bigWig

Luther Vucic [1]***, Andréé 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

27 July 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 as an [`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: 0x6000007d2910>
##
## $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
```

The full set of attributes can be printed out on the console using `print.bigWig`. [see later in documentation]

4.1.3 unload.bigWig

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

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

4.1.4 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`

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"            "bed6"           "bed6loc"        "bedloc"
##  [5] "bw"             "bw.bp"          "bw.probes"      "bw.probes.Q"
##  [9] "bw.splitprobes" "mat2"
#remove variable in R
remove(bw)
ls()
## [1] "bed"           "bed6"          "bed6loc"       "bedloc"
## [5] "bw.bp"         "bw.probes"     "bw.probes.Q"   "bw.splitprobes"
## [9] "mat2"
```

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).

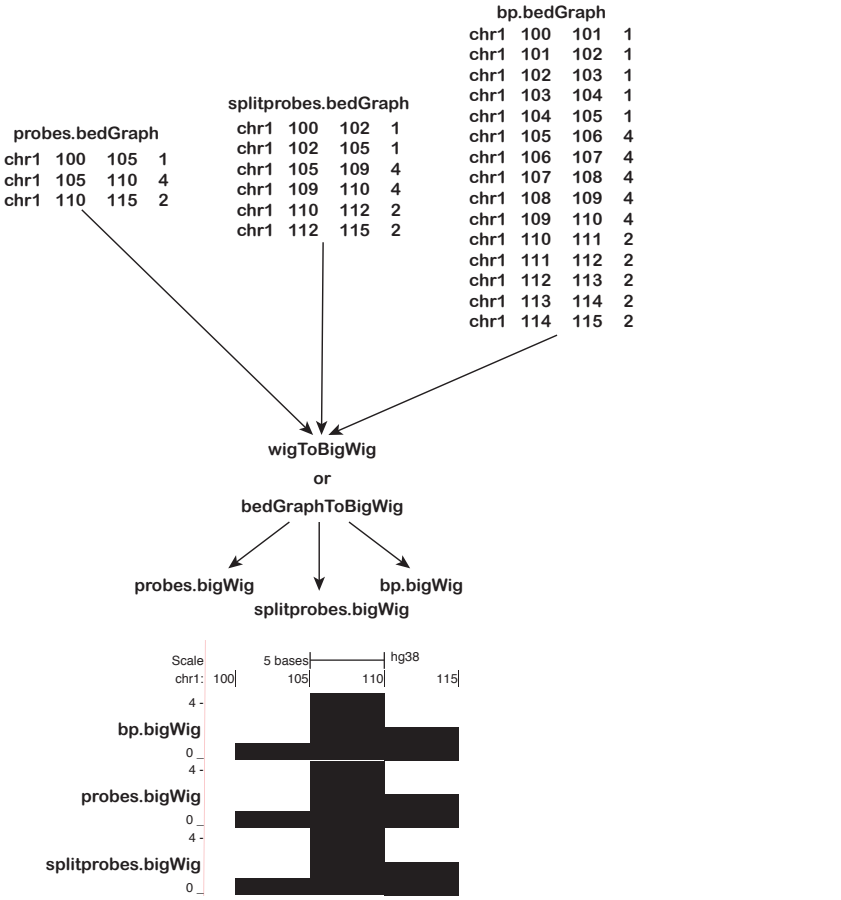

Figure 1: Structured bigWig files
Three bigWig files with identical values at each position are structured differently to later highlight the differences between *Query.bigWig functions.

```
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 referring to what chromosome is referenced
  - `start` is an integer value designation the starting position
  - `end` is an integer value designation the ending position
  - `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 map regions of the genomic interval that can't be mapped because the sequence is repeated in the genome.

4.3.1 region query

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 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 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:

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

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, resulting in 2.714286, as determined by: $((1*3) + (4*4)) / (3 + 4)$.

```
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 determined by as determined by: $((1*5) + (4*5) + (2*1)) / (5 + 5 + 1)$.

```
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 load, create and manipulate BED files.

4.4.1 BED format

A standard three column BED file is a tab delimited file that consists of the name of the chromosome, the starting, and ending point on the chromosome. 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.

```
bedloc='../inst/extdata/testBED1.bed'
bed=read.table(bedloc, 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`.

```
bed6loc='../inst/extdata/testBED1_strand.bed'
bed6=read.table(bed6loc, 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, so if they are passed a BED6 file, `threeprime.bed` and `fiveprime.bed`, upstream and downstream are relative to the strand information.

```
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 bed file, we can test a few different scenarios.

Row 1 is an example of a difference that is even. Row 2 is an example of a difference is odd and less than 1. Row 3 is an example of a difference is odd and greater than 1.

```
bed
```

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

```
center.bed(bed, upstreamWindow = 0, downstreamWindow = 0)
```

From this you can see that Row 1 has an anchor point of 305, because the difference of `start` and `end` is 10. Divide by 2 and added to the original `start` of 300 gives us 305.

Row 2 has a anchor point of 310 because a difference of 1. Divided by 2 results in 0.5. Since it was a odd difference we round down to 0. 0 + the original `start` is the original `start`.

Row 3 has an anchor point of 411. This is because half of the difference is 1.5, which is rounded down to 1 and added to the original `start`.

Note that all of the new `end` values are the anchor point plus 1.

Now take a look at a few situations where the `upstreamWindow` and `downstreamWindow` are not 0.

Windows Equal and Positive

Here is an example when they are equal and positive.

```
center.bed(bed, upstreamWindow = 5, downstreamWindow = 5)
```

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

Windows Unequal and Negative Now let's try a negative value.

```
center.bed(bed, upstreamWindow = -1, downstreamWindow = 4)
```

Notice that the `start` value is actually the anchor point + 1. This is due to the subtracting a negative is the same as adding the positive value. If you due use negative values be aware of the possibility that your `start` can be larger than your `end`, which will cause errors with other bigWig functions.

5 Prime and 3 Prime

By setting `upstreamWindow = 0` and `downstreamWindow = 0`, you can see that the difference between `start` and `end` have no influence on the anchor point, but rather the function `fiveprime.bed` and `threeprime.bed` does.

```
fiveprime.bed(bed, upstreamWindow = 0, downstreamWindow = 0)
threeprime.bed(bed, upstreamWindow = 0, downstreamWindow = 0)
```

`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)
threeprime.bed(bed, upstreamWindow = 1, downstreamWindow = 5)

# negative value
fiveprime.bed(bed, upstreamWindow = -1, downstreamWindow = 5)
threeprime.bed(bed, upstreamWindow = -1, downstreamWindow = 5)
```

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. However, if you pass a BED6 file it will align with the strand.

```
fiveprime.bed(bed6, upstreamWindow=4, downstreamWindow=2)
threeprime.bed(bed6, upstreamWindow=4, downstreamWindow=2)
```

See that when you change the `strand`, it changes the anchor point from which the window is calculated.

- 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`

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(bed, downstreamWindow = 5)
upstream.bed(bed, upstreamWindow = 5)
```

Note that negative numbers for `downstreamWindow` and `upstreamWindow` will return a BED, but it will cause errors when used in other bigWig functions.

If you use a BED6 file, it follows the `strand` alignment.

```
downstream.bed(bed6,5)
upstream.bed(bed6,5)
```

4.4.5 foreach

`foreach.bed` is a way to quickly apply a function across all rows of a bed file.

```
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 (usually '+' 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)
```

Everything is wrapped into a function `sizes.bed`. This can be anything you want, just be sure it's descriptive of what it does.

`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`. Note that the function is written within the `foreach.bed` but could be written outside and called by the variable.

```
func <- function(i, chrom, start, end, strand) {
  sizes[i] <<- end - start
}

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

  foreach.bed(bed, func)

  return(sizes)
}
sizes.bed(bed)
```

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

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

This can be scaled up to being as complex as needed.

4.4.6 Region by Bed

```
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`
  - `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
    - `wavg` weighted average of the counts only pertains to probeQuery
  - `abs.value` is a logical argument which determines if the absolute value of the input is performed before the `op`.

This function is similar to `region.bpQuery.bigWig` except that when defining the areas we want to examine is defined in a bed file rather than `chrom`, `start`, and `end`.

The source of the bed file can be something created by hand or previous identified regions from other experiments. The basics of the bed is that it's in a R data frame.

```
bed=data.frame('chr1',10496,10497)
#set column headers
colnames(bed)=c('chrom','start', 'end')
```

Now this is for a single factor in R. When creating a dataframe in R, it automatically turns strings into factors. This limits the ability to add different `chrom` designations. Meaning that when created the original bed file, `chr1` was the only level created. It will return an error if you just try to add

```
rbind(bed, c('chr2', 10000, 20000))
```

If you ever want to add different factors, you'll need to use `levels()`

```
levels(bed$chrom)=c('chr1', 'chr2')
```

Take a look at how the data.frame is structured

```
dim(bed)
attributes(bed)
bed
```

`dim` returns the size of the matrix [1 row, 3 columns]. while `attributes` returns information on column names, row names and class type.

You can take this bed file and run it through the bigWig file to see what regions overlap

```
# note: If you leave out op='', it will default to op='sum'
bed.region.bpQuery.bigWig(bw, bed)
```

Now adding a few other regions to the data frame

```
bed=rbind(bed, c('chr2', 10500,10501))
```

In the original query, this region is occupied by a `chr1` and since the bed file refers to a `chr2` the sum should be the same because there is no overlap.Then if you rerun

```
bed.region.bpQuery.bigWig(bw, bed)
```

We see that the then returned values are 1 and 0. This is because the first region of the bed file overlaps regions of the bigWig, but the second bed region does not overlap any regions of the bigWig.

Now adding a third row to the bed file that will overlap a larger range of the bigWig and rerun

```
bed2=rbind(bed, c('chr1', 13000,14001))
bed.region.bpQuery.bigWig(bw, bed2)
```

The returned values are the sums of the counts in those regions.

4.4.7 bed.region with gap.value

As shown in the gap.value section above, we can build queries where there are counts and where there are no counts.

First, the query with a count is

```
query.bigWig(bw, chrom='chr2',start=229990, end=229992)
```

Next, a query without any counts

```
query.bigWig(bw, chrom='chr2',start=229993, end=230001)
```

From these 2 queries, we can build a bed file

```
bedWgap =data.frame('chr2', 229990, 229992)
bedWgap=rbind(bedWgap,c('chr2', 229993, 230001))
colnames(bedWgap)=c('chrom', 'start', 'end')
```

Finally, we can run a bed.region function with a `gap.value=270` and see the results.

```
bed.region.bpQuery.bigWig(bw, bedWgap, op='avg', gap.value=270)
```

Be aware that bed.region.bpQuery defaults to `gap.value=0`, while bed.region.probeQuery defaults to `gap.value=NA`. Both 0 and NA can be substituted in each version as shown below.

```
bed.region.bpQuery.bigWig(bw, bedWgap, op='avg', gap.value=NA)
bed.region.bpQuery.bigWig(bw, bedWgap, op='avg', gap.value=0)
bed.region.probeQuery.bigWig(bw, bedWgap, op='avg', gap.value=NA)
bed.region.probeQuery.bigWig(bw, bedWgap, op='avg', gap.value=0)
```

The NA and 0 versions accomplish the same thing as denoting that there was no data returned. The distinction comes further down the line of the analysis when you filter out NULL values by searching for 0 or NA. With that being said, when you set gap.value to anything but 0 or NA, there is no way to distinguish if the value is a null.

4.5 Step

The following functions operate over defined steps and is described by `step=` argument. This means in a given region [`start=1` and `end=10`] and a `step=5`, the function will create subregions of 5. In this example, it will run on [`start=1`, `end=5`] and [`start=6`, `end=10`]. Again, `probeQuery` and `bpQuery` functions are the same, exce;pt when calculating `op=avg`.

4.5.1 Step through region

```
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 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.

The Step function will run through the range provide breaking it up into equal size steps as defined by `step =`. The key here is that the length of the range [`end-start`] has to be a multiple of the step. For example if `end=21` and `start=1`, 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 =1` and `op = 'min'`, then the return would be 20 minimums.

Now if `step = 5` and `op = 'max'`, the return will be a 4 element array of the maximum value in the step.

Let's take a look over a 20000 interval `start=1`, `end=20001` and a `step=1000`.

```
step.bpQuery.bigWig(bw,chrom='chr1',start=1, end=20001, op='sum', step=1000)
```

The result is a 20 element array of the sum of all the counts in the interval. Notice that the steps that have no counts are zero. If we needed to fill these values in with a specific number like 10, we use `gap.value=10`

```
#gap.value=0
step.bpQuery.bigWig(bw,chrom='chr1',start=1, end=20001, op='sum', step=10000,
                    gap.value=0, with.attributes=FALSE)

#gap.value=10
step.bpQuery.bigWig(bw,chrom='chr1',start=1, end=20001, op='sum', step=10000,
                    gap.value=10, with.attributes=FALSE)
```

4.5.2 Step through region by Bed

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

bed.step.probeQuery.bigWig(bw, bed, step,
                    op = "wavg", abs.value = FALSE, gap.value = NA,
                    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`
  - `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.

This is similar to `bed.region.bigWig()`, where you can add a bed of regions that you are interested in.

```
#Create bed dataframe
bed3 = data.frame('chr1', 15000, 25000)
colnames(bed3)=c('chrom', 'start', 'end')
bed3=rbind(bed3, c("chr1", 30000, 35000))
bed.step.bpQuery.bigWig(bw, bed3, step=1000, op='avg', with.attributes=FALSE)
```

Notice that the defined regions in the bed file are exact multiples of the step. This is explained in the `bed.bpQuery.bigWig` example. The other attribute of this bed file is the regions defined do not need to be the same size. row 1 in the bed files contains 10 steps, while Row 2 has 5 steps. the final aspect of this example is that `bpQuery` version uses the `step` size as the denominator in the average. While `probeQuery` will use the number of rows in the query

```
bed.step.probeQuery.bigWig(bw, bed3, step=1000, op='avg', with.attributes=FALSE)
```

In the `probe` version, we end up with where there are no overlapping regions. This is because dividing by zero is not possible. Instead the function returns a `NA`.

4.5.3 Bed6 files

`bed.region.bpQuery.bigWig()` and `bed.step.bpQuery.bigWig()` have counterparts that can take a bed6 file. The bed6 file is similar to a bed file except it has 3 more columns of data.

Remember the standard bed file has `chrom`, `start` and `end`. The bed6 adds `name`, `score`, `strand` columns to its structure. For these functions, we only need the added `strand` column. However this column needs to be in the 6th position. Meaning even though `name` and `score` columns exist in the dataframe, they can be populated with nulls. You could populate it with identifying information, but the function essentially ignores them. The `strand` column requires either a `+` or `-` to denote the plus or minus strand.

Here is an example

```
bed6=data.frame('chr1',1,100000,'','','+')
colnames(bed6)=c('chrom', 'start', 'end', 'name', 'score', 'strand')
```

This introduces the biological concept of plus and minus strands. Because DNA is double stranded and the strands are antiparallel to one another, a particular read will map to only a single strand. This is useful for stranded xxx-seq protocols, such as PRO-seq.

```
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, step,
                          op = "wavg", abs.value = FALSE, gap.value = NA,
                          with.attributes = TRUE, as.matrix = FALSE,
                          follow.strand = FALSE)
```

4.5.3.1 bed6.region

- 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.

Let's look at an example. We will use data from the negative values used with `abs.value = TRUE` In the Region section.

```
dtDir = '/home/directory'
dtFnPlus='GSM3452725_K562_Nuc_NoRNase_plus.bw'
dtFnMinus='GSM3452725_K562_Nuc_NoRNase_minus.bw'
bw.plus=load.bigWig(paste0(dtDirNeg, dtFnPlus))
bw.minus=load.bigWig(paste0(dtDirNeg, dtFnMinus))
```

Using the `bw.plus` and `bw.minus` strands, we can evaluate a `bed6.region` function. First, take a look at the query for each strand.

```
query.bigWig(bw.minus, chrom='chr1', start=25000, end=50000)
query.bigWig(bw.plus, chrom='chr1', start=25000, end=50000)
```

These will be used as reference for when we use the function.

```
bed6=data.frame('chr1',25000,50000,'','','+')
colnames(bed6)=c('chrom', 'start', 'end', 'name', 'score', 'strand')
```

This particular bed file defines a region between `start = 25000` and `end = 50000` on the `+` strand.

```
bed6.region.probeQuery.bigWig(bw.plus, bw.minus,
                  bed6, op='wavg', abs.value = FALSE, gap.value=0)
```

The query of the plus strand shows only one overlapping region. The average of 1 region with 1 count is 1.

Now add another row to our `bed6` file and rerun the previous `bed6.region.probeQuery.bigWig` function.

```
levels(bed6$strand)=c('+', '-')
bed6=rbind(bed6, c('chr1', 25000, 50000, '', '', '-'))
bed6.region.probeQuery.bigWig(bw.plus, bw.minus, bed6, op='sum', abs.value = FALSE, gap.value=0)
```

Similarly to the `bed.region` function the return is 2 values one for each overlapping region.

We can invoke `ab.value = TRUE` argument and our second result change to a positive value.

```
bed6.region.probeQuery.bigWig(bw.plus, bw.minus, bed6,
                  op='sum', abs.value = TRUE, gap.value=0)
```

```
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)
```

4.5.3.2 bed6.step

- 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.

This is just like `step.bed.xxx` functions.

```
bed6.step.bpQuery.bigWig(bw.plus, bw.minus, bed6, step=5000,
                         op = "sum", abs.value = FALSE, gap.value = 0,
                         bwMap = NULL, with.attributes = FALSE, as.matrix = FALSE,
                         follow.strand = FALSE)
```

4.5.3.3 as.matrix

Here the attribute `as.matrix` will be introduced. This attribute causes the output to be a matrix.

```
bed6=data.frame('chr1', 1, 100001, 'a', 'c', '+')
colnames(bed6)=c('chrom', 'start', 'end', 'name', 'score', 'strand')
bed6.step.bpQuery.bigWig(bw.plus, bw.minus, bed6, step=5000,
                         op = "sum", abs.value = FALSE, gap.value = 0,
                         bwMap = NULL, with.attributes = TRUE, as.matrix = TRUE,
                         follow.strand = FALSE)
```

4.5.3.4 follow.strand

`follow.strand` is an attribute that will switch the direction of how it reads the `-` strand. This allows you to read both strands `+` and `-` from the 3' end. This attribute is commonly set to TRUE when the specific genomic feature in the bed file has inherent strandedness. For example, a sequence motif or transcription start site. It is useful to know how the counts relate to the orientation of the bed file feature.To show this we can see that the results are mirrors of each other.

```
#follow.strand = FALSE
bed6=data.frame('chr1', 1, 100001, 'a', 'c', '-')
colnames(bed6)=c('chrom', 'start', 'end', 'name', 'score', 'strand')
bed6.step.bpQuery.bigWig(bw.plus, bw.minus, bed6, step =5000,
                         op = "sum", abs.value = FALSE, gap.value = 0,
                         bwMap = NULL, with.attributes = FALSE, as.matrix = TRUE,
                         follow.strand = FALSE)
```

Mirrors

```
#follow.strand = TRUE
bed6.step.bpQuery.bigWig(bw.plus, bw.minus, bed6, step =5000,
                         op = "sum", abs.value = FALSE, gap.value = 0,
                         bwMap = NULL, with.attributes = FALSE, as.matrix = TRUE,
                         follow.strand = TRUE)
```

4.6 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 viable `mat` are: * `bed.step.bpQuery.bigWig` * `bed.step.probeQuery.bigWig` * `bed6.step.bpQuery.bigWig` * `bed6.step.probeQuery.bigWig`

All of these functions require the `as.matrix=TRUE` attribute.

4.6.1 Quantiles

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

For this example, we'll create a simple bed file and run it through `bed6.step.bpQuery.bigWig` with `as.matrix=TRUE` to get a `mat`.

```
bed6=data.frame('chr1', 1, 100001, 'a', 'c', '+')
colnames(bed6)=c('chrom', 'start', 'end', 'name', 'score', 'strand')
bed6=rbind(bed6, c('chr1', 200001, 300001, 'a', 'c', '+'))
bed6=transform(bed6, start=as.numeric(start), end=as.numeric(end))
bed6
mat=bed6.step.bpQuery.bigWig(bw.plus, bw.minus, bed6, step=50000,
                          op = "sum", abs.value = FALSE, gap.value = 0,
                          bwMap = NULL, with.attributes = TRUE, as.matrix = TRUE,
                          follow.strand = FALSE)
mat
```

We can then pass this `mat` to `quantiles.metaprofile`

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

The result of `quantiles.metaprofile` is a list of `quantile` values for the number and step size. The above example returns 2 values per quantile value because there are 2 steps in the given window.

4.6.2 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(mat, quantiles = c(0.875, 0.5, 0.125), fraction = 0.90,
                                 n.samples = 5000)
```

4.6.3 Confidence Interval

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

```
confinterval.metaprofile(mat, 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

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

4.6.4 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.[1]

[1] https://cran.r-project.org/web/packages/dabestr/vignettes/bootstrap-confidence-intervals.htmllink

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

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

4.6.5 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.

```
metaprofile.bigWig(bed6, bw.plus, bw.minus = bw.minus, step = 50000, 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(bed6, bw.plus, bw.minus = bw.minus, step = 50000, name = "Signal",
                   matrix.op = NULL, profile.op = bootstrapped.confinterval.metaprofile, alpha=0.05, n.samples
```

4.7 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

4.7.1 RPKM

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

```
# Original mat
mat

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

4.7.2 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)
```

4.7.3 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.

4.7.4 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)
```

4.7.5 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 = 50000, name = "Signal",
                   matrix.op = zeroToOne.scale,
                   profile.op = bootstrapped.confinterval.metaprofile)
```

4.8 bwMap

4.9 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 instance for sense strand
  - minus.profile is Optional meta-profile instance for reverse strand
  - X0 is Numeric offset in base pairs (bp) to shift (subtract) "zero" position.
  - draw.error is Logical value indicating if profile error polygon should be drawn.
  - col is Vector of colors to use for, respectively, sense strand profile line, reverse strand profile line, sense strand error polygon, reverse strand error polygon.
  - ylim is The (y1, y2) limits of the plot.
  - xlim is The (x1, x2) limits of the plot.
  - xlab is Label for x-axis.
  - ylab is Label for y-axis.

In order for this plot function to work, you need a `metaprofile.bigWig` object. You get this by setting it to a variable `x=metaprofile.bigWig`. Then you can call `plot.metaprofile`.

```
x=metaprofile.bigWig(bed6, bw.plus, bw.minus = bw.minus, step = 50000, name = "Signal",
                  matrix.op = NULL, profile.op = confinterval.metaprofile)

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)
```

4.9.1 Reverse strand

To add the reverse strand, you need a `metaprofile.bigWig` for it. Here we set `xr` to the reverse strand.

```
x=metaprofile.bigWig(bed6, bw.plus, bw.minus = bw.minus, step = 50000, name = "Signal",
                  matrix.op = NULL, profile.op = confinterval.metaprofile)
xr=metaprofile.bigWig(bed6, bw.minus, bw.minus = bw.plus, step = 50000, name = "Signal",
                  matrix.op = NULL, profile.op = confinterval.metaprofile)
plot.metaprofile(x, minus.profile = xr, X0 = x$X0,
       draw.error = TRUE, col = c("red", "blue", "lightgrey", "lightgrey"),
       ylim = NULL, xlim = NULL, xlab = "Distance (bp)", ylab = x$name)
```

Note that `xr` switches the `bw.plus` and `bw.plus` inputs. This gives our reverse strand because from the original data they are the reverse of each other.

4.9.2 Offset start

`X0` input allows you to change the "zero" point of the data. By default it uses the `X0=x$X0`, but it could be changed manually too.

```
plot.metaprofile(x, minus.profile = xr, X0 = 25000,
      draw.error = TRUE, col = c("red", "blue", "lightgrey", "lightgrey"),
      ylim = NULL, xlim = NULL, xlab = "Distance (bp)", ylab = x$name)
```

Notice the x axis distance changes. This is because you are setting the start to be offset by 25000.

4.9.3 Axes limits

`ylim` and `xlim` are the lower and upper limits of the axes and are automatically calculated if `NULL`. However, you can choose your own values by passing a vector [low, high].

```
plot.metaprofile(x, minus.profile = xr, X0 = 25000,
      draw.error = TRUE, col = c("red", "blue", "lightgrey", "lightgrey"),
      ylim = c(-500,500), xlim = c(-1000, 50000), xlab = "Distance (bp)", ylab = x$name)
```

Looking at the axes, you can ow see the change in lower and upper limits.

4.9.4 Error regions

`draw.error` is a logical flag that turns on [`draw.error=TRUE`] error regions [light grey areas] or off [`draw.error=FALSE`]

```
plot.metaprofile(x, minus.profile = xr, X0 = 25000,
                 draw.error = FALSE, col = c("red", "blue", "lightgrey", "lightgrey"),
                 ylim = NULL, xlim = NULL, xlab = "Distance (bp)", ylab = x$name)
```

You can see that the light grey error regions have been removed.

4.9.5 Colors

`col` input is a vector of predefined colors for plot. `R` has hundreds of predefined colors. To obtain a list, you can call `colors()`. Here is the 1st 25 colors.

```
colors()[1:25]
```

The order of the `col` vector is sense strand profile line, reverse strand profile line, sense strand error polygon, reverse strand error polygon. You can change the colors to help clarify each region.

```
plot.metaprofile(x, minus.profile = xr, X0 = 25000,
                 draw.error = TRUE, col = c("green", "yellow", "purple", "grey"),
                 ylim = NULL, xlim = NULL, xlab = "Distance (bp)", ylab = x$name)
```