regioneR: an R package for the management and statistical comparison of genomic regions

Bernat Gel*, Anna Díez-Villanueva and Roberto Malinverni

June 9, 2015

Contents

*bgel@imppc.org

1 Introduction

The important technological advances we have seen since the publication of the human genome such as microarrays and NGS technologies have allowed us to produce massive amounts of data, making data analysis and interpretation the bottleneck in genomic and epigenomic research. In many cases, this data can be converted to a set of regions in the genome: the positions of the genes in an expression microarray experiment, the regions with peaks in ChIP-seq data or the contact regions in Hi-C. Statistically assessing the spatial relations between these region sets and other region sets or even other genomic features is a fundamental part of this analysis.

regioneR has been created to address this problem and provides functions to statistically evaluate the associations between region sets using permutation tests. Its permutation test framework has been specifically designed to work with genomic regions and all functions are genome- and mask-aware. regioneR includes a number of predefined randomization and evaluation functions covering the most frequent use cases, but the user can also provide custom functions to extend its functionality. In addition to textual output, regioneR also can plot the permutation test results and so it's possible to go from a set of genomic regions to a publication ready figure in a few lines of code.

In addition, regionerR includes a set of helper functions based on Bioconductor's GenomicRanges infrastructure to manage and manipulate region sets with a simple and consistent interface.

2 Permutation Tests

The core functionality of regioneR is to statistically evaluate the association between different RS or between a RS and other genomic features using a permutation test approach. This functionality is supported by the *permTest* function, a parallel and highly customizable function performing the permutation tests and producing the statistical evaluation of the results.

With *permTest* and the right biological data it is possible to answer different types of biological questions such as:

- Do my regions contain more SNPs than expected by chance?
- Are my regions enriched in repetitive regions?
- Are my regions significantly closer to genes?
- Are DNA methylation levels in my RS higher than expected?
- Do my regions overlap with recurrent SCNAs in my samples?
- Are my ChIP peaks associated with that other histone mark?
- Do my ChIP regions associate with repressed genes?

NOTE: It is important to take into account, however, that regioneR's permutation tests can only test the association between a set of regions and some other feature but not identify which regions contribute the most to that association.

Therefore, for questions of the type Identify the region in my RS that associate with something, regioneR is not a suitable choice, and more specific analysis tools should be used.

2.1 How Does a Permutation Test Work?

There are 3 main elements needed to perform a permutation test: our RS, a randomization strategy and an evaluation function. Let's see how it all works with an example:

Imagine we have obtained a set of genes, my special genes, and we want to show they tend to lie in certain parts of the genome, in our case a set of regions we know are altered, for example, they present a copy number gain.

First of all we need to have our RSs loaded into R. To do that we can use the *toGRanges* function.

```
special <- toGRanges("http://gattaca.imppc.org/regioner/data/my.special.genes.bed")
all.genes <- toGRanges("http://gattaca.imppc.org/regioner/data/all.genes.bed")
altered <- toGRanges("http://gattaca.imppc.org/regioner/data/my.altered.regions.bed")
length(special)

## [1] 200

length(all.genes)

## [1] 49646

length(altered)

## [1] 8
```

Next thing we need is an evaluation function. In our example we want to test the overlap of our RS with the altered regions, so we will use the *numOverlaps* function, which, given two RS returns the number of overlaps between them.

Using *numOverlaps* we can count the number of overlaps between the two RS, in our case we can compute the number of special genes overlapping an altered region.

```
numOverlaps(special, altered)

## [1] 114
```

But is this number, 114 out of 200, big or small? Does it mean that genes are associated with the altered regions? or might be just by chance?

Here is where the randomization strategy plays its role. We need a randomization strategy that creates a new set of regions that is random with respect to our evaluation function but takes into account the specificities of our original

region set. For example, if our original RS comes from an NGS experiment, all of its regions would lie in mappable parts of the genome and wouldn't make any sense to randomize a region into a centromere or any other non-mappable part of the genome.

To help with that, many randomization functions provided by regioneR accept a mask, indicating where a random region cannot be placed. In any case, selecting the best randomization strategy is a key part of a permutation test and can have a great impact in the final results.

The least restricted function included in regioneR is *randomizeRegions*, that given a RS, a genome and an optional mask, returns a new RS with the same number of regions and of the same width as the original ones but randomly placed along the non-masked parts of the genome. This is also the slowest of the randomization functions available.

In our example, since the special genes are a subset of the bigger set of all genes it is much better to use *resampleRegions*, that given a universe of regions, randomly selects a subset of them to create the randomized region sets.

```
random.RS <- resampleRegions(special, universe=all.genes)
random.RS

## GRanges object with 200 ranges and 2 metadata columns:
##         seqnames                ranges strand   |           V4       V6
##            <Rle>             <IRanges>  <Rle>   |     <factor> <factor>
##     [1]    chr20 [ 50069441,  50069514]      *   |    NR_036162        -
##     [2]     chr6 [ 31110215,  31125566]      *   | NM_001105563        -
##     [3]    chr12 [ 57647547,  57704246]      *   |    NM_014925        -
##     [4]     chrX [123509755, 124097666]      *   | NM_001163279        -
##     [5]     chr2 [ 88838237,  88875128]      *   |    NR_110236        +
##     ...      ...                   ...    ... ...          ...      ...
##   [196]    chr14 [ 55308723,  55369542]      *   | NM_001024070        -
##   [197]    chr12 [ 53907767,  53911392]      *   |    NR_046221        +
##   [198]     chr7 [ 94927668,  94953884]      *   |    NM_000446        -
##   [199]     chr1 [236849753, 236927927]      *   |    NM_001103        +
##   [200]    chr19 [ 46913585,  46916919]      *   |    NM_032040        -
##   -------
##   seqinfo: 24 sequences from an unspecified genome; no seqlengths
```

Now, we can use the evaluation function to test the level of association of the randomized RS with the altered regions. And we can repeat that and get different evaluations.

```
random.RS <- resampleRegions(special, universe=all.genes)
numOverlaps(random.RS, altered)
```

```
## [1] 42

random.RS <- resampleRegions(special, universe=all.genes)
numOverlaps(random.RS, altered)

## [1] 61

random.RS <- resampleRegions(special, universe=all.genes)
numOverlaps(random.RS, altered)

## [1] 54

random.RS <- resampleRegions(special, universe=all.genes)
numOverlaps(random.RS, altered)

## [1] 51
```

If we do this many times we will build a distribution of the evaluation obtained from random RS and so, we can compare our initial evaluation with those obtained randomly and determine whether it is plausible that our original evaluation was obtained by chance or not. Actually, just counting the number of times the evaluation of the random RS is higher (or lower) than our original evaluation, we can compute the probability of seeing our original evaluation by chance, and that value is exactly the p-value of the permutation test. In addition, we compute the z-score which is the distance between the evaluation of the original RS and the mean of the random evaluations divided by the standard deviation of the random evaluations. The z-score, although not directly comparable, can help in assessing "the strength" of the evaluation.

2.2 How to perform a permutation test with regioneR

The main function to perform a permutation test with regioneR is `permTest`, a function taking a a region set (RS) in any of the accepted formats (see section 4 - Region Sets), a randomization function and an evaluation function and returning a *permTestResults* object.

The function performs the whole permutation test analysis described above, evaluating the original RS, creating a number of randomizations and evaluating them and nally computing the p-value and z-score. It takes advantage of the parallel package to randomize and evaluate in parallel where possible.

In addition to the 3 required parameters, *permTest* accepts other fixed parameters -*ntimes* to specify the number of randomizations, *verbose* to toggle the drawing of a progress bar, *force.parallel* to force or forbid the use of multiple cores to run the analysis...- and it also accepts any additional parameter required by the randomization function (usually a genome and a mask) or the evaluation function.

For example to check whether my regions overlap with repeats more than expected, we could use:

```
# NOT RUN
pt <- permTest(A=my.regions, B=repeats, randomize.function=randomizeRegions,
evaluate.function=overlapRegions)
```

or if we want to check if my special genes have higher methylation levels we could use a test like this:

```
# NOT RUN
pt <- permTest(A=my.genes, randomize.function=resampleRegions, universe=all.genes,
evaluate.function=meanInRegions, x=methylation.levels.450K)
```

Following our example before with my special genes and the altered regions we could call *permTest* like this:

```
pt <- permTest(A=special, ntimes=50, randomize.function=resampleRegions, universe=all.genes,
 evaluate.function=numOverlaps, B=altered, verbose=FALSE)

## [1] "Note: The minimum p-value with only 50 permutations is 0.0196078431372549. You should consider inc
```

NOTE: Since *permTest* uses the ellipsis operator (...) to forward the required additional parameters to the evaluation and randomization functions it is strongly recommended to always use named parameters (e.g. A=RS1 instead of only RS1). Failing to do that can result in hard to debug errors.

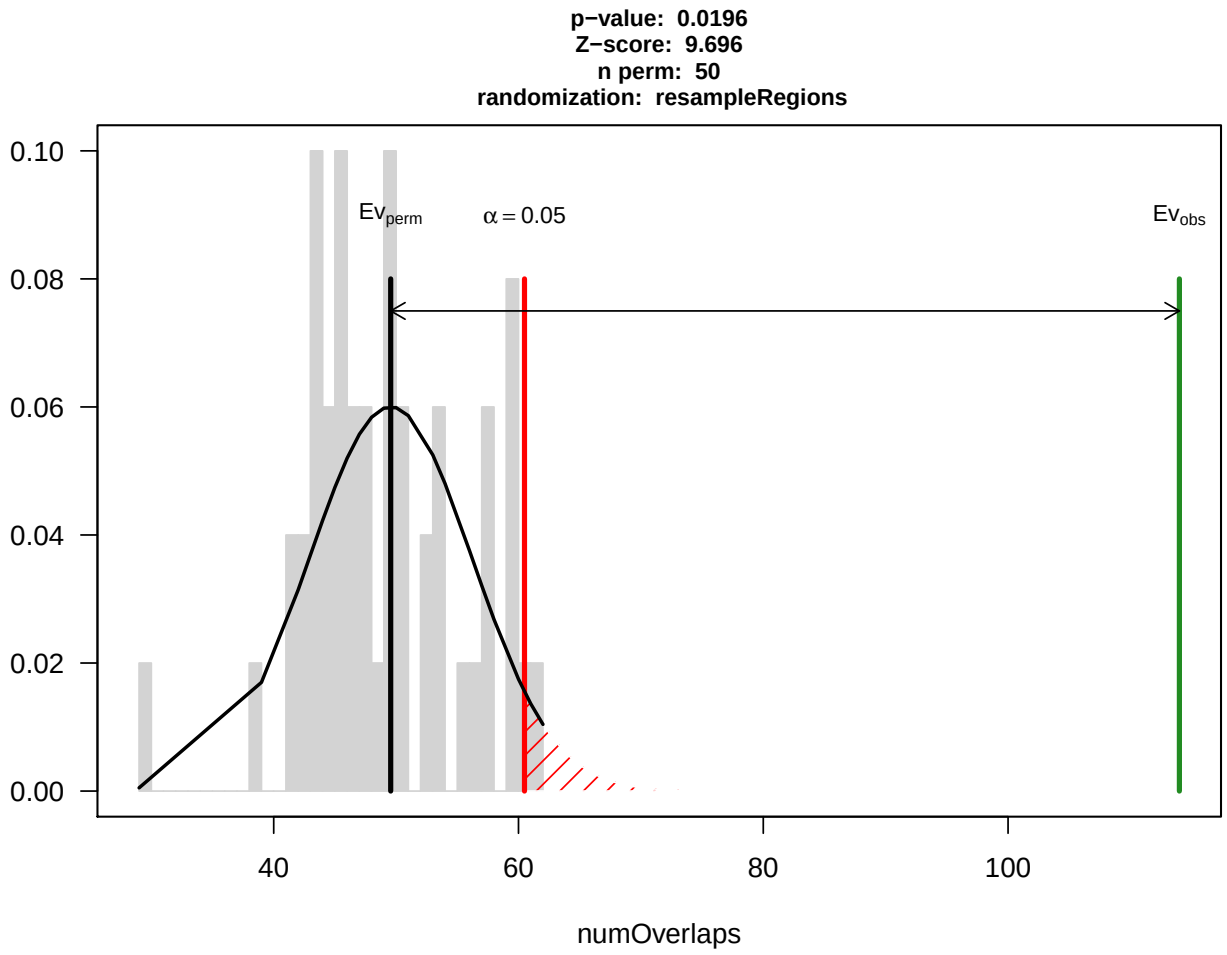
In any case we would get a *permTestResults* object with the results of the analysis. To view this result we can just print it or use summary. In this case we can see that the association is statistically significant and so we can conclude that the special genes are associated with my altered regions.

```
pt

## P-value: 0.0196078431372549
## Z-score: 9.696
## Number of iterations: 50
## Alternative: greater
## Evaluation of the original region set: 114
## Evaluation function: numOverlaps
## Randomization function: resampleRegions

summary(pt)

## Number of permutations: 50
##
## Alternative: greater
```

```
##
## Evaluation of the original region set: 114
##
## Summary of the evaluation of the permuted region set:
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   29.00   45.00   48.50   49.56   54.00   62.00
##
##
## Z-score: 9.696
##
## P-value: 0.0196078431372549 *
## ---
##  Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

And we can get a graphic representing the results of the permutation test using `plot`. It depicts a gray histogram representing the evaluation of the randomized RS with a fitted normal, a black bar representing the mean of the randomized evaluations and a green bar representing the evaluation of the original RS. In addition, a red bar (and red shading) represents the significance limit (by default 0.05). Thus, if the green bar is in the red shaded region it means that the original evaluation is extremely unlikely and so the p-value will be significant.

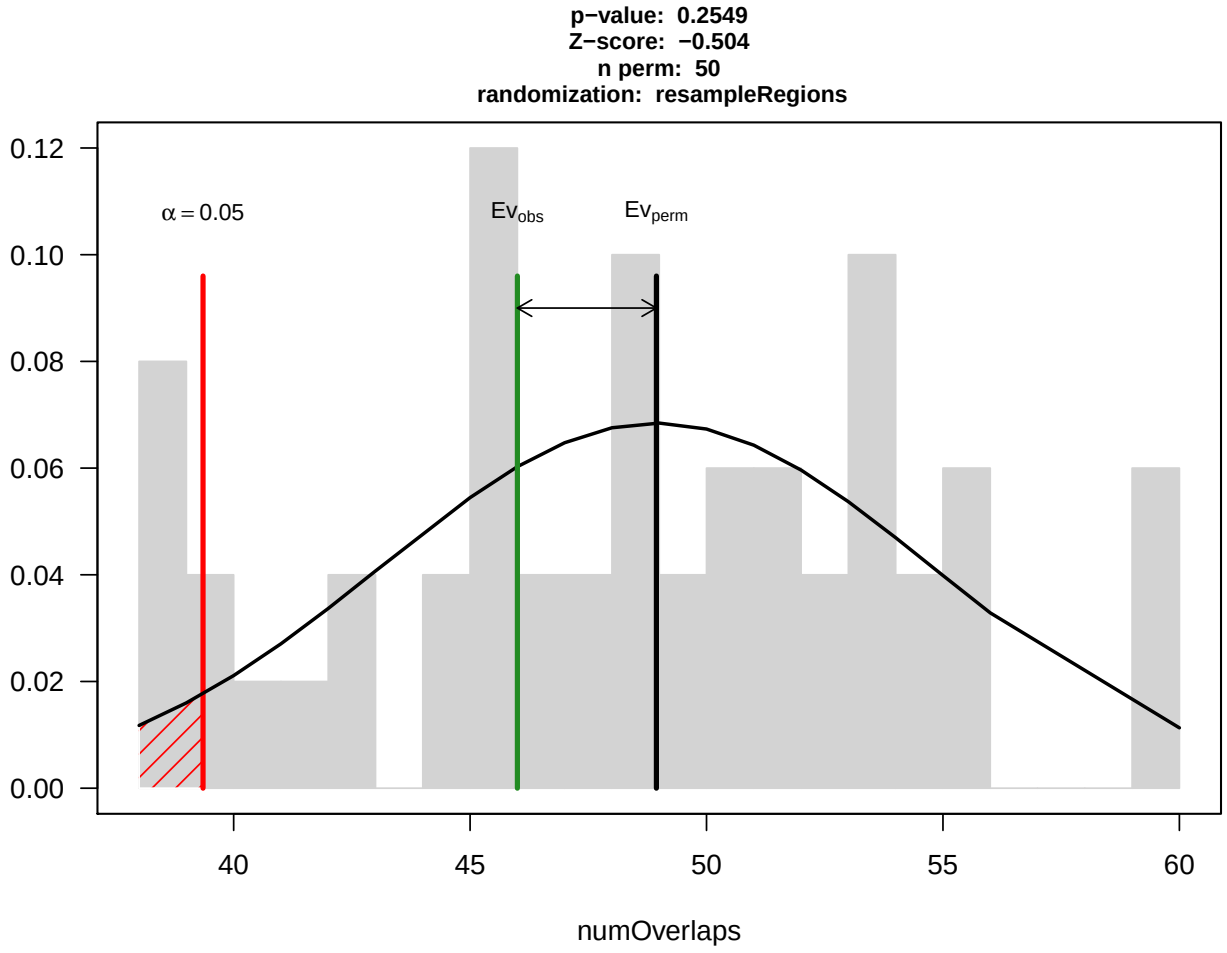
```
plot(pt)
```



To compare, imagine we have a second subset of genes, my regular genes, that are not associated with the altered regions. We can run the same test with them and we will get a negative result. In this case we get a non-significant p-value and we can see in the plot that the original evaluation is close to the mean of the randomized ones.

```
regular <- toGRanges("http://gattaca.imppc.org/regioner/data/my.regular.genes.bed")
length(regular)

## [1] 200

numOverlaps(regular, altered)

## [1] 46

pt.reg <- permTest(A=regular, ntimes=50, randomize.function=resampleRegions, universe=all.genes,
 evaluate.function=numOverlaps, B=altered, verbose=FALSE)

## [1] "Note: The minimum p-value with only 50 permutations is 0.0196078431372549. You should consider inc
```

```
pt.reg

## P-value: 0.254901960784314
## Z-score: -0.5044
## Number of iterations: 50
## Alternative: less
## Evaluation of the original region set: 46
## Evaluation function: numOverlaps
## Randomization function: resampleRegions

plot(pt.reg)
```



2.3 A note on the number of permutations

Choosing the right number of permutations is not a simple task. A large number of permutations will produce more accurate results and a nicer-looking plot but a permutation test can be computationally expensive and depending on the number of regions in the RS and the randomization strategy selected it might take up to several hours to perform a permutation test with a few thousand permutations. On the other hand, the lowest p-value is limited by the number of permutations and performing a permutation test with a low permutation number can produce less accurate results. A good strategy could be to try first with a low number of permutations and continue only if the results look promising or at least unclear, since if after some tens of permutations the original evaluation is really close to the mean of the randomizations, the probability it will end up being significant is really small. With a low number of permutations, regioneR will generate a note stating the lowest p-value achievable and ecouraging to increase the permutation number.

As an example, the two permutation tests above, if run with 5000 permutations would produce a plot like these.

```
#NOT RUN - See Figure 1
pt.5000 <- permTest(A=special, ntimes=5000, randomize.function=resampleRegions,
universe=all.genes, evaluate.function=numOverlaps, B=altered, verbose=TRUE)
plot(pt.5000)
```

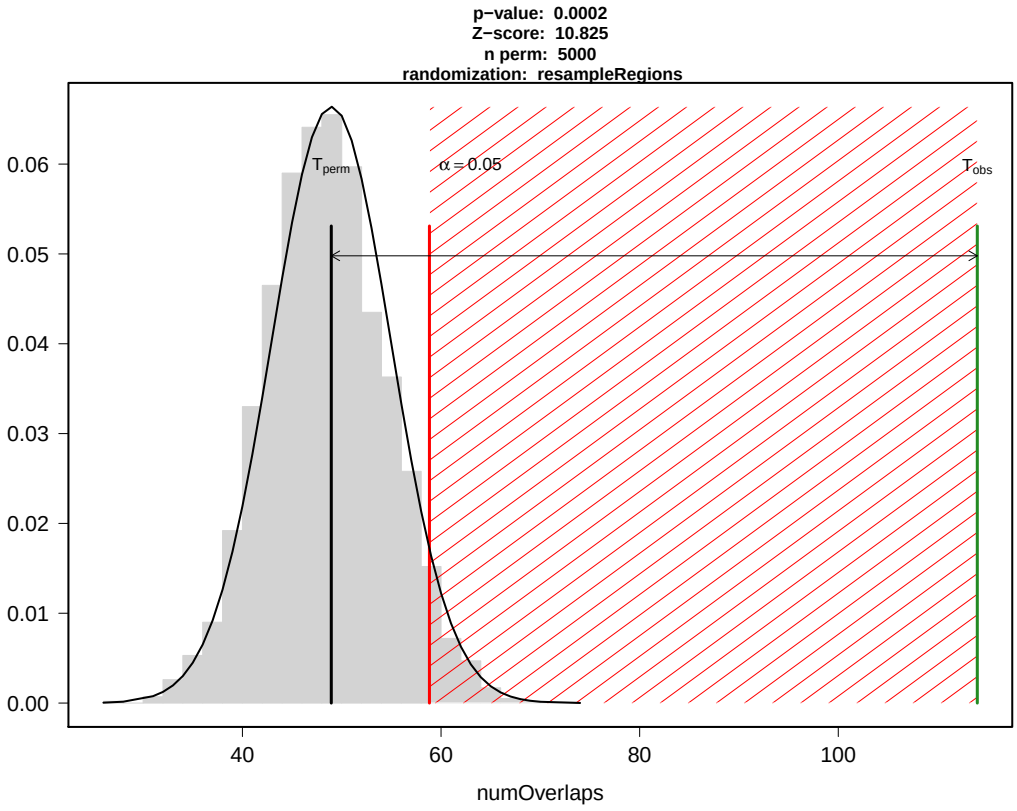


Figure 1: Significant permutation test This is the plot of a significant permutation test with 5000 permutations.

```
#NOT RUN - See Figure 2
pt.5000.reg <- permTest(A=regular, ntimes=5000, randomize.function=resampleRegions,
universe=all.genes, evaluate.function=numOverlaps, B=altered, verbose=TRUE)
plot(pt.5000.reg)
```

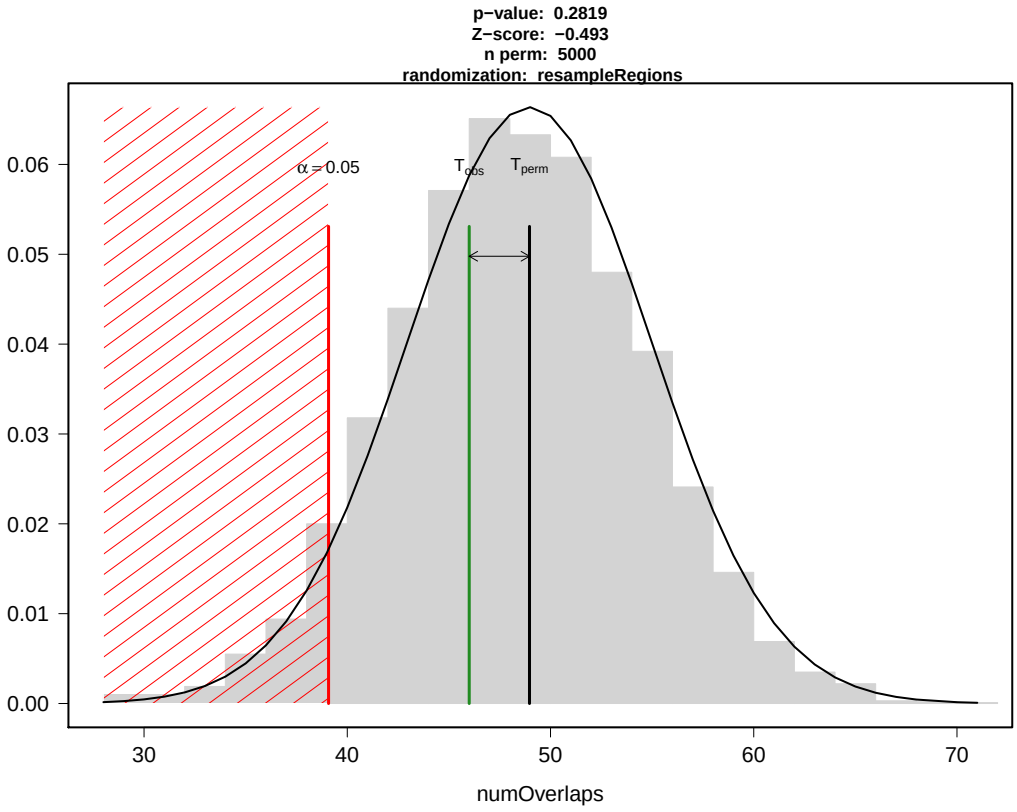


Figure 2: Non significant permutation test This is the plot of a non significant permutation test with 5000 permutations.

2.4 Randomization Functions

Choosing a good randomization strategy is crucial when performing a permutation test. One should choose a randomization strategy that ramdomizes the regions with respect to the association being evaluated while maintaining as much as possible the intrinsic structure and characteristics of the regions. For example, in general it will not make sense to randomize a region into a centromere or into any unavailable region such an unmappable region with NGS data.

In regioneR, a randomization function is any function that takes a RS as an argument and returns a RS with the randomized regions. Different randomization functions are included in the package, but it is also possible to create

custom randomization functions for more specific needs (see section 4 - Region Sets).

2.4.1 randomizeRegions

This is the most general randomization strategy. It randomly places all regions along the genome independently. The function takes every region in the original RS and randomizes the chromosome and position, while maintaining its size and any associated metadata. The optional mask argument might be used to specify parts of the genome where a region cannot be placed (e.g. centromeres).

In addition there are two addition parameters:

- *non.overlapping* (default=FALSE): by default, two randomized regions can overlap. Setting non.overlapping to TRUE will force all randomized regions to be non-overlapping. Creating non-overlapping regions is slower.
- *per.chromosome* (default=FALSE): if this option is set to true, the chromosome is not randomized, but only the position within it. This option can be useful when there original data is intrinsically biased towards some chromosomes or there is an association between the regions and the crhomosomes.

As an example, we can create a RS A with 3 regions in chromosome 1.

```
A <- toGRanges(data.frame(chr=c("chr1", "chr1", "chr1"), start=c(20000, 50000, 100000),
end=c(22000, 70000, 400000)))
```

We can randomize it along the whole genome:

```
randomizeRegions(A, genome="hg19")

## GRanges object with 3 ranges and 0 metadata columns:
##       seqnames                ranges strand
##          <Rle>             <IRanges>  <Rle>
##   [1]     chr4 [ 26440312,  26442313]      *
##   [2]     chr1 [ 39245771,  39265772]      *
##   [3]     chr5 [123098543, 123398544]      *
##   -------
##   seqinfo: 3 sequences from an unspecified genome; no seqlengths
```

In contrast, if we set per.chromosome to TRUE, all randomized regions will be in chr1.

```
randomizeRegions(A, genome="hg19", per.chromosome=TRUE)

## GRanges object with 3 ranges and 0 metadata columns:
##       seqnames                ranges strand
```

```
##          <Rle>              <IRanges>  <Rle>
##   [1]     chr1 [ 48748951,  48750952]      *
##   [2]     chr1 [212739312, 212759313]      *
##   [3]     chr1 [149120918, 149420919]      *
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

2.4.2 circularRandomizeRegions

Another randomization function available in regioneR is the circularRandomizeRegion. In this case, instead of randomizing all regions independently, the randomization process maintains the order and distance of the regions, while changing their position in the chromosome. Conceptually, each chromosome is "circularized" and given a random spin while keeping the regions steady. If a mask is specified, in this case, given the greater constrains in the randomization process, it is not possible to ensure that no region will overlap the mask. Instead of that the user can specify the maximum proportion of the regions overlapping the mask. If apter spinning the chromosome too much overlap is detected, a new random spin will be applied. This process will be repeated until a suitable solution if found or max.retries spins have been applied. If after the spinning, a region lies between the end and the start of the chromosome, it will be splitted into two regions. Therefore, the number of regions is not guarateed to be the same as teh original RS. `circularRandomizeRegions` can be significantly faster than `randomizeRegions`.

As an example, we can randomize the same the same RS as before, and observe how the randomized regions are in the same chromosome and at the same relative distance:

```
circularRandomizeRegions(A,genome="hg19")
```

```
## GRanges object with 3 ranges and 0 metadata columns:
##       seqnames                ranges strand
##          <Rle>             <IRanges>  <Rle>
##   [1]     chr1 [124543107, 124545107]      *
##   [2]     chr1 [124573107, 124593107]      *
##   [3]     chr1 [124623107, 124923107]      *
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

2.4.3 resampleRegions

The last available randomization function is resampleRegions. In this case, the original RS is a subset of a finite set of all valid regions called universe. For example, a small set of genes as a subset of all genes in the genome. The randomization process in this case consists in selecting other random members of the universe, it is, resampling it.

2.5 Evaluation Functions

The evaluation functions are the functions that define what is being tested in the permutation test. Some evaluation functions are provided in regioneR covering the most common uses but it is possible (and easy) to create your own custom evaluation functions (see section 2.6.1). In general, an evaluation function is simply a function taking a RS and returning a single numeric value.

2.5.1 numOverlaps

The numOverlaps function receives 2 RS A and B and returns the number of regions in A that overlap a region in B. This is the evaluation function used when testing whether our RS A overlaps with a second RS more (or less) than expected by chance -i.e Do my set of ChIP-seq peaks tend to lie on CpG islands?

2.5.2 meanDistance

If instead of evaluating the overlap between two RS we want to test their distance, we can use meanDistance, which given two RS A and B, computes the mean of the distance from every region in A to the closest region in B. It is useful to answer question of the type "Are my highly expressed genes closer to a certain TFBS than expected by chance?"

2.5.3 meanInRegion

The meanInRegion function is useful when we want to evaluate the relation of a RS against some sort of numeric value. For example, we could test whether the methylation levels of our RS is higher than expected, or if it accumulates fewer mutation than one would expect.

2.6 Custom Functions

It is possible to use custom randomization and evaluation strategies in permTest. To do so, the user simply needs to create a suitable function and give it as an argument to the permTest function.

2.6.1 Custom Evaluation

An evaluation function can be any function with at least a RS as its first parameter and an ellipsis (...) returning a single numeric value. It can also include any other parameters.

For example, this is a valid (although very slow) function to evaluate the GC content of our regions.

```
gcContent <- function(A, bsgenome, ...) {
  A <- toGRanges(A)
  reg.seqs <- getSeq(bsgenome, A)
  base.frequency <- alphabetFrequency(reg.seqs)
  gc.pct <- (sum(base.frequency[,"C"]) + sum(base.frequency[,"G"]))/sum(width(A))
  return(gc.pct)
}
```

2.6.2 Custom Randomization

A randomization function is any function with at least a RS and an ellipsis as parameters and returning a RS. Defining a different and meaningful randomization function, though, is not as easy as coming up with an interesting evaluation function.

For example, a valid randomization strategy could be to permute the metada values associated with each region. This is a simple function that permutes the first column of metadata and returns the permuted RS.

```
permuteRegionsMetadata <- function(A, ...) {
  A <- toGRanges(A)
  mcols(A)[,1] <- mcols(A)[sample(length(A)),1]
  return(A)
}
```

3 Local Z-score

The minimum p-value achievable by permutation depends on the number of permutations used. On the other hand, the z-score is a measure of the strength of the association that is independent of the number of permutations. It is defined as the distance between the expected value and the observed one, measured in standard deviations.

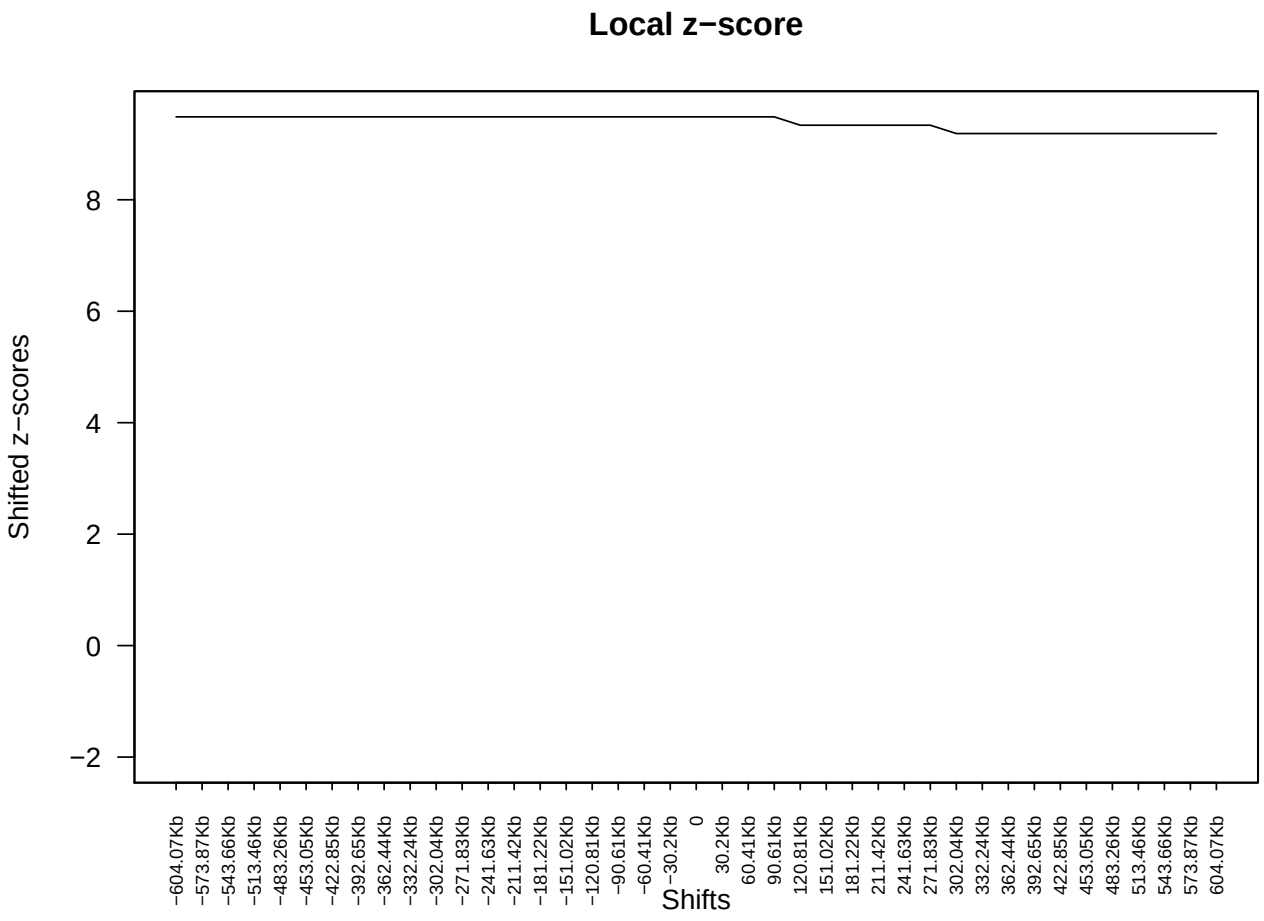
When performing an association analysis it is possible to detect associations that, while statistically significant, are not biologically relevant. For example, with ChIP-seq data, it is uaual to detect a significant overlap between broad chromatin

marks covering gene-rich regions and transcription factors. This association, while true, is indirect and based on the fact that both regions tend to cluster around genes.

While we can't detect if an association is biologically relevant, with the localZscore function we can at least check if the association is specifically linked to the exact position of the regions in the RS. To do that, the RS is shifted a number of bases to 5' and 3' from its original position and the evaluation function is computed for every shifted RS. Plotting the evaluations over the shifted positions we can estimate how the value of the z-score changes when moving the regions in the RS: a sharp peak at the center indicates that the association is highly dependent on the exact position of the regions while a flat profile will indicate that the association is regional, since we can move the regions around and obtain the same association values.

For example, in a previous example, we have identified a positive association between a set of "special" genes and a set of altered regions in the genome. The special genes tend to lie inside these alterede regions, but is the association highly dependant on their exact position? Or is it a regional association? To test that, we can move around the position of the genes as evaluate how this position change would affect the z-score.

```
pt <- permTest(A=special, ntimes=50, randomize.function=resampleRegions, universe=all.genes,
               evaluate.function=numOverlaps, B=altered, verbose=FALSE)

## [1] "Note: The minimum p-value with only 50 permutations is 0.0196078431372549. You should consider inc

lz <- localZScore(A=special, pt=pt, window=10*mean(width(special)),
                  step=mean(width(special))/2, B=altered)
plot(lz)
```

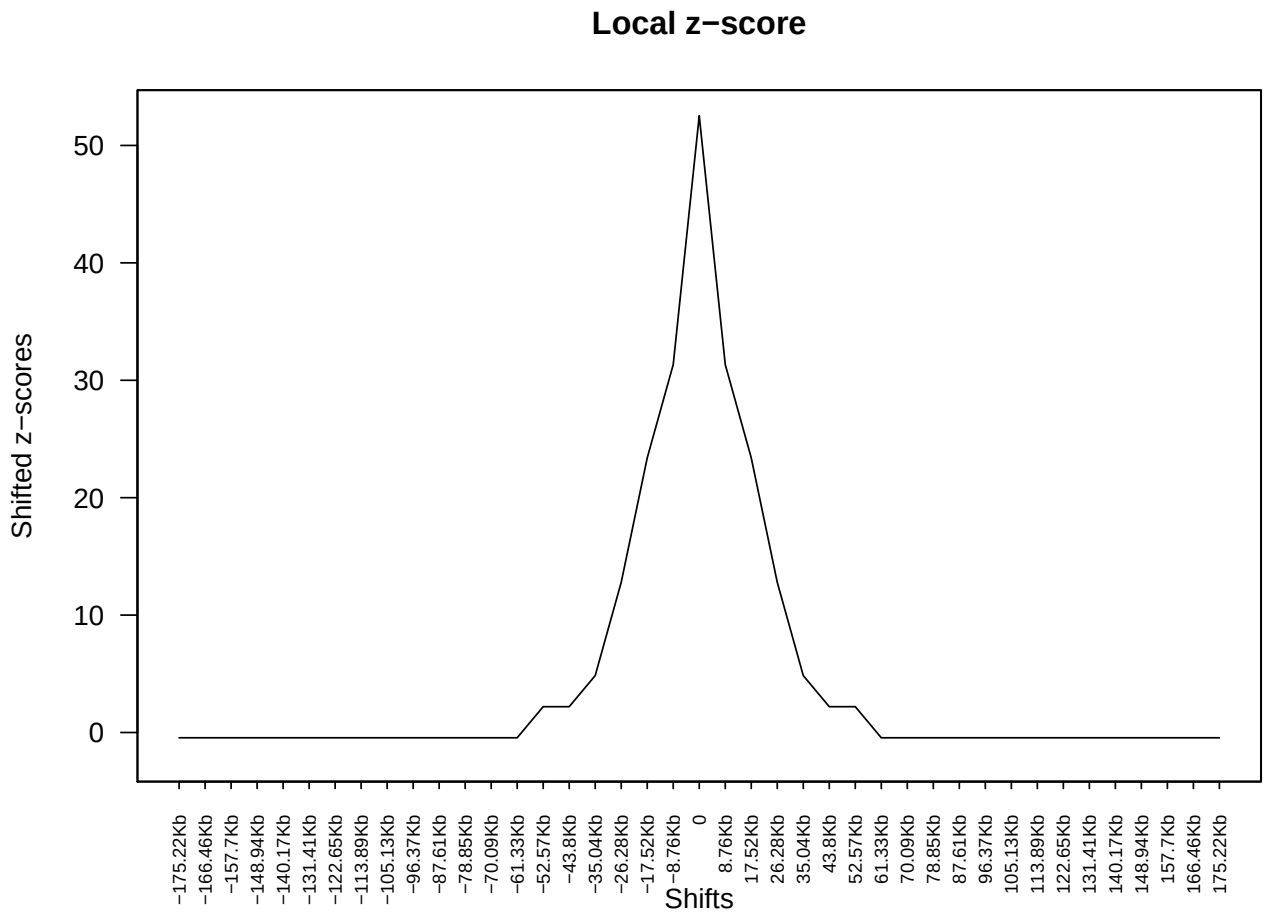


In this case we can see that there is no change in the z-scores if we move the regions around. This is expected, since the altered regions are much bigger than the special genes and have a size of tens of megabases and thus, the association is not highly dependant on the exact position of the regions.

On the contrary, we can create an example for an association highly dependant on the exact position by testing the association of a RS with a subset of it.

```
genome <- filterChromosomes(getGenome("hg19"), keep.chr="chr1")
B <- createRandomRegions(nregions=100, length.mean=10000, length.sd=20000, genome=genome,
                         non.overlapping=FALSE)
A <- B[sample(20)]

pt <- overlapPermTest(A=A, B=B, ntimes=100, genome=genome, non.overlapping=FALSE)
pt

## P-value: 0.0099009900990099
```

```
## Z-score: 52.5263
## Number of iterations: 100
## Alternative: greater
## Evaluation of the original region set: 20
## Evaluation function: numOverlaps
## Randomization function: randomizeRegions

lz <- localZScore(A=A, B=B, pt=pt, window=10*mean(width(A)), step=mean(width(A))/2 )
plot(lz)
```



In this case we can see how as soon as we displace the regions a bit away from their original position, the z-score drops almost to 0.

4 Region Sets

In regioneR, a genomic region is a part of a genome and is defined by three parameters: chromosome, start and end, where start is less than or equal to end and both are inside the limits of the chromosome. This definition includes from single bases (start equal to end) to whole chromosomes. In addition, genomic regions can have associated data or annotations which will usually depend on the nature of the data. A region set, then, is defined as a group of one or more genomic regions and is referred to as RS along this document.

Internally, regioneR uses *GRanges* to represent a RS, and all functions returning an RS return a *GRanges* object. However, regioneR accepts different data formats as input thanks to its `toGRanges` function. In particular, it accepts:

- **GRanges:** A *GRanges* object from the *GenomicRanges* Bioconductor package.
- **Data Frames:** A *data.frame* or any other class inheriting from it with a "BED-like" structure with the first three columns representing chromosome, start and end. There are a few restrictions regarding the naming of the columns in the data frame which are detailed in the *toGRanges* function help page.
- **A file:** a file name or connection to a file in any of the formats accepted by *rtracklayer*'s import function (bed, gff, wigg, ...)

Two utility functions are provided in regioneR to manage the transformation to and from *GRanges*: *toGRanges* and *toDataframe*.

- **toGRanges** accepts a RS in any of the supported formats and creates a *GRanges* object from it
- **toDataframe** accepts a *GRanges* object and transforms it into a *data.frame*

For example, we can create a *GRanges* from a *data.frame* representing a RS with 3 regions in chromosome 1 with 2 additional values (x and y) using *toGRanges*:

```
A <- data.frame(chr=1, start=c(1,15,24), end=c(10,20,30),  x=c(1,2,3), y=c("a","b","c"))
B <- toGRanges(A)
B

## GRanges object with 3 ranges and 2 metadata columns:
##       seqnames    ranges strand |         x        y
##          <Rle> <IRanges>  <Rle> | <numeric> <factor>
##   [1]        1  [ 1, 10]      * |         1        a
##   [2]        1  [15, 20]      * |         2        b
##   [3]        1  [24, 30]      * |         3        c
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

and we can get back to a *data.frame* structure using the *toDataframe* function.

```
toDataframe(B)
```

```
##   chr start end x y
## 1   1     1  10 1 a
## 2   1    15  20 2 b
## 3   1    24  30 3 c
```

5 Genomes and Masks

Most functions in regioneR are prepared to work with genomes and masks. The genome represents the complete set of chromosomes available with their length and the mask the set of genomic regions "not available" to work with, for example centromeres, highly repetitive regions, etc... The perfect mask is application specific and might have an important impact on the final results.

regioneR provides a function to get genomes and masks, `getGenomeAndMask`, but usually the user won't need to use it directly. In regioneR, all functions accepting a genome and a mask as arguments use this function internally and so, accept any genome and mask specification accepted by these function.

5.1 Genomes

In regioneR a genome is simply a list of chromosomes and their length. Therefore, it is possible to specify a genome by giving a *data.frame* with 2 columns (*chr* and *length*) or 3 columns (*chr*, *start* and *end*) or a *GRanges* object with one region per chromosome. However, when working with a standard chromosome available as a *BSgenome*, it is possible and much more convenient to rely on the automatic loading of the genome (given that is installed in the system). By specifying the *BSgenome* name of the genome -e.g. hg19, mm9, dm2 ...- regioneR will fetch the correct genome information.

5.2 Masks

As with genomes, it is possible to specify a mask as a *data.frame* or *GRanges* object with the "forbidden" regions. However, it is also possible and more convenient to use the automatic retrieving of the default mask from *BSgenome* if it's suitable for the application. Again, to automatically retrieve the mask, the masked version of the *BSgenome* must be installed.

5.3 How to retrieve a genome and mask

The *getGenomeAndMask* accepts two parameters, a genome and an optional mask. In general, the genome will be a genome assembly name ('hg19', 'mm10', etc...) and the function will check if there is a genome with that name installed and return it. If none is available, it will prompt the user to install it. If the masked version of the genome is available, that is the one that will be used. If a genome is already available in the form of a *data.frame* or *GRanges* object (or any other valid RS format), the function will return it as a *GRanges*.

The second parameter, *mask*, can be used to specify a mask or to explicitly request no to retrieve one. In particular, if *mask* is NULL (it's default value), it will try to retrieve a mask from the *BSGenome* package, if *mask* is NA it will return an empty mask and if *mask* is anything accepted by *toGRanges* it will return it as the mask.

The function returns a *list* with a genome and a mask elements, both of them *GRanges* objects.

5.4 Filtering Chromosomes

The latest versions of some genomes include a number of additional "chromosomes" representing different alternative assemblies and patches for specific complex regions. While these additional chromosomes are useful in some cases, in others they interfere with the analysis, and so it's best to remove them from the genome. To do that and to generally filter any subset of chromosomes, regioneR includes the *filterChomosomes* function. It take a RS and returns a filtered version of it containing only the regions in the specified chromosomes. To use it one need to specify the RS, the organism and the type of chromosome to keep (autosomal, canonical, ...). In addition it is possible to specify a custom list of chromosome names to keep.

The available chromosome types are dependant on the organism, and the *listChrTypes* function can be used to see what organisms and chromosome subsets are available.

```
human.genome <- getGenomeAndMask("hg19")$genome
human.canonical <- filterChromosomes(human.genome, organism="hg")
listChrTypes()

## Homo sapiens (hg): autosomal, canonical
## Mus musculus (mm): autosomal, canonical
## Bos taurus (bosTau): autosomal, canonical
## Caenorhabditis elegans (ce): autosomal, canonical
## Danio rerio (danRer): canonical
## Macaca mulata (rheMac): autosomal, canonical
## Rattus norvegicus (rn): autosomal, canonical
## Saccharomyces cerevisiae (sacCer): autosomal, canonical
```

```
## Drosophila melanogaster (dm): autosomal, canonical
## Pan troglodytes (panTro): autosomal, canonical

human.autosomal <- filterChromosomes(human.genome, organism="hg", chr.type="autosomal")
human.123 <- filterChromosomes(human.genome, keep.chr=c("chr1", "chr2", "chr3"))
```

6 Region Set Helper Functions

regioneR includes a set of helper functions based on the *GenomicRanges* infrastructure to manage and manipulate RS: *joinRegions*, *commonRegions*, *mergeRegions*, *overlapRegions*, etc... All these functions share a standard and simple signature, with one or two RS in any of the accepted formats and almost all of them (except *splitRegions* and *overlapRegions*) return a *GRanges* object.

6.1 Functions operating on a single RS

- **extendRegions:** This function takes a RS and two integers and extends (enlarges) the regions by the specified amount. If a genome is given, it takes into account the chromosome lengths and does not extend the regions over the chromosome limits. It can take negative extension values to shrink the regions instead of enlarging them.
- **joinRegions:** This function takes a RS and one integer, *min.dist*, and joins the regions that are less than *min.dist* bases apart.

6.2 Functions operating on two RS

- **commonRegions:** Given two RS A and B this functions returns a *GRanges* object with their intersection, the genomic regions common to both of them.
- **mergeRegions:** Given two RS A and B this functions returns a *GRanges* object with the regions of the genome contained in either of them, it is equivalent to merging the to RS into one and then fusing the overlapping regions.
- **subtractRegions:** Given two RS A and B this functions returns a *GRanges* object with the regions in A not present in B, it is, the regions in A minus the parts overlapping those in B.
- **uniqueRegions:** Given two RS A and B this functions return a *GRanges* object with the regions of the genome covered by A or B but not both.

6.3 Other Functions: functions not returning a GRanges object

- **overlapRegions:** *overlapRegions* is one of most powerful and useful RS management functions in regioneR. It has a number of parameters and types of returned values. For a thorough description, refer to the regioneR user guide.

Given two RS A and B, *overlapRegions* identifies the regions in A overlapping a region in B, filters them and returns them. It is possible to filter by the relation between the regions (e.g. "the region from A must be included in the one from B") or by the amount of overlap (e.g. "at least 10 bases", "at least half the region in A"). The returned value might be a complete table with information, for every overlap, of the region in A, the region in B and their relation, a boolean vector indicating for every region in A whether it overlaps any region in B following the filtering criteria or simply the number of valid overlaps between the two RS. In addition, it is possible to specify if the additional data columns associated with the RS have to be kept or might just be ignored.

Example: Given the two same RS each with 10 regions in chromosome 1 we have been using so far we can get the complete table with their overlaps:

```
overlaps.df <- overlapRegions(A, B)
overlaps.df
##   chr startA endA startB endB  type
## 1   1      1   10      1   10 equal
## 2   1     15   20     15   20 equal
## 3   1     24   30     24   30 equal
```

We can filter the overlaps and get only those where a region from A contains a region from B. Additionally, get the number of bases of the region from A covered by the region from B:

```
overlaps.df <- overlapRegions(A, B, type="BinA", get.pctA=TRUE)
overlaps.df
##   chr startA endA startB endB  type pct.basesA
## 1   1      1   10      1   10 equal        100
## 2   1     15   20     15   20 equal        100
## 3   1     24   30     24   30 equal        100
```

Or get only the overlaps of at least 5 bases

```
overlaps.df <- overlapRegions(A, B, min.bases=5)
overlaps.df
##   chr startA endA startB endB  type
## 1   1      1   10      1   10 equal
## 2   1     15   20     15   20 equal
## 3   1     24   30     24   30 equal
```

or boolean vector indicating the regions in A overlapping a region in B with 5 or more bases (this is very handy to subset a RS based on how the regions overlap another RS)

```
overlaps.bool <- overlapRegions(A, B, min.bases=5, only.boolean=TRUE)
overlaps.bool
```

```
## [1] TRUE TRUE TRUE
```

or the number of regions in A overlapping a region from B with at least 5 bases

```
overlaps.int <- overlapRegions(A, B, min.bases=5, only.count=TRUE)
overlaps.int
## [1] 3
```