bigWig

Luther Vucic

09/15/2019

Contents

Introduction

bigWig package is written to analyze bigWig files. Functions in this package are organized around a genomic interval. They provide several variations on counting either by region or step wise. Statistical profiles are available as well scaling functions in case the data isn't normalized before analyzing. The functions can incorporate a mappability file, which determines areas of the genome that are not mappable and exculdes them from further calculations. Finally, there a few graphing functions to display data. To understand this package a few definitions need to be established.

- **Genomic interval** is the basic unit that all of these functions and is a segment of genetic code. 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 raw counts within the genomic interval
- **Base pair** or bp is an individually indexed nucleic acid
- **Probe** refers to instances where a count exists for a bp
- **Region** contains 1 or more [BED file] genomic intervals that 1 of 4 operations has been applied to.
- **Bed** is a file that contains multiple genomic intervals which the operation will be performed on each. See UCSC's description of BED file formast. 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.

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.

bigWig format

bigWig files are genetic sequence fragments stored as indexed binary format. This means it is not readily readable by humans. Rather it is stored in binary. This allows for large continuous data to be stored compactly and quickly accessed.

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

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: 0x561a8592b310>
#>
#> $names
#>  [1] "version"          "isCompressed"     "isSwapped"        "primaryDataSize"
#>  [5] "primaryIndexSize" "zoomLevels"       "chroms"           "chromSizes"
#>  [9] "basesCovered"     "mean"             "min"              "max"
#> [13] "std"
```

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

#access individual attribute
bw$basesCovered
#> [1] 5010318
```

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

unload.bigWig

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

- arguements
  - `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` obmect. To do that use `rm()` or `remove()`

```
#destroy C object
unload.bigWig(bw)
ls()
#> [1] "bw"    "dtDir" "dtFn"
#remove variable in R
remove(bw)
ls()
#> [1] "dtDir" "dtFn"
```

query.bigWig

```
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.

`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 pointer of the underlying `C` object 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
```

BED utilities

These functions are used to load, create and manipulate BED files.

BED format

BED files come in a few variations. The are meant to store information about genetic sequences by indexing them by several data points. There are several associated data elements. More information can be found on UCSC website. For this documentation, we are concerned with 2 variations. The base version, which we refer to as a BED file, is a tab delimited file that consists of the name of the chromosome, the starting and ending point on the chromosome. These 3 points can give precise areas that a researcher wants to analyze. A BED6 file contains all of the BED columns plus 3 more. These pieces of data are name, score and strand. Name and score are not particular of interest, but strand is. Strand defines whether the BED track is looking at the + or - stand of DNA.

BED files are saved with a .bed extension. It properly identifies the type of file. The internal structure is more important. The file could have a header, which is identified by the keyword "track". The track gives information on where the data came from or how it was prepped.

Load BED file

There are several ways to read a bed file into R. One way to load a BED file is to use R's `read.table` function. This will allow you to transform the tab delimited file into a R data.frame.

To load a specific first set file location to a variable like `floc`. Then you can use the following code.

```
bed=read.table(floc, header=FALSE, sep='\t', stringsAsFactors=FALSE)
```

`header` refers to if column names are in the first line. BED files don't usually include headers so we can set `header=FALSE`. This assumes that there is no track or miscellaneous information lines. If there are extra lines before table starts, `read.table` will produce an error. To solve this use the `skip=` keyword to skip the number of lines that are on the beginning of the file.

```
bed=read.table(floc, header=FALSE, sep='\t', stringsAsFactors=FALSE, skip=1)
```

Create BED file

A BED file can be created by creating a data frame object in R and then using `write.table` to save it.

You can start by defining arrays for each column that you want included. Here is an example of a basic BED file with 3 columns. Note that we use `stringsAsFactors=FALSE` because if we don't it will make an extra step later when adding a different chromosome. Now you can add as many rows as needed, just make sure each array is the same length.

```
chrom=c('chr1')
start=c(300)
end=c(310)
bedT=data.frame(chrom, start, end, stringsAsFactors=FALSE)
bedT
#>   chrom start end
#> 1  chr1   300 310
```

Now to add a single line to the data frame, use the `rbind` function.

```
bedT=rbind(bedT,c('chr1',400,402))
bedT=transform(bedT, start=as.numeric(start), end=as.numeric(end))
bedT
#>   chrom start end
#> 1  chr1   300 310
#> 2  chr1   400 402
```

The reason we use `transform` is that R for some reason converts everything into a string when adding rows with `rbind`. What this does will give an error in the bigWig functions because they require numerical [integers or floating point] to make the calculations.

This process can be repeated for every row you want to add.

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`.

```
chrom=c('chr1', 'chr1','chr1', 'chr1')
start=c(300,400,500,600)
end=c(310,410,510,610)
name=c('na','na','na','na')
score=c(1,1,1,1)
strand=c('+','-','+','-')
bed6=data.frame(chrom,start,end,name,score,strand,stringsAsFactors=FALSE)
bed6
#>   chrom start end name score strand
#> 1  chr1   300 310   na     1      +
#> 2  chr1   400 410   na     1      -
#> 3  chr1   500 510   na     1      +
#> 4  chr1   600 610   na     1      -
```

center, fiveprime, threeprime

These 3 functions take an original BED file and transform each rows start and end columns. The only difference is where the function determines the anchor point of the window. These functions are strand specific, meaning that if passed a BED6 file `threeprime.bed` and `fiveprime.bed` will work from respective starting points.

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

fiveprime.bed(bed, upstreamWindow, downstreamWindow)

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

- 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.

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. This is rounded down to x. If the difference is even we get x.The anchor point is the `start` + x.

`fiveprime.bed` uses the `start` as the anchor point.

`threeprime.bed` uses the `end` as the anchor point.

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
#>     V1  V2  V3
#> 1 chr1 300 310
#> 2 chr1 400 410
```

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

```
center.bed(bed, upstreamWindow = 0, downstreamWindow = 0)
#>     V1  V2  V3
#> 1 chr1 305 306
#> 2 chr1 405 406
```

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)
#>     V1  V2  V3
#> 1 chr1 300 311
#> 2 chr1 400 411
```

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)
#>     V1  V2  V3
#> 1 chr1 306 310
#> 2 chr1 406 410
```

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)
#>     V1  V2  V3
#> 1 chr1 300 301
#> 2 chr1 400 401
threeprime.bed(bed, upstreamWindow = 0, downstreamWindow = 0)
#>     V1  V2  V3
#> 1 chr1 309 310
#> 2 chr1 409 410
```

`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 299 306
#> 2 chr1 399 406
threeprime.bed(bed, upstreamWindow = 1, downstreamWindow = 5)
#>     V1  V2  V3
#> 1 chr1 308 315
#> 2 chr1 408 415

# negative value
fiveprime.bed(bed, upstreamWindow = -1, downstreamWindow = 5)
#>     V1  V2  V3
#> 1 chr1 301 306
#> 2 chr1 401 406
threeprime.bed(bed, upstreamWindow = -1, downstreamWindow = 5)
#>     V1  V2  V3
#> 1 chr1 310 315
#> 2 chr1 410 415
```

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)
#>   chrom start end name score strand
#> 1  chr1   296 303   na     1      +
#> 2  chr1   408 415   na     1      -
#> 3  chr1   496 503   na     1      +
#> 4  chr1   608 615   na     1      -
threeprime.bed(bed6, upstreamWindow=4, downstreamWindow=2)
```

```
#>   chrom start end name score strand
#> 1  chr1   305 312   na     1      +
#> 2  chr1   398 405   na     1      -
#> 3  chr1   505 512   na     1      +
#> 4  chr1   598 605   na     1      -
```

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`

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)
#>     V1  V2  V3
#> 1 chr1 300 305
#> 2 chr1 400 405
upstream.bed(bed, upstreamWindow = 5)
#>     V1  V2  V3
#> 1 chr1 295 300
#> 2 chr1 395 400
```

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)
#>   chrom start end name score strand
#> 1  chr1   300 305   na     1      +
#> 2  chr1   405 410   na     1      -
#> 3  chr1   500 505   na     1      +
#> 4  chr1   605 610   na     1      -
upstream.bed(bed6,5)
#>   chrom start end name score strand
#> 1  chr1   295 300   na     1      +
#> 2  chr1   410 415   na     1      -
#> 3  chr1   495 500   na     1      +
#> 4  chr1   610 615   na     1      -
```

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)
#> [1] 10 10
```

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.

Region

The following sections group `bpQuery` and `probeQuery` functions together because they operate the same except on how they calculate the average and the ability to incorporate `bwMap` files. `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 a common repeated area.

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, 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.bigWigv`
  - `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`.

- `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

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 the sum of all the instances there are in '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 and uses this as the denominator. . 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
```

gap.value `gap.value` changes how the function handles instances where there was no data returned.

Take a look at the previous 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
```

Notice that if you were to query between 229993 and 230001, that there would be no return.

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

So if you were to run `region.bpQuery.bigWig` on that genomic interval, you would get 0 for all of the operations.

```
region.bpQuery.bigWig(bw, chrom='chr2', start=229993, end=230001, op='min')
#> [1] 0
region.bpQuery.bigWig(bw, chrom='chr2', start=229993, end=230001, op='max')
#> [1] 0
region.bpQuery.bigWig(bw, chrom='chr2', start=229993, end=230001, op='sum')
#> [1] 0
region.bpQuery.bigWig(bw, chrom='chr2', start=229993, end=230001, op='avg')
#> [1] 0
```

By adding `gap.value = 1` or any numeric value, the result is

```
region.bpQuery.bigWig(bw, chrom='chr2', start=229993, end=230001, op='min', gap.value=1)
#> [1] 1
region.bpQuery.bigWig(bw, chrom='chr2', start=229993, end=230001, op='max', gap.value=1)
#> [1] 1
region.bpQuery.bigWig(bw, chrom='chr2', start=229993, end=230001, op='sum', gap.value=1)
#> [1] 1
region.bpQuery.bigWig(bw, chrom='chr2', start=229993, end=230001, op='avg', gap.value=1)
#> [1] 1
```

Notice that the `gap.value` replaces all values no matter what operation is used. This attribute is used more in bed and step variations, as will be seen later.

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))
#> 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.

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)
#>       start    end value
#> [1,] 229991 229992     1
```

Next, a query without any counts

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

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)
#> [1]   0.5 270.0
```

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)
#> [1] 0.5  NA
bed.region.bpQuery.bigWig(bw, bedWgap, op='avg', gap.value=0)
#> [1] 0.5 0.0
bed.region.probeQuery.bigWig(bw, bedWgap, op='avg', gap.value=NA)
#> [1]  1 NA
bed.region.probeQuery.bigWig(bw, bedWgap, op='avg', gap.value=0)
#> [1] 1 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.

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 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)
#>  [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 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)
#> [[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 1.111111
#>  [9] 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. 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)
```

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)
#>      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.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)
```

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

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.

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
```

```
#>   chrom  start    end name score strand
#> 1  chr1      1 100001    a     c      +
#> 2  chr1 200001 300001    a     c      +
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
#>      [,1] [,2]
#> [1,]    1    3
#> [2,]  224    4
#> attr(,"step")
#> [1] 50000
```

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

```
quantiles.metaprofile(mat, quantiles = c(0.875, 0.5, 0.125))
#> $step
#> [1] 50000
#>
#> $top
#>   87.5%   87.5%
#> 196.125   3.875
#>
#> $middle
#>   50%   50%
#> 112.5   3.5
#>
#> $bottom
#>  12.5%  12.5%
#> 28.875  3.125
```

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.

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)
#> $step
#> [1] 50000
#>
#> $top
#> 87.5% 87.5%
#> 112.5   3.5
#>
#> $middle
#>   50%   50%
#> 112.5   3.5
#>
#> $bottom
#> 12.5% 12.5%
```

```
#> 112.5   3.5
```

Confidence Interval

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

```
confinterval.metaprofile(mat, alpha = 0.05)
#> $step
#> [1] 50000
#>
#> $top
#> [1] 205.62702   3.91761
#>
#> $middle
#> [1] 112.5   3.5
#>
#> $bottom
#> [1] 19.37298  3.08239
```

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

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

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]

```
bootstrapped.confinterval.metaprofile(mat, alpha = 0.05, n.samples = 300)
#> $step
#> [1] 50000
#>
#> $top
#> [1] 268.572245   4.199876
#>
#> $middle
#> [1] 112.5   3.5
#>
#> $bottom
#> [1] -51.748911   2.763458
```

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

metaprofile

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

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

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)
#> $name
#> [1] "Signal"
#>
#> $X0
#> [1] 0
#>
#> $step
#> [1] 50000
#>
#> $top
#>   87.5%   87.5%
#> 196.125   3.875
#>
#> $middle
#>   50%   50%
#> 112.5   3.5
#>
#> $bottom
#>  12.5%  12.5%
#> 28.875  3.125
#>
#> attr(,"class")
#> [1] "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.s
#> $name
#> [1] "Signal"
#>
#> $X0
#> [1] 0
#>
#> $step
#> [1] 50000
#>
#> $top
#> [1] 267.105599   4.193299
#>
#> $middle
#> [1] 112.5   3.5
#>
#> $bottom
#> [1] -42.105599   2.806701
#>
#> attr(,"class")
#> [1] "metaprofile"
```

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

RPKM

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

$factor = \frac{(step/1000)}{libsize/1000000}$

```
# Original mat
mat
#>      [,1] [,2]
#> [1,]    1    3
#> [2,]  224    4
#> attr(,"step")
#> [1] 50000

rpkm.scale(mat, step=50000, libSize=1000000)
#>       [,1] [,2]
#> [1,]    50  150
#> [2,] 11200  200
#> attr(,"step")
#> [1] 50000
```

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)
#>           [,1]       [,2]
#> [1,] 0.2500000 0.75000000
#> [2,] 0.9824561 0.01754386
```

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
#>      [,1] [,2]
#> [1,]    0    0
#> [2,]    4    9

densityToOne.scale(mat1, na.on.zero = TRUE)
#>           [,1]      [,2]
#> [1,]        NA        NA
#> [2,] 0.3076923 0.6923077
densityToOne.scale(mat1, na.on.zero = FALSE)
#>           [,1]      [,2]
```

```
#> [1,] 0.0000000 0.0000000
#> [2,] 0.3076923 0.6923077
```

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
#>      [,1] [,2]
#> [1,]    1    3
#> [2,]  224    4
#> attr(,"step")
#> [1] 50000
maxToOne.scale(mat)
#>           [,1]       [,2]
#> [1,] 0.3333333 1.00000000
#> [2,] 1.0000000 0.01785714
mat1
#>      [,1] [,2]
#> [1,]    0    0
#> [2,]    4    9
maxToOne.scale(mat1)
#>           [,1] [,2]
#> [1,] 0.0000000    0
#> [2,] 0.4444444    1
```

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

Zero to one

`zeroToOne.scale` compares the differences between the max and min of each row. It uses the following formula. $x_n = \frac{x_o - min}{max - min}$

```
mat
#>      [,1] [,2]
#> [1,]    1    3
#> [2,]  224    4
#> attr(,"step")
#> [1] 50000
maxToOne.scale(mat)
#>           [,1]       [,2]
#> [1,] 0.3333333 1.00000000
#> [2,] 1.0000000 0.01785714
```

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
#>      [,1] [,2]
#> [1,]    0    0
#> [2,]    4    9
maxToOne.scale(mat1)
#>           [,1] [,2]
#> [1,] 0.0000000    0
```

```
#> [2,] 0.4444444    1
```

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

```
mat2
#>      [,1] [,2]
#> [1,]    2    2
#> [2,]    4    9
zeroToOne.scale(mat2)
#>      [,1] [,2]
#> [1,]    1    1
#> [2,]    0    1
```

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)
#> $name
#> [1] "Signal"
#>
#> $X0
#> [1] 0
#>
#> $top
#> [1] 1.157685 1.224351
#>
#> $middle
#> [1] 0.5 0.5
#>
#> $bottom
#> [1] -0.2243512 -0.1576846
#>
#> attr(,"class")
#> [1] "metaprofile"
```

bwMap

plots.bigWig

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

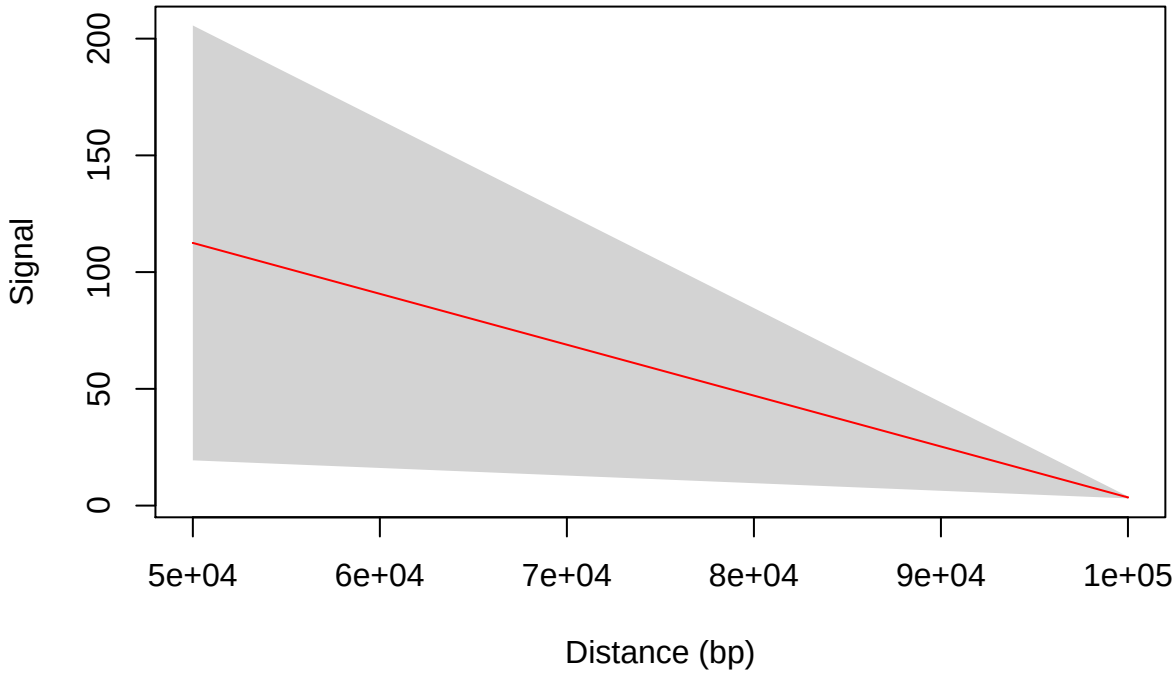
```
plot.metaprofile(x, minus.profile = NULL, X0 = x$X0,
      draw.error = TRUE, col = c("red", "blue", "lightgrey", "lightgrey"),
      ylim = NULL, xlim = NULL, xlab = "Distance (bp)", ylab = x$name)
```

- 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`.

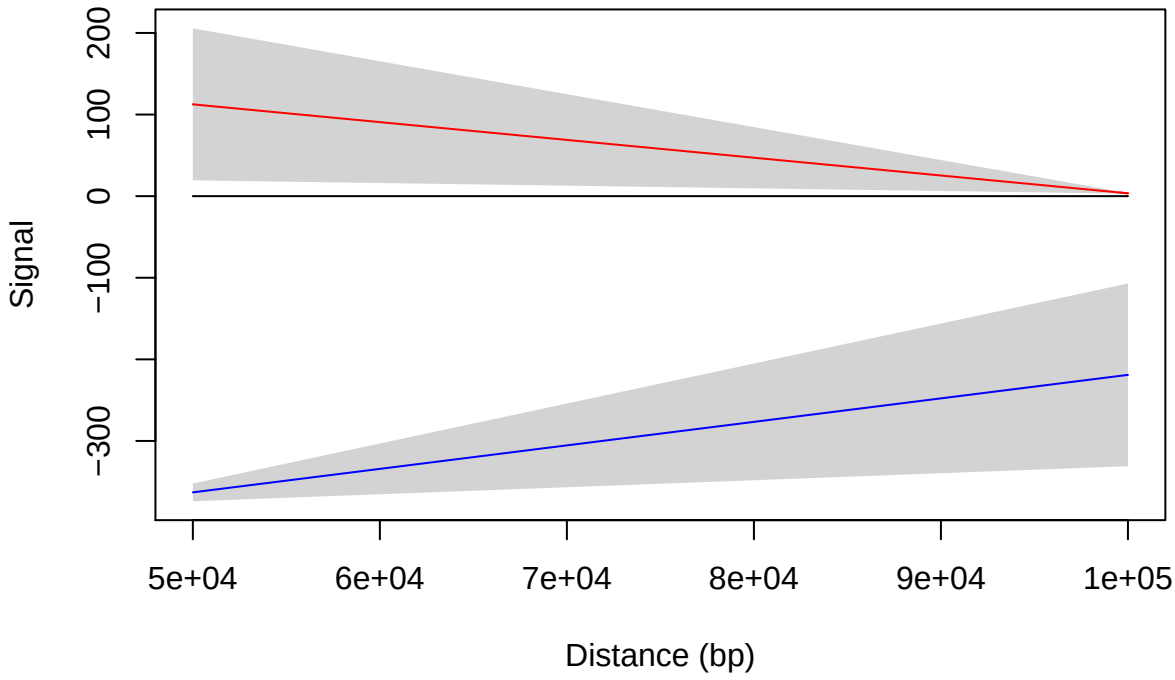
```
x=metaprofile.bigWig(bed6, bw.plus, bw.minus = bw.minus, step = 50000, name = "Signal",
                  matrix.op = NULL, profile.op = confinterval.metaprofile)

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)
```



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)
```

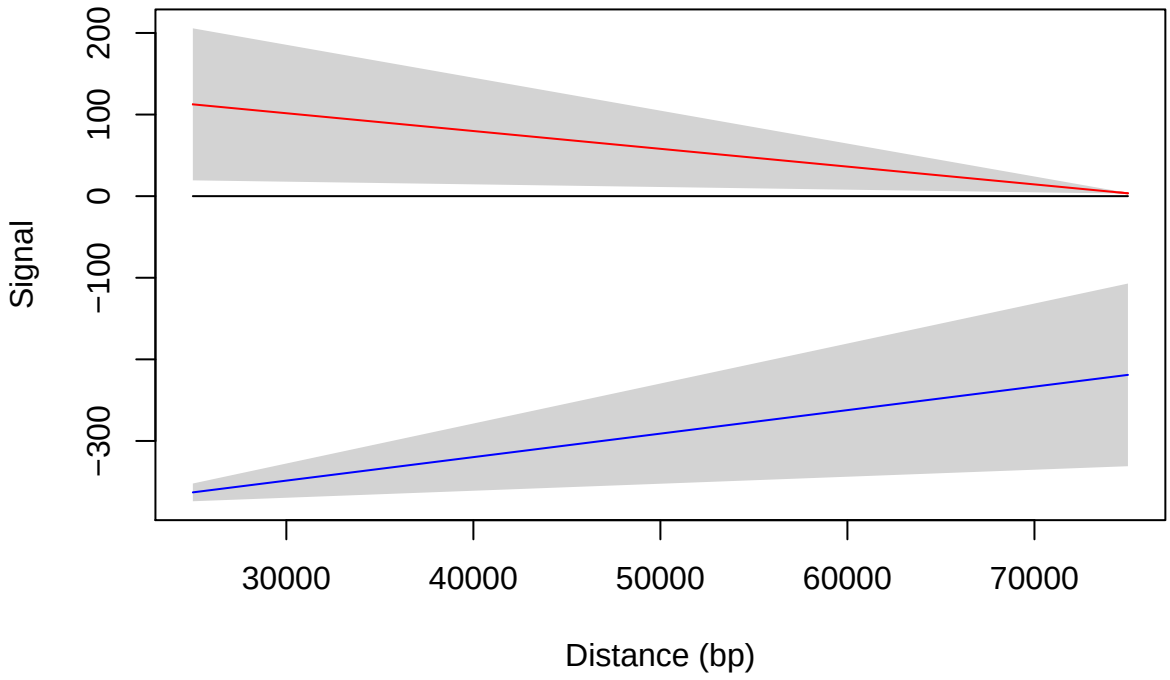


Note that `xr` switches the `bw.plus` and `bw.plus` inputs. This gives our reverse strand because from the original data they are the reverse of each other.

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)
```

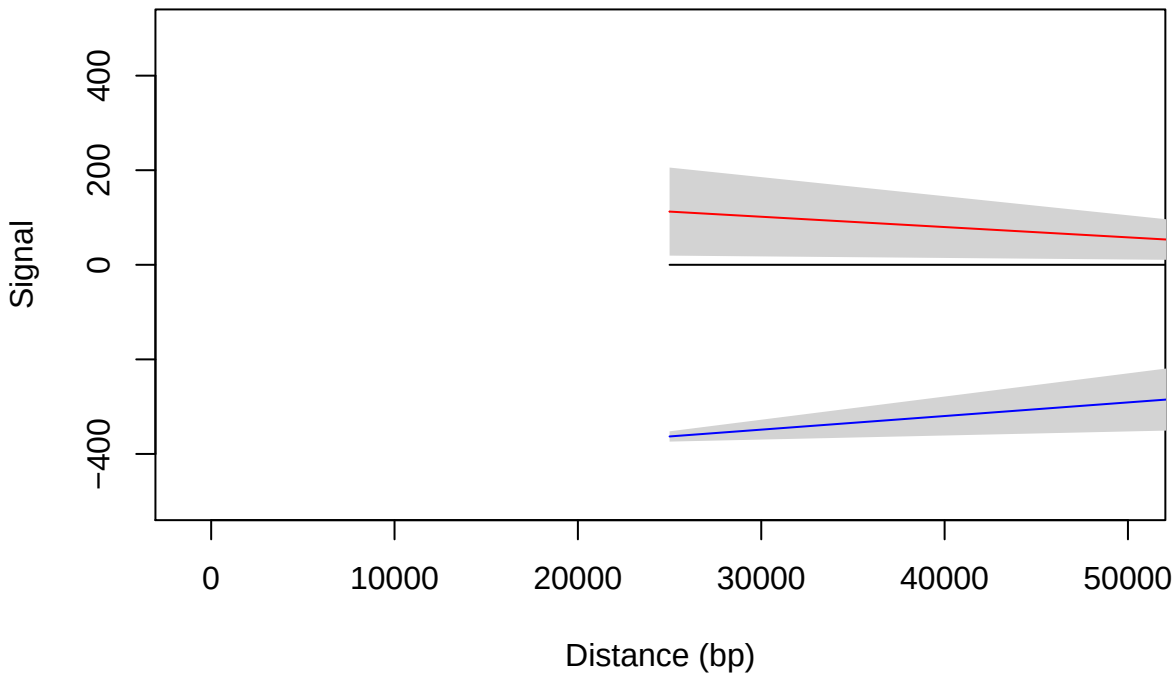


Notice the x axis distance changes. This is because you are setting the start to be offset by 25000.

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)
```

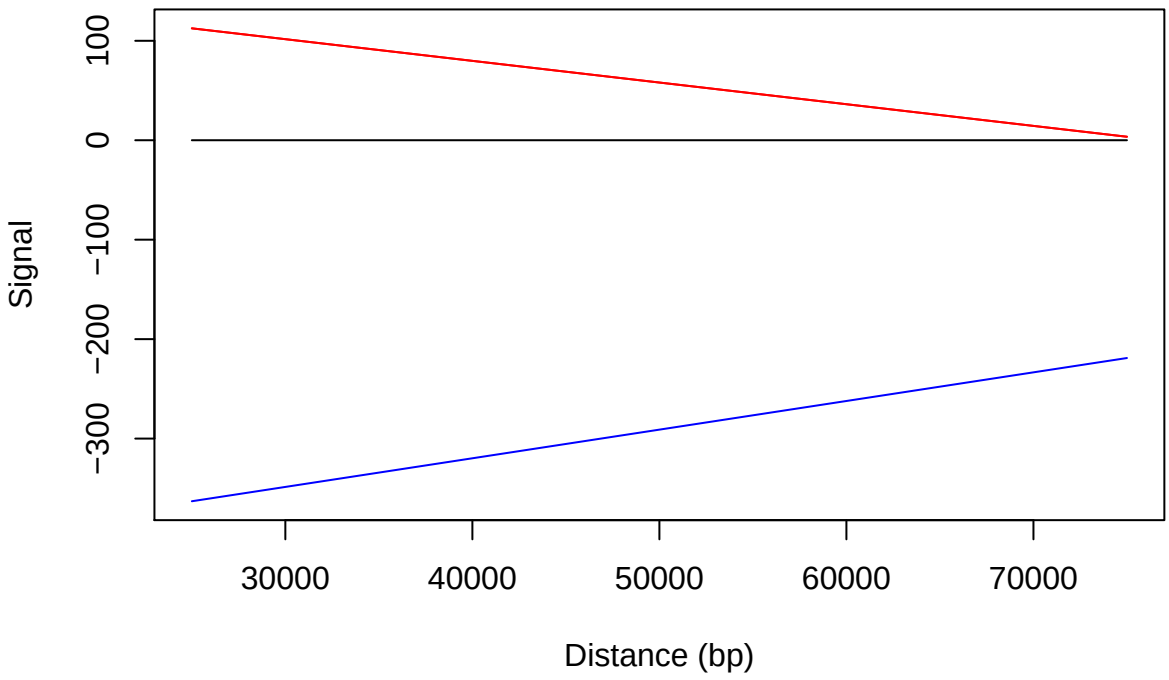


Looking at the axes, you can ow see the change in lower and upper limits.

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.

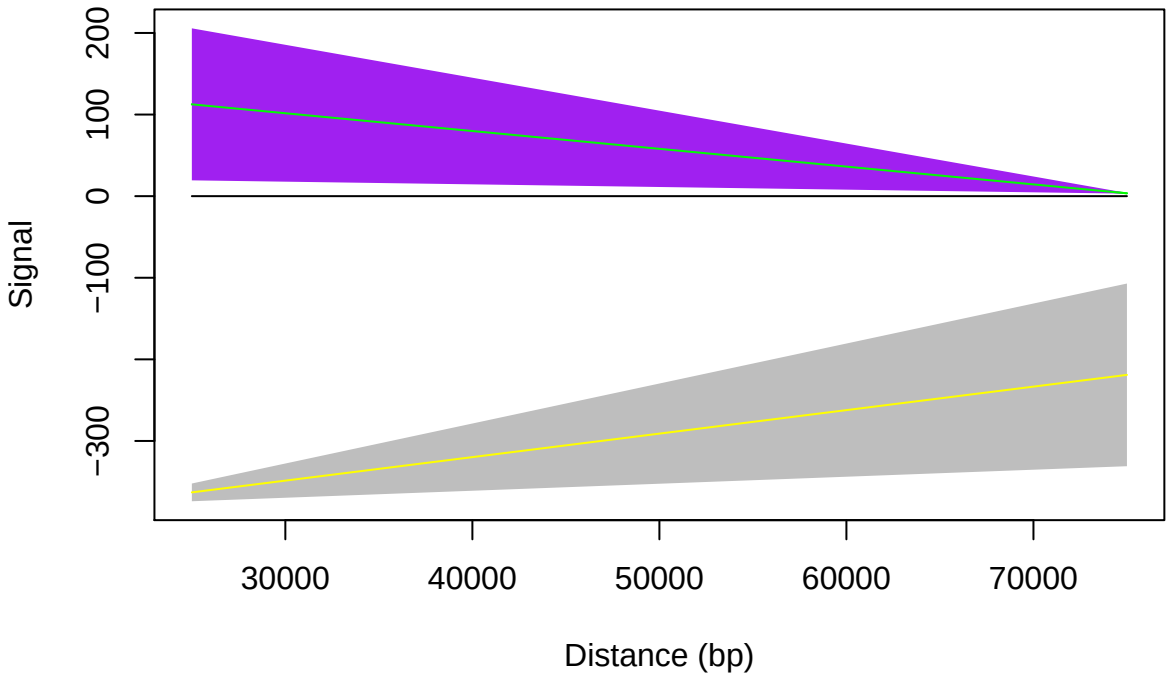
Colors

`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]
#>  [1] "white"           "aliceblue"       "antiquewhite"    "antiquewhite1"
#>  [5] "antiquewhite2"   "antiquewhite3"   "antiquewhite4"   "aquamarine"
#>  [9] "aquamarine1"     "aquamarine2"     "aquamarine3"     "aquamarine4"
#> [13] "azure"           "azure1"          "azure2"          "azure3"
#> [17] "azure4"          "beige"           "bisque"          "bisque1"
#> [21] "bisque2"         "bisque3"         "bisque4"         "black"
#> [25] "blanchedalmond"
```

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)
```


Signal
200
100
0
−100
−300
30000
40000
50000
60000
70000
Distance (bp)