig')
#bwPlus=load.bigWig('bpPlus.bigWig')
#bwMinus=load.bigWig('bpMinus.bigWig')
#bed6=read.table('testBED1_strand.bed', header=FALSE, sep='\t', stringsAsFactors=FALSE)

#bed6.region.bpQuery.bigWig(bwPlus, bwMinus, bed6, bwMap=x)
#bed6.region.bpQuery.bigWig(bwPlus, bwMinus, bed6, bwMap=y)
```

```
unload.bwMap(bwMap)
region.bpQuery.bwMap(bwMap, chrom, start, end, strand, op = "thresh")

bed6.region.bpQuery.bwMap(bwMap, bed6, op = "thresh")

step.bpQuery.bwMap(bwMap, chrom, start, end, step, strand,
                   op = "thresh", with.attributes = TRUE)

bed6.step.bpQuery.bwMap(bwMap, bed6, step,
                   op = "thresh", with.attributes = FALSE, as.matrix = FALSE)
```

- arguments
  - `bwMap` is a saved bigWig Mappability object
  - `chrom` is a string representing the chromosome to which the query interval belongs
  - `start` is an integer value defining the start of the query interval
  - `end` is an integer value defining the end of the query interval
  - `strand` + or - character indicating the strand of the supplied coordinates (bpQuery only)
  - `op` is a string representing the operation to perform on the interval.
    - `sum` adds all the counts
    - `avg` averages the counts
    - `thresh` threshold returns a value of 1 (unmappable) or 0 (mappable) for the interval. If the average value in the interval is greater than or equal to the threshold, then the interval value is 1, or unmappable; otherwise the value is 0.
  - `bed6` is a BED6 style data.frame that specifies a `strand` value in column 6
  - `step` is the step size in base pairs
  - `with.attributes` is a logical argument that determines if the results are returned annotated with their source components and/or step size.
  - `as.matrix` is a logical argument that will return the results in a matrix format. The *BED* data.frame has to be a fixed window for all entries.

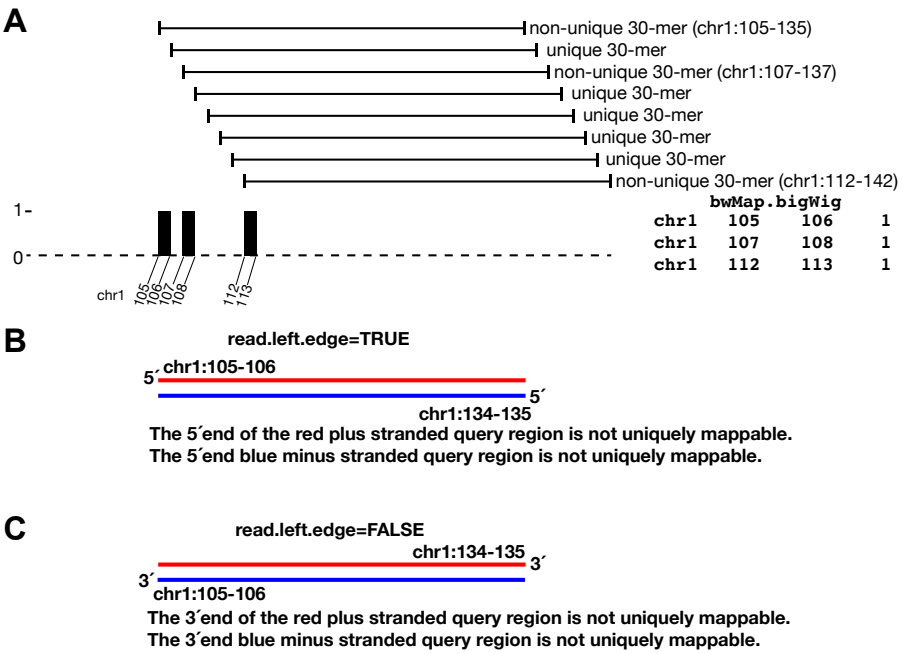

Figure 5: Mappability bigWig files
A) Mappability bigWig files contain a 1 at each postion in the genome that is not uniquely mappable. Note that the bigWig specifies the start of the k-mer (here a 30-mer) that corresponds to the plus strand sequence. B) The mappability bigWig file is often used in conjunction with typical bigWig data files. If the bigWig data file specifies the left-most or right-most base of the original sequence read, then the corresponding mappability file needs to specify the appropriate 'read.left.edge' option. By doing so, the plus and minus reads are appropriately shifted by the mappability k-mer size.

The following four queries all refer to the unmappable position chr1:105-106 in the *bwMap.bigWig* file. The query region specifying this signal is either chr1:105-106 or 30 bases away, at chr1:134-135, depending upon `read.left.edge` argument and `strand` argument.

```
bwMap.right=load.bwMap('../inst/extdata/bwMap.bigWig',
                       read.len = 30, read.left.edge=FALSE)
bwMap.left=load.bwMap('../inst/extdata/bwMap.bigWig',
                       read.len = 30, read.left.edge=TRUE)


region.bpQuery.bwMap(bwMap.left, "chr1", 134, 135,
                       strand = "-", op = "sum")
## [1] 1
region.bpQuery.bwMap(bwMap.left, "chr1", 105, 106,
                       strand = "+", op = "sum")
## [1] 1

region.bpQuery.bwMap(bwMap.right, "chr1", 105, 106,
                       strand = "-", op = "sum")
## [1] 1
region.bpQuery.bwMap(bwMap.right, "chr1", 134, 135,
```

```
                                strand = "+", op = "sum")
## [1] 1
```

The operation that in introduced with mappability function is `thresh`. It is reasonable to consider intervals as mappable only if the average mappability of an interval exceeds a threshold. Here `thresh` refers to the average *unmappability*, since a value of 1 indicates the position is not mappable. Therefore, a threshold of 0.2 means that if 20% or more of the positions in the interval are not mappable, then the entire interval is considered to be unmappable.

```
bwMap=load.bwMap('../inst/extdata/bwMap.bigWig',
                 read.len = 30, read.left.edge=FALSE, threshold.fraction = 0.2)
region.bpQuery.bwMap(bwMap, "chr1", 130, 135, strand = "+", op = "avg")
## [1] 0.2
region.bpQuery.bwMap(bwMap, "chr1", 130, 136, strand = "+", op = "avg")
## [1] 0.1666667
region.bpQuery.bwMap(bwMap, "chr1", 130, 135, strand = "+", op = "thresh")
## [1] 1
region.bpQuery.bwMap(bwMap, "chr1", 130, 136, strand = "+", op = "thresh")
## [1] 0
```

The functions `bed6.region.bpQuery.bwMap`, `step.bpQuery.bwMap`, and `bed6.step.bpQuery.bwMap` have the same extended functionality as the corresponding `bigWig` functions described above, but intervals and step intervals are treated as described for `region.bpQuery.bwMap`.

4.8 Profiles

Profiles are a group of functions that either calculate the quantile cutoff or confidence interval statistic. `metaprofile.bigWig` function creates a class object that can be passed on to the matrix scaling or plotting functions.

```
quantiles.metaprofile(mat, quantiles = c(0.875, 0.5, 0.125))

subsampled.quantiles.metaprofile(mat, quantiles = c(0.875, 0.5, 0.125),
            fraction = 0.10, n.samples = 1000)

confinterval.metaprofile(mat, alpha = 0.05)

bootstrapped.confinterval.metaprofile(mat, alpha = 0.05, n.samples = 300)

metaprofile.bigWig(bed, bw.plus, bw.minus = NULL, step = 1, name = "Signal",
           matrix.op = NULL, profile.op = subsampled.quantiles.metaprofile, ...)
```

- arguments
  - `mat`the input data matrix; each row corresponds to a query region, columns to steps. Created from functions that have `as.matrix=true`
  - `quantiles`vector of size three with top, middle and bottom quantile breaks to use in creating the summary profile.
  - `fraction`fraction of the data (query regions) to include in each subsample.
  - `n.samples`number of data samples to generate.
  - `alpha`alpha value for confidence intervals (confidence level = 1 - alpha).
  - `bed`the input BED data.frame defining the set of query regions.

- `bw.plus`either an R object of class 'bigWig' or a character vector containing the prefix and suffix to the path of each bigWig fragment (path =).
- `bw.minus`same as 'bw.plus', but for use with minus strand queries.
- `step`step size in base pairs.
- `name`character vector describing the data.
- `matrix.op`matrix scalling function to apply to the data.
- `profile.op`summary profile funcion.
- `...`extra arguments to be passed to matrix.op and/or profile.op.

The main input for all of these functions is `mat`. This particular matrix of integers is a of y rows and x columns. The integers represent the result of the operation performed on the window provided by a bed file. Each row in the bed file is a row in the matrix [y]. If there is more than 1 column, this means that the bed file was processed with a `step` attribute.

- Functions that can produce a matrix output, as.matrix=TRUE', are:
  - `bed.step.bpQuery.bigWig`
  - `bed.step.probeQuery.bigWig`
  - `bed6.step.bpQuery.bigWig`
  - `bed6.step.probeQuery.bigWig`

4.8.1 Quantiles

`quantiles.metaprofile` invokes `R's` `quantile` function on the integer in the matrix for each `quantile`.

We pass *tss.matrix*, to `quantiles.metaprofile`

```
quantiles.metaprofile(tss.matrix, quantiles = c(0.95, 0.5, 0.05))
```

The result of `quantiles.metaprofile` is a list of `quantile` values for the number and step size.

4.8.2 Subsampled

The `subsampled.quantiles.metaprofile` function returns values like quantiles except that it takes random subsamples of the original `mat` and the applies `quantiles.metaprofile` to the new matrix.

```
subsampled.quantiles.metaprofile(tss.matrix,
                  quantiles = c(0.875, 0.5, 0.125), fraction = 0.90,
                  n.samples = 5000)
```

4.8.3 Confidence Interval

`confinterval.metaprofile` is used to calculate a confidence intervals.

```
confinterval.metaprofile(tss.matrix, alpha = 0.05)
```

The result is a list of confidence interval values for each step for the given `alpha` value. There are 3 different levels of confidence intervals: Top, Middle and Bottom. Each of these are based on 2 values. The population mean, which is the mean of each column in `mat`. Then the delta, which is

$delta = P(1 - \alpha/2) * SE$

SE is the Standard Error of the column.

Using this delta and the means

$Top = mean + delta$ $Middle = mean$ $Bottom = mean - delta$

4.8.4 Bootstrap

`bootstrapped.confinterval.metaprofile` The bootstrap method produces a confidence interval like `confinterval.metaprofiles` except that it uses multiple samples to form a distribution and from this we can use the Central Limit Theorem to determine the confidence interval.

```
bootstrapped.confinterval.metaprofile(tss.matrix, alpha = 0.05, n.samples = 300)
```

This tends to be a more robust calculation of the confidence interval. The more `n.samples` provides a better estimation.

4.8.5 metaprofile

`metaprofile.bigWig` creates a class object of the data. That will be used in `plot.profile.bigWig`.

So, if we wanted to run `quantiles.metaprofile` on the bigWig, `profile.op = quantiles.metaprofile`. `matrix.op = NULL` will be discussed in another section. tss.matrix = bed6.step.bpQuery.bigWig(plusPRO, minusPRO, bedTSSwindow, step = 10, as.matrix=TRUE, follow.strand=TRUE)

```
metaprofile.bigWig(bedTSSwindow, plusPRO, bw.minus = minusPRO,
                   step = 10, name = "Signal", matrix.op = NULL,
                   profile.op = quantiles.metaprofile)
```

This function automatically creates the `mat` variable and will use the default values for the rest of the inputs. In the case of `bootstrapped.confinterval.metaprofile`, to change `alpha=0.05` and `n.samples=300` you would have to pass new inputs of `alpha=0.05`, and `n.samples=1000`.

```
metaprofile.bigWig(bedTSSwindow, plusPRO, bw.minus = minusPRO,
                   step = 10, name = "Signal", matrix.op = NULL,
                   profile.op = bootstrapped.confinterval.metaprofile,
                   alpha=0.05, n.samples=1000)
```

4.9 Matrix Scaling

These functions will scale a matrix depending on which method is used.

- arguments
  - mat is the input data matrix; each row corresponds to a query region, columns to steps
  - step is step size in base pairs
  - libSize is total library mapped read count
  - na.on.zero is logical indicating if steps with zero counts should be marked as NA

4.9.1 RPKM

RPKM [Reads Per Kilobase of transcript per Million mapped reads]. This function will scale everything by a factor of

The `libSize` can be calculated from attributes in the loaded *bigWig* object. The product bw$basesCovered * bw$mean is the number of raw reads in the *bigWig* if it is a counts-based *bigWig* file.

```
# Original mat
mat

rpkm.scale(mat, step=50000, libSize=1000000)


bwPlus$basesCovered * bwPlus$mean
```

4.9.2 Density to One

`densityToOne` is a scaling factor that takes each cell in a row of the matrix and divides it by the sum of each row and.

```
densityToOne.scale(mat, na.on.zero = TRUE)
```

The `na.on.zero = TRUE` input is used if you want NAs to populate the matrix row when the `sum(row)=0`. This would happen because dividing by 0 will result in NA. Otherwise if you 0 to replace NA then `na.on.zero=FALSE` should be used.

```
#Original Matrix
mat1

densityToOne.scale(mat1, na.on.zero = TRUE)
densityToOne.scale(mat1, na.on.zero = FALSE)
```

4.9.3 Max to one

`maxToOne.scale` will take the maximum value for each row and set it equal to 1. Every other cell in the row will be divided by the max.

```
#Original Matrix
mat
maxToOne.scale(mat)
mat1
maxToOne.scale(mat1)
```

Note that if the `max=0` then the whole row is set to 0. This avoids NAs.

4.9.4 Zero to one

`zeroToOne.scale` compares the differences between the max and min of each row. It uses the following formula.

```
mat
maxToOne.scale(mat)
```

There are 2 conditions where this does not apply. First is when `max=0`. In this case to avoid NAs, the row is set to 0.

```
mat1
maxToOne.scale(mat1)
```

The other condition is when the max is equal to the min. When this happens, the row is set to 1.

```
mat2
zeroToOne.scale(mat2)
```

4.9.5 metaprofile with a matrix.op

Now we can add a scaling factor into the `metaprofile.bigWig`

```
metaprofile.bigWig(bed6, bw.plus, bw.minus = bw.minus, step = 1,
                   name = "Signal", matrix.op = zeroToOne.scale,
                   profile.op = bootstrapped.confinterval.metaprofile)
```

4.10 plots.bigWig

`plots.bigWig` produces a standardized plot for a `metaprofile.bigWig` object.

```
plot.metaprofile(x, minus.profile = NULL, X0 = x$X0,
      draw.error = TRUE, col = c("red", "blue", "lightgrey", "lightgrey"),
      ylim = NULL, xlim = NULL, xlab = "Distance (bp)", ylab = x$name)
```

- arguments
  - x is meta-profile, or composite profile, instance for sense strand.
  - minus.profile is an optional meta-profile instance for the reverse strand.
  - X0 is the numeric offset in base pairs (bp) to shift (subtract) "zero" position.
  - draw.error is the ogical value indicating if profile error polygon should be drawn.
  - col is the vector of colors to use for the profiles lines and the error polygons.
  - ylim is the (y1, y2) limits of the plot.
  - xlim is the (x1, x2) limits of the plot.
  - xlab is the label for x-axis.
  - ylab is the label for y-axis.

First generate `metaprofile.bigWig` objects for the reads aligning to the coding and non-coding strands of the gene annotations and set them to variables `x=metaprofile.bigWig` and `y=metaprofile.bigWig`. Note that `y` switches the `plusPRO` and `minusPRO` inputs to calculate signal aligning to non-coding strand. Next, invoke `plot.metaprofile` to visualized the metaprofiles. The `X0` argument offsets x-axis "zero" postion of the data. The `ylim` and `xlim` arguments are the lower and upper limits of the axes, which are automatically calculated if `NULL`. `draw.error` is a logical flag that draws error regions, with defauly light grey polygons. The order of the `col` vector is sense strand profile line, reverse strand profile line, sense strand error polygon, and reverse strand error polygon.

```
x = metaprofile.bigWig(bedTSSwindow, plusPRO, bw.minus = minusPRO,
                  step = 10, name = "Signal", matrix.op = NULL,
                  profile.op = bootstrapped.confinterval.metaprofile,
                  alpha=0.05, n.samples=1000)

y = metaprofile.bigWig(bedTSSwindow, minusPRO, bw.minus = plusPRO,
                  step = 10, name = "Signal", matrix.op = NULL,
                  profile.op = bootstrapped.confinterval.metaprofile,
                  alpha=0.05, n.samples=1000)

plot.metaprofile(x, minus.profile = y, X0 = 250,
      draw.error = TRUE, col = c("red", "blue", "lightgrey", "lightgrey"),
      ylim = NULL, xlim = c(-200,200), xlab = "Distance (bp)",
      ylab = "PRO-seq signal")
```

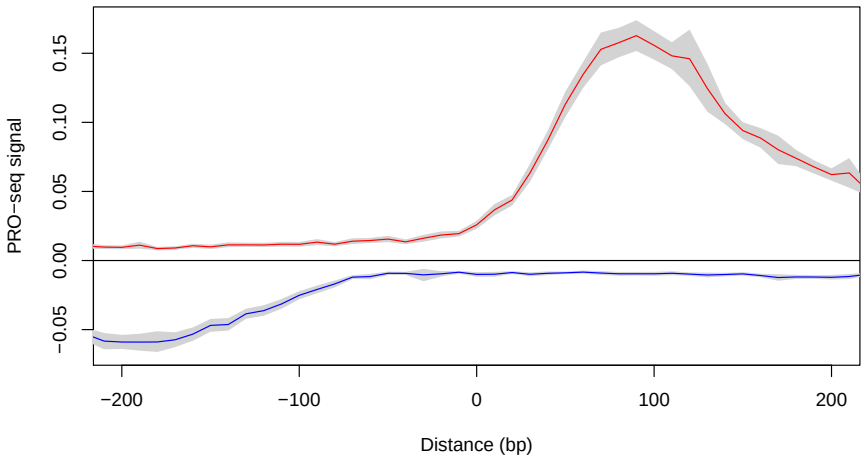

Figure 6: Composite metagene profiles of RNA polymerase density
PRO-seq signal accumulates at the RNA pausing site downstream of the TSS and there is a divergent RNA polymerse peakin the opposite orientation.