bigWig

Luther Vucic

09/15/2019

Contents

Introduction

bigWig is a R package that has utility function for analyzing and manipulating bigWig files.

Setup

Install package

Since bigWig is not available on CRAN, we can not use the basic `install.packages('bigWig')`. However, there are several ways to install packages. Here I will explain 2 of them.

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)
```

From local directory

If you don't have an internet connection or don't use github, you can build from the sources file.

```
#install devtools if necessary
install.packages("devtools")
library('devtools')
#Set the working directory to the directory where the source files are located
setwd('~/Dir')
build()
```

Usage

After installation like any other package bigwig needs to be loaded with

```
library(bigWig)
```

For these examples, we will use PRO-seq data from GSE126919_RAW.

Download the tar ball and unpack it in a local directory of your choice. Then in R make sure that you define the directory and filename.

```
#directory where data is stored
dtDir='/home/directory'
# specific bigWig file being used
dtFn='GSM3618124_HEK293T_TIR1_C14_3hrDMSO_rep1_minus_body_0-mer.bigWig'
```

bigWig utilities

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

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. Theseare destroyed when you unload the bigWig. If left as the default `udcDir = NULL`, then it uses `/tmp/udcCache`.

`load.bigWig` creates a list in `R`. This list contains relevant informasyion 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.

```
bw=load.bigWig(paste0(dtDir, dtFn))
```

unload.bigWig

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

- arguements
  - `bw` is the `R` pointer created in `load.bigWig`

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

```
unload.bigWig(bw)
ls()
#> [1] "bw"    "dtDir" "dtFn"
remove(bw)
ls()
#> [1] "dtDir" "dtFn"
```

query.bigWig

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

- arguments
  - `bw` is the `R` pointer 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.

`query.bigWig` allows you to search the bigWig files for specific chromosomes (`chrom='chr1'`, a string representative of the desired chromosome within a defined window (`start=1, end = 12000` both are integers and end is inclusive meaning it searches up to and including `end`). It then prints query results to the command line.

```
query.bigWig(bw, chrom='chr1', start=1, end=12000)
#>       start   end value
#>  [1,] 10496 10497     1
#>  [2,] 10500 10501     1
#>  [3,] 10518 10519     1
#>  [4,] 10521 10522     2
#>  [5,] 10527 10528     1
#>  [6,] 10535 10536     1
#>  [7,] 10541 10542     1
#>  [8,] 10554 10555     1
#>  [9,] 10933 10934     1
#> [10,] 11081 11082     2
#> [11,] 11165 11166     1
#> [12,] 11456 11457     1
#> [13,] 11584 11585     1
```

You can set the query to a variable for storage

```
bwQ=query.bigWig(bw, chrom='chr1', start=1, end=20000)
bwQ[3]
#> [1] 10518
```

Then access the array like any other indexed array. This returns the entire row.

```
bwQ[1,]
#> start   end value
#> 10496 10497     1
```

This returns the specific row and column.

```
bwQ[1,2]
#>   end
#> 10497
```

It can be accesed by keyword too.

```
bwQ[1,'start']
#> start
#> 10496
```

print.bigWig

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

```
print.bigWig(bw)
```

- arguments
  - `bw` is the `R` pointer created in `load.bigWig`

```
#> bigWig
#>  version: 4
#>  isCompressed TRUE
#>  isSwapped FALSE
#>  primaryDataSize: 11,243,315
#>  primaryIndexSize: 176,800
#>  zoomLevels: 7
#>  chromCount: 455
#>      chr1 248956422
#>      chr10 133797422
#>      chr10_GL383545v1_alt 179254
#>      chr10_GL383546v1_alt 309802
#>      chr10_KI270824v1_alt 181496
#>      ...
#>      chrX 156040895
#>      chrX_KI270880v1_alt 284869
#>      chrX_KI270881v1_alt 144206
#>      chrX_KI270913v1_alt 274009
#>      chrY 57227415
#>      chrY_KI270740v1_random 37240
#>  basesCovered: 5,010,318
#>  mean:  1.187913
#>  min:  1
#>  max:  3034
#>  std:  5.843396
```

Region

The following sections group `bpQuery` and `probeQuery` functions together because they operate the same except on how they calculate the average.

By region

This set of 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,
                      op = "sum", abs.value = FALSE
                     bwMap = NULL)
region.probeQuery.bigWig(bw, chrom, start, end,
                     op = "wavg", abs.value = FALSE, gap.value = NA)
```

- arguments
  - `bw` is the `R` pointer 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`.
  - `bwMap` is a bigWig file of areas that cannot be mapped for a reason

This allows you to find out basic information on a specific query. Starting with a specific query,

```
query.bigWig(bw, chrom='chr2', start=229990, end=230235)
#>       start    end value
#> [1,] 229991 229992     1
#> [2,] 230002 230003     2
#> [3,] 230077 230078     1
#> [4,] 230082 230083     1
#> [5,] 230113 230114     1
#> [6,] 230132 230133     1
#> [7,] 230146 230147     2
```

Operations

Sum op='sum'

To find how many instances there are where there is a 'chr2' in the bigWig

```
region.bpQuery.bigWig(bw,chrom='chr2',start=229990, end=230235, op='sum')
#> [1] 9
region.probeQuery.bigWig(bw,chrom='chr2',start=229990, end=230235, op='sum')
#> [1] 9
```

Maximum op='max'

If you want to find the highest number of instances of chr2

```
region.bpQuery.bigWig(bw,chrom='chr2',start=229990, end=230235, op='max')
#> [1] 2
region.probeQuery.bigWig(bw,chrom='chr2',start=229990, end=230235, op='max')
#> [1] 2
```

Minimum op='min'

To find the lowest number of instances

```
region.bpQuery.bigWig(bw,chrom='chr2',start=229990, end=230235, op='min')
#> [1] 1
region.probeQuery.bigWig(bw,chrom='chr2',start=229990, end=230235, op='min')
#> [1] 1
```

Average op='avg'

To find the average

```
region.bpQuery.bigWig(bw,chrom='chr2',start=229990, end=230235, op='avg')
#> [1] 0.03673469
region.probeQuery.bigWig(bw,chrom='chr2',start=229990, end=230235, op='avg')
#> [1] 1.285714
```

Notice the difference in the return of the average. This is because `bpQuery` counts the number of base pairs to use as the denominator of the average. This is the difference of the `end` value and the `start` value.

```
230235-229990
#> [1] 245
```

`probeQuery` counts the number of probes. Essentially, this is the number of rows returned by the query. In this example, it is 7.

abs.value = FALSE

Sometimes, the bigWig will have negative values. To keep these values in the counts the `abs.value=TRUE` option can be used. For this example, you'll need a different data set. negative bigWig files. Download both

- GSM3452725_K562_Nuc_NoRNase_minus.bw
- GSM3452725_K562_Nuc_NoRNase_plus.bw

and store them in thier own directory. Then using the `load.bigWig` and store them as `bw.plus` and `bw.minus`, respectively. We will use them in later examples.

When we run a query on `bw.minus`, you can see that it returns negative counts. You can check out the appendix to see how to search and find negative values.

```
query.bigWig(bw.minus, chrom='chr1', start=10140, end=10190)
#>      start   end value
#> [1,] 10151 10153    -1
#> [2,] 10153 10154    -2
#> [3,] 10154 10155    -1
#> [4,] 10158 10160    -1
```

There is a reason the bigWig file returns negative values. We only care about the case that there is a recorded event [+/-]. In this case, we apply the `abs.value=TRUE` which takes the absolute value of each count before applying the operation. Remember the default is `abs.value=FALSE`

```
region.probeQuery.bigWig(bw.minus,chrom='chr1',start=10140, end=10190, op='avg')
#> [1] -1.25
region.probeQuery.bigWig(bw.minus,chrom='chr1',start=10140, end=10190, op='avg', abs.value=TRUE)
#> [1] 1.25
```

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 `R` pointer 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`.

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))
#> Warning in `[<-.factor`(`*tmp*`, ri, value = "chr2"): invalid factor level,
#> NA generated
#>   chrom start   end
#> 1  chr1 10496 10497
#> 2  <NA> 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)
#> [1] 1 3
attributes(bed)
#> $names
#> [1] "chrom" "start" "end"
#>
#> $row.names
#> [1] 1
#>
#> $class
#> [1] "data.frame"
bed
#>   chrom start   end
#> 1  chr1 10496 10497
```

`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)
#> [1] 1
```

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)
#> [1] 1 0
```

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)
#> [1]  1  0 11
```

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

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`.

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 `R` pointer 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)
#>  [1]  0  0  0  0  0  0  0  0  0  0 10  5 11 11  1  1  2  0  4  3
#> attr(,"chrom")
#> [1] "chr1"
#> attr(,"start")
#> [1] 1
#> attr(,"end")
#> [1] 20001
#> attr(,"step")
#> [1] 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)
#> [1]  0 48

#gap.value=10
step.bpQuery.bigWig(bw,chrom='chr1',start=1, end=20001, op='sum', step=10000,
                    gap.value=10, with.attributes=FALSE)
#> [1] 10 48
```

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 `R` pointer 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)
#> [[1]]
#>  [1] 0.001 0.002 0.000 0.004 0.003 0.002 0.007 0.010 0.003 0.004
#>
#> [[2]]
#> [1] 0 0 0 0 0
```

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)
#> [[1]]
#>  [1] 1.000000 1.000000       NA 1.000000 1.000000 1.000000 1.000000
#>  [8] 1.111111 1.000000 1.000000
#>
#> [[2]]
#> [1] NA NA NA NA NA
```

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`.

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. This refers to the situation where the biological experiment returns the sense [plus strand] and it's complimentary RNA strand, antisense [minus strand]

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, 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.

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)
#>      start   end value
#> [1,] 28567 28568    -1
#> [2,] 28570 28571    -1
#> [3,] 46605 46606    -1
#> [4,] 47071 47072    -1
#> [5,] 49218 49219    -1
query.bigWig(bw.plus, chrom='chr1', start=25000, end=50000)
#>      start   end value
#> [1,] 29459 29460     1
```

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)
#> [1] 1
```

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)
#> [1]  1 -5
```

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)
#> [1] 1 5
```

bed6.step

```
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.

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)
#> [[1]]
#> [1] 1 0 0 0 0
#>
#> [[2]]
#> [1] -2  0  0  0 -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)
#>      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
#> [1,]    0    0    0    0    0    1    0    0    0     0     0     0     1
#>      [,14] [,15] [,16] [,17] [,18] [,19] [,20]
#> [1,]     0     0     1     1     0     0     0
#> attr(,"step")
#> [1] 5000
```

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. 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)
#>      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
#> [1,]    0    0 -129 -215   -1   -2    0    0    0    -3   -12   -32   -67
#>      [,14] [,15] [,16] [,17] [,18] [,19] [,20]
#> [1,]   -32   -21   -14   -19  -115   -29   -12
```

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)
#>      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
#> [1,]  -12  -29 -115  -19  -14  -21  -32  -67  -32   -12    -3     0     0
#>      [,14] [,15] [,16] [,17] [,18] [,19] [,20]
#> [1,]     0    -2    -1  -215  -129     0     0
```