Some Basic Analysis of ChIP-Seq Data

July 23, 2010

Our goal is to describe the use of Bioconductor software to perform some basic tasks in the analysis of ChIP-Seq data. We will use several functions in the as-yet-unreleased chipseq package, which provides convenient interfaces to other powerful packages such as ShortRead and IRanges. We will also use the lattice and rtracklayer packages for visualization.

```r
> library(chipseq)
> library(GenomicFeatures)
> library(lattice)
>
```

Example data

The `cstest` data set is included in the chipseq package to help demonstrate its capabilities. The dataset contains data for three chromosomes from Solexa lanes, one from a CTCF mouse ChIP-Seq, and one from a GFP mouse ChIP-Seq. The raw reads were aligned to the reference genome (mouse in this case) using an external program (MAQ), and the results read in using the readAligned function in the ShortRead, in conjunction with a filter produced by the chipseqFilter function. This step filtered the reads to remove duplicates, to restrict mappings to the canonical, autosomal chromosomes and ensure that only a single read maps to a given position. A quality score cutoff was also applied. The remaining data were reduced to a set of aligned intervals (including orientation). This saves a great deal of memory, as the sequences, which are unnecessary, are discarded. Finally, we subset the data for chr10 to chr12, simply for convenience in this vignette.

We outline this process with this unevaluated code block:

```r
> qa_list <- lapply(sampleFiles, qa)
> report(do.call(rbind, qa_list))
> ## spend some time evaluating the QA report, then proceede
> filter <- compose(chipseqFilter(), alignQualityFilter(15))
> cstest <- seqapply(sampleFiles, function(file) {
+ as(readAligned(file, filter), "GRanges")
+ })
> cstest <- cstest[seqnames(cstest) %in% c("chr10", "chr11", "chr12")]
```

The above step has been performed in advance, and the output has been included as a dataset in this package. We load it now:

```r
> data(cstest)
> cstest
```
GRangesList object of length 2:

$cctcf

GRanges object with 450096 ranges and 0 metadata columns:

<table>
<thead>
<tr>
<th>seqnames</th>
<th>ranges</th>
<th>strand</th>
</tr>
</thead>
<tbody>
<tr>
<td>chr10</td>
<td>[3012936, 3012959]</td>
<td>+</td>
</tr>
<tr>
<td>chr10</td>
<td>[3012941, 3012964]</td>
<td>+</td>
</tr>
<tr>
<td>chr10</td>
<td>[3012944, 3012967]</td>
<td>+</td>
</tr>
<tr>
<td>chr10</td>
<td>[3012955, 3012978]</td>
<td>+</td>
</tr>
<tr>
<td>chr10</td>
<td>[3012963, 3012986]</td>
<td>+</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>chr12</td>
<td>[121239376, 121239399]</td>
<td>-</td>
</tr>
<tr>
<td>chr12</td>
<td>[121245849, 121245872]</td>
<td>-</td>
</tr>
<tr>
<td>chr12</td>
<td>[121245895, 121245918]</td>
<td>-</td>
</tr>
<tr>
<td>chr12</td>
<td>[121246344, 121246367]</td>
<td>-</td>
</tr>
<tr>
<td>chr12</td>
<td>[121253499, 121253522]</td>
<td>-</td>
</tr>
</tbody>
</table>

seqinfo: 35 sequences from an unspecified genome

cstest is an object of class GRangesList, and has a list-like structure, each component representing the alignments from one lane, as a GRanges object of stranded intervals.

> cstest$cctcf

GRanges object with 450096 ranges and 0 metadata columns:

<table>
<thead>
<tr>
<th>seqnames</th>
<th>ranges</th>
<th>strand</th>
</tr>
</thead>
<tbody>
<tr>
<td>chr10</td>
<td>[3012936, 3012959]</td>
<td>+</td>
</tr>
<tr>
<td>chr10</td>
<td>[3012941, 3012964]</td>
<td>+</td>
</tr>
<tr>
<td>chr10</td>
<td>[3012944, 3012967]</td>
<td>+</td>
</tr>
<tr>
<td>chr10</td>
<td>[3012955, 3012978]</td>
<td>+</td>
</tr>
<tr>
<td>chr10</td>
<td>[3012963, 3012986]</td>
<td>+</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>chr12</td>
<td>[121239376, 121239399]</td>
<td>-</td>
</tr>
<tr>
<td>chr12</td>
<td>[121245849, 121245872]</td>
<td>-</td>
</tr>
<tr>
<td>chr12</td>
<td>[121245895, 121245918]</td>
<td>-</td>
</tr>
<tr>
<td>chr12</td>
<td>[121246344, 121246367]</td>
<td>-</td>
</tr>
<tr>
<td>chr12</td>
<td>[121253499, 121253522]</td>
<td>-</td>
</tr>
</tbody>
</table>

seqinfo: 35 sequences from an unspecified genome

Extending reads

Solexa gives us the first few (24 in this example) bases of each fragment it sequences, but the actual fragment is longer. By design, the sites of interest (transcription factor binding sites) should be somewhere in the fragment, but not necessarily in its initial part. Although the actual lengths of
fragments vary, extending the alignment of the short read by a fixed amount in the appropriate direction, depending on whether the alignment was to the positive or negative strand, makes it more likely that we cover the actual site of interest.

It is possible to estimate the fragment length, through a variety of methods. There are several implemented by the `estimate.mean.fraglen` function. Generally, this only needs to be done for one sample from each experimental protocol. Here, we use SSISR, the default method:

```r
> fraglen <- estimate.mean.fraglen(cstest$ctcf)
> fraglen[!is.na(fraglen)]
chr10  chr11  chr12
179.6794 172.4884 181.6732
```

Given the suggestion of 190 nucleotides, we extend all reads to be 200 bases long. This is done using the `resize` function, which considers the strand to determine the direction of extension:

```r
> ctcf.ext <- resize(cstest$ctcf, width = 200)
> ctcf.ext
GRanges object with 450096 ranges and 0 metadata columns:
  seqnames ranges strand
  <Rle> <IRanges> <Rle>
[1] chr10 [3012936, 3013135] +
[2] chr10 [3012941, 3013140] +
[3] chr10 [3012944, 3013143] +
[4] chr10 [3012955, 3013154] +
[5] chr10 [3012963, 3013162] +
... ... ...
[450092] chr12 [121239200, 121239399] -
[450093] chr12 [121245673, 121245872] -
[450094] chr12 [121245719, 121245918] -
[450095] chr12 [121246168, 121246367] -
[450096] chr12 [121253323, 121253522] -
```

seqinfo: 35 sequences from an unspecified genome

We now have intervals for the CTCF lane that represent the original fragments that were precipitated.

**Coverage, islands, and depth**

A useful summary of this information is the `coverage`, that is, how many times each base in the genome was covered by one of these intervals.

```r
> cov.ctcf <- coverage(ctcf.ext)
> cov.ctcf
RleList of length 35
$chr1
integer-Rle of length 197195432 with 1 run
  Lengths: 197195432
```
Values : 0

$chr2
integer-Rle of length 181748087 with 1 run
  Lengths: 181748087
  Values : 0

$chr3
integer-Rle of length 159599783 with 1 run
  Lengths: 159599783
  Values : 0

$chr4
integer-Rle of length 155630120 with 1 run
  Lengths: 155630120
  Values : 0

$chr5
integer-Rle of length 152537259 with 1 run
  Lengths: 152537259
  Values : 0

...<30 more elements>

For efficiency, the result is stored in a run-length encoded form.
The regions of interest are contiguous segments of non-zero coverage, also known as *islands*.

```r
> islands <- slice(cov.ctcf, lower = 1)
> islands
RleViewsList of length 35
names(35): chr1 chr2 chr3 chr4 ... chrX_random chrY_random chrUn_random

For each island, we can compute the number of reads in the island, and the maximum coverage depth within that island.

```r
> viewSums(islands)
IntegerList of length 35
[["chr1"]]
integer(0)
[["chr2"]]
integer(0)
[["chr3"]]
integer(0)
[["chr4"]]
integer(0)
[["chr5"]]
integer(0)
[["chr6"]]
integer(0)
[["chr7"]]
integer(0)
[["chr8"]]
integer(0)
[["chr9"]]
integer(0)
[["chr10"]]
2400 200 200 200 200 200 600 ... 200 200 400 200 200 200 200 200 200 ...

<25 more elements>
```r

```r
> viewMaxs(islands)
IntegerList of length 35
[["chr1"]]
integer(0)
[["chr2"]]
integer(0)
[["chr3"]]
integer(0)
[["chr4"]]
integer(0)
[["chr5"]]
integer(0)
[["chr6"]]
integer(0)
[["chr7"]]
integer(0)
[["chr8"]]
integer(0)
[["chr9"]]
integer(0)
[["chr10"]]
11 1 1 1 1 1 3 1 1 1 1 1 2 1 ... 1 2 1 1 1 1 3 1 1 1 2 1 1 1 1 ...

<25 more elements>
```r

```r
> nread.tab <- table(viewSums(islands) / 200)
> depth.tab <- table(viewMaxs(islands))
> nread.tab[,1:10]

     1  2  3  4  5  6  7  8  9 10
chr1 0 0 0 0 0 0 0 0 0 0
chr2 0 0 0 0 0 0 0 0 0 0
```
 chr3 0 0 0 0 0 0 0 0 0 0
 chr4 0 0 0 0 0 0 0 0 0 0
 chr5 0 0 0 0 0 0 0 0 0 0
 chr6 0 0 0 0 0 0 0 0 0 0
 chr7 0 0 0 0 0 0 0 0 0 0
 chr8 0 0 0 0 0 0 0 0 0 0
 chr9 0 0 0 0 0 0 0 0 0 0
 chr10 68101 13352 3019 924 418 246 191 123 133 100
 chr11 71603 15993 4334 1410 619 338 245 199 180 151
 chr12 59141 11279 2613 816 344 175 140 119 84 71
 chr13 0 0 0 0 0 0 0 0 0 0
 chr14 0 0 0 0 0 0 0 0 0 0
 chr15 0 0 0 0 0 0 0 0 0 0
 chr16 0 0 0 0 0 0 0 0 0 0
 chr17 0 0 0 0 0 0 0 0 0 0
 chr18 0 0 0 0 0 0 0 0 0 0
 chr19 0 0 0 0 0 0 0 0 0 0
 chrX 0 0 0 0 0 0 0 0 0 0
 chrY 0 0 0 0 0 0 0 0 0 0
 chrM 0 0 0 0 0 0 0 0 0 0
 chr1_random 0 0 0 0 0 0 0 0 0 0
 chr3_random 0 0 0 0 0 0 0 0 0 0
 chr4_random 0 0 0 0 0 0 0 0 0 0
 chr5_random 0 0 0 0 0 0 0 0 0 0
 chr7_random 0 0 0 0 0 0 0 0 0 0
 chr8_random 0 0 0 0 0 0 0 0 0 0
 chr9_random 0 0 0 0 0 0 0 0 0 0
 chr13_random 0 0 0 0 0 0 0 0 0 0
 chr16_random 0 0 0 0 0 0 0 0 0 0
 chr17_random 0 0 0 0 0 0 0 0 0 0
 chrX_random 0 0 0 0 0 0 0 0 0 0
 chrY_random 0 0 0 0 0 0 0 0 0 0
 chrUn_random 0 0 0 0 0 0 0 0 0 0

> depth.tab[,1:10]

   1  2  3  4  5  6  7  8  9 10
chr1 0 0 0 0 0 0 0 0 0 0
chr2 0 0 0 0 0 0 0 0 0 0
chr3 0 0 0 0 0 0 0 0 0 0
chr4 0 0 0 0 0 0 0 0 0 0
chr5 0 0 0 0 0 0 0 0 0 0
chr6 0 0 0 0 0 0 0 0 0 0
chr7 0 0 0 0 0 0 0 0 0 0
chr8 0 0 0 0 0 0 0 0 0 0
chr9 0 0 0 0 0 0 0 0 0 0
chr10 68149 14748 2386 547 256 180 150 129 120 101
chr11 71677 17945 3527 862 362 268 205 179 181 130
chr12 59181 12441 2078 482 191 131 131 108 95 77

6
<table>
<thead>
<tr>
<th>Chr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>chr13</td>
<td>0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>chr14</td>
<td>0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>chr15</td>
<td>0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>chr16</td>
<td>0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>chr17</td>
<td>0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>chr18</td>
<td>0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>chr19</td>
<td>0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>chrX</td>
<td>0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>chrY</td>
<td>0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>chrM</td>
<td>0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>chr1_random</td>
<td>0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>chr3_random</td>
<td>0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>chr4_random</td>
<td>0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>chr5_random</td>
<td>0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>chr7_random</td>
<td>0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>chr8_random</td>
<td>0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>chr9_random</td>
<td>0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>chr13_random</td>
<td>0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>chr16_random</td>
<td>0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>chr17_random</td>
<td>0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>chrX_random</td>
<td>0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>chrY_random</td>
<td>0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>chrUn_random</td>
<td>0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>
Processing multiple lanes

Although data from one lane is often a natural analytical unit, we typically want to apply any procedure to all lanes. A function that is useful for this purpose is `seqapply`, which applies a function to a `Sequence` and returns the result as another `Sequence`, if possible. In this case, our input `Sequence` is `GRangesList`, and our expected output is a `DataFrameList`. Here is a simple summary function that computes the frequency distribution of the number of reads.

```r
islandReadSummary <- function(x) {
  g <- resize(x, 200)
  s <- slice(coverage(g), lower = 1)
  tab <- table(viewSums(s) / 200)
  df <- DataFrame(tab)
  colnames(df) <- c("chromosome", "nread", "count")
  df$nread <- as.integer(df$nread)
  df
}
```

Applying it to our test-case, we get

```r
> head(islandReadSummary(cstest$ctcf))

DataFrame with 6 rows and 3 columns

chromosome  nread  count
<factor>  <integer> <integer>
1  chr1     1     0
2  chr2     1     0
3  chr3     1     0
4  chr4     1     0
5  chr5     1     0
6  chr6     1     0
```

We can now use it to summarize the full dataset, flattening the returned `DataFrameList` with the `stack` function.

```r
> nread.islands <- seqapply(cstest, islandReadSummary)
> nread.islands <- stack(nread.islands, "sample")
> nread.islands

DataFrame with 4025 rows and 4 columns

sample  chromosome  nread  count
<Rle>  <factor>  <integer> <integer>
1  cctcf  chr1     1     0
2  cctcf  chr2     1     0
3  cctcf  chr3     1     0
4  cctcf  chr4     1     0
5  cctcf  chr5     1     0
...     ...      ...    ...
4021    gfp  chr16_random  34    0

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4022</td>
<td>gfp</td>
<td>chr17_random</td>
<td>34</td>
</tr>
<tr>
<td>4023</td>
<td>gfp</td>
<td>chrX_random</td>
<td>34</td>
</tr>
<tr>
<td>4024</td>
<td>gfp</td>
<td>chrY_random</td>
<td>34</td>
</tr>
<tr>
<td>4025</td>
<td>gfp</td>
<td>chrUn_random</td>
<td>34</td>
</tr>
</tbody>
</table>
> xyplot(log(count) ~ nread | sample, as.data.frame(nread.islands),
  +      subset = (chromosome == "chr10" & nread <= 40),
  +      layout = c(1, 2), pch = 16, type = c("p", "g"))
If reads were sampled randomly from the genome, then the null distribution number of reads per island would have a geometric distribution; that is,

\[ P(X = k) = p^{k-1}(1 - p) \]

In other words, \( \log P(X = k) \) is linear in \( k \). Although our samples are not random, the islands with just one or two reads may be representative of the null distribution.

```r
> xyplot(log(count) ~ nread | sample, as.data.frame(nread.islands),
+     subset = (chromosome == "chr10" & nread <= 40),
+     layout = c(1, 2), pch = 16, type = c("p", "g"),
+     panel = function(x, y, ...) {
+       panel.lmline(x[1:2], y[1:2], col = "black")
+       panel.xyplot(x, y, ...)
+     })
```
We can create a similar plot of the distribution of depths.

```r
> islandDepthSummary <- function(x)
+ {
+   g <- resize(x, 200)
+   s <- slice(coverage(g), lower = 1)
+   tab <- table(viewMaxs(s) / 200)
+   df <- DataFrame(tab)
+   colnames(df) <- c("chromosome", "depth", "count")
+   df$depth <- as.integer(df$depth)
+   df
+ }
> depth.islands <- seqapply(cstest, islandDepthSummary)
> depth.islands <- stack(depth.islands, "sample")
> xyplot(log(count) ~ depth | sample, as.data.frame(depth.islands),
+   subset = (chromosome == "chr10" & depth <= 20),
+   layout = c(1, 2), pch = 16, type = c("p", "g"),
+   panel = function(x, y, ...) {
+     lambda <- 2 * exp(y[2]) / exp(y[1])
+     null.est <- function(xx) {
+       xx * log(lambda) - lambda - lgamma(xx + 1)
+     }
+     log.N.hat <- null.est(1) - y[1]
+     panel.lines(1:10, -log.N.hat + null.est(1:10), col = "black")
+     panel.xyplot(x, y, ...)
+   })
> ## depth.islands <- summarizeReads(cstest, summary.fun = islandDepthSummary)
> 
> ```
The above plot is very useful for detecting peaks, discussed in the next section. As a convenience, it can be created for the coverage over all chromosomes for a single sample by calling the `islandDepthPlot` function:

```r
> islandDepthPlot(cov.ctcf)
```
**Peaks**

To obtain a set of putative binding sites, i.e., peaks, we need to find those regions that are significantly above the noise level. Using the same Poisson-based approach for estimating the noise distribution as in the plot above, the `peakCutoff` function returns a cutoff value for a specific FDR:

```r
> peakCutoff(cov.ctcf, fdr = 0.0001)
[1] 6.959837
```

Considering the above calculation of 7 at an FDR of 0.0001, and looking at the above plot, we might choose 8 as a conservative peak cutoff:

```r
> peaks.ctcf <- slice(cov.ctcf, lower = 8)
> peaks.ctcf
```

RleViewsList of length 35
names(35): chr1 chr2 chr3 chr4 ... chrX_random chrY_random chrUn_random

To summarize the peaks for exploratory analysis, we call the `peakSummary` function:

```r
> peaks <- peakSummary(peaks.ctcf)
```

The result is a `RangedData` object with two columns: the view maxs and the view sums. Beyond that, this object is often useful as a scaffold for adding additional statistics. It is meaningful to ask about the contribution of each strand to each peak, as the sequenced region of pull-down fragments would be on opposite sides of a binding site depending on which strand it matched. We can compute strand-specific coverage, and look at the individual coverages under the combined peaks as follows:

```r
> peak.depths <- viewMaxs(peaks.ctcf)
> cov.pos <- coverage(ctcf.ext[strand(ctcf.ext) == "+"])
> cov.neg <- coverage(ctcf.ext[strand(ctcf.ext) == "-"])
> peaks.pos <- Views(cov.pos, ranges(peaks.ctcf))
> peaks.neg <- Views(cov.neg, ranges(peaks.ctcf))
> wpeaks <- tail(order(peak.depths$chr10), 4)
> wpeaks
[1] 971 989 1079 922
```

Below, we plot the four highest peaks on chromosome 10.
\texttt{> coverageplot(peaks.pos$chr10[wpeaks[1]], peaks.neg$chr10[wpeaks[1]])}

\hspace{1cm}

\hspace{1cm}

\texttt{> coverageplot(peaks.pos$chr10[wpeaks[2]], peaks.neg$chr10[wpeaks[2]])}
> coverageplot(peaks.pos$chr10[wpeaks[3]], peaks.neg$chr10[wpeaks[3]])

> coverageplot(peaks.pos$chr10[wpeaks[4]], peaks.neg$chr10[wpeaks[4]])
Differential peaks

One common question is: which peaks are different in two samples? One simple strategy is the following: combine the two peak sets, and compare the two samples by calculating summary statistics for the combined peaks on top of each coverage vector.

```r
> ## find peaks for GFP control
> cov.gfp <- coverage(resize(cstest$gfp, 200))
> peaks.gfp <- slice(cov.gfp, lower = 8)
> peakSummary <- diffPeakSummary(peaks.gfp, peaks.ctcf)
> xyplot(asinh(sums2) ~ asinh(sums1) | space,
+ data = as.data.frame(peakSummary),
+ panel = function(x, y, ...) {
+   panel.xyplot(x, y, ...)
+   panel.abline(median(y - x), 1)
+ },
+ type = c("p", "g"), alpha = 0.5, aspect = "iso")
```

We use a simple cutoff to flag peaks that are different.

```r
> peakSummary <-
+ within(peakSummary,
+ {
+   diffs <- asinh(sums2) - asinh(sums1)
+   resids <- (diffs - median(diffs)) / mad(diffs)
+   up <- resids > 2
+   down <- resids < -2
+   change <- ifelse(up, "up", ifelse(down, "down", "flat"))
+ })
```

![Graph showing differential peaks](image-url)
Placing peaks in genomic context

Locations of individual peaks may be of interest. Alternatively, a global summary might look at classifying the peaks of interest in the context of genomic features such as promoters, upstream regions, etc. The `GenomicFeatures` package facilitates obtaining gene annotations from different data sources. We could download the UCSC gene predictions for the mouse genome and generate a `GRanges` object with the transcript regions (from the first to last exon, contiguous) using `makeTranscriptDbFromUCSC`; here we use a library containing a recent snapshot.

```r
> library(TxDb.Mmusculus.UCSC.mm9.knownGene)
> gregions <- transcripts(TxDb.Mmusculus.UCSC.mm9.knownGene)
> gregions
```

### GRanges object with 55419 ranges and 2 metadata columns:

<table>
<thead>
<tr>
<th>seqnames</th>
<th>ranges</th>
<th>strand</th>
<th>tx_id</th>
<th>tx_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Rle&gt;</td>
<td>&lt;IRanges&gt;</td>
<td>&lt;Rle&gt;</td>
<td>&lt;integer&gt;</td>
<td>&lt;character&gt;</td>
</tr>
<tr>
<td>[1] chr1</td>
<td>[4797974, 4832908]</td>
<td>+</td>
<td>1</td>
<td>uc007afg.1</td>
</tr>
<tr>
<td>[2] chr1</td>
<td>[4797974, 4836816]</td>
<td>+</td>
<td>2</td>
<td>uc007afh.1</td>
</tr>
<tr>
<td>[3] chr1</td>
<td>[4847775, 4887990]</td>
<td>+</td>
<td>3</td>
<td>uc007afi.2</td>
</tr>
<tr>
<td>[4] chr1</td>
<td>[4847775, 4887990]</td>
<td>+</td>
<td>4</td>
<td>uc011wht.1</td>
</tr>
<tr>
<td>[5] chr1</td>
<td>[4848409, 4887990]</td>
<td>+</td>
<td>5</td>
<td>uc011whu.1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>[55415] chrY_random</td>
<td>[48388361, 48390745]</td>
<td>-</td>
<td>55343</td>
<td>uc009vif.1</td>
</tr>
<tr>
<td>[55416] chrY_random</td>
<td>[50814754, 50817087]</td>
<td>-</td>
<td>55344</td>
<td>uc009vih.1</td>
</tr>
<tr>
<td>[55417] chrY_random</td>
<td>[52063165, 52089373]</td>
<td>-</td>
<td>55345</td>
<td>uc009vij.1</td>
</tr>
<tr>
<td>[55418] chrY_random</td>
<td>[52063165, 52089373]</td>
<td>-</td>
<td>55346</td>
<td>uc009vik.1</td>
</tr>
<tr>
<td>[55419] chrY_random</td>
<td>[53190054, 53190080]</td>
<td>-</td>
<td>55347</td>
<td>uc012htg.1</td>
</tr>
<tr>
<td>-------</td>
<td>---------</td>
<td>------</td>
<td>------</td>
<td>--------------</td>
</tr>
</tbody>
</table>
| seqinfo: 35 sequences (1 circular) from mm9 genome

We can now estimate the promoter for each transcript:

```r
> promoters <- flank(gregions, 1000, both = TRUE)
```

And count the peaks that fall into a promoter:

```r
> peakSummary$inPromoter <- peakSummary %over% promoters
> xtabs(~ inPromoter + change, peakSummary)
```

<table>
<thead>
<tr>
<th>change</th>
<th>inPromoter</th>
</tr>
</thead>
<tbody>
<tr>
<td>down</td>
<td>21 5158</td>
</tr>
<tr>
<td>flat</td>
<td></td>
</tr>
<tr>
<td>TRUE</td>
<td>2 625</td>
</tr>
</tbody>
</table>

Or somewhere upstream or in a gene:

```r
> peakSummary$inUpstream <- peakSummary %over% flank(gregions, 20000)
> peakSummary$inGene <- peakSummary %over% gregions
```

```r
> sumtab <-
+ as.data.frame(rbind(total = xtabs(~ change, peakSummary),
```
Visualizing peaks in genomic context

While it is generally informative to calculate statistics incorporating the genomic context, eventually one wants a picture. The traditional genome browser view is an effective method of visually integrating multiple annotations with experimental data along the genome.

Using the rtracklayer package, we can upload our coverage and peaks for both samples to the UCSC Genome Browser:

```r
library(rtracklayer)
session <- browserSession()
genome(session) <- "mm9"
session$gfpCov <- cov.gfp
session$gfpPeaks <- peaks.gfp
session$ctcfCov <- cov.ctcf
session$ctcfPeaks <- peaks.ctcf

Once the tracks are uploaded, we can choose a region to view, such as the tallest peak on chr10 in the CTCF data:

```r
peak.ord <- order(unlist(peak.depths), decreasing=TRUE)
peak.sort <- as(peaks.ctcf, "GRanges")[peak.ord]
view <- browserView(session, peak.sort[1], full = c("gfpCov", "ctcfCov"))
```

We coerce to GRanges so that we can sort the ranges across chromosomes. By passing the full parameter to browserView we instruct UCSC to display the coverage tracks as a bar chart. Next, we might programmatically display a view for the top 5 tallest peaks:

```r
views <- browserView(session, head(peak.sort, 5), full = c("gfpCov", "ctcfCov"))
```

Version information

```r
sessionInfo()
```

R Under development (unstable) (2014-10-07 r66723)
Platform: x86_64-unknown-linux-gnu (64-bit)

locale:
[1] LC_CTYPE=en_US.UTF-8 LC_NUMERIC=C
[3] LC_TIME=en_US.UTF-8 LC_COLLATE=C
[5] LC_MONETARY=en_US.UTF-8 LC_MESSAGES=en_US.UTF-8
[7] LC_PAPER=en_US.UTF-8 LC_NAME=C
[9] LC_ADDRESS=C LC_TELEPHONE=C
attached base packages:
[1] stats4 parallel stats graphics grDevices utils datasets
[8] methods base

other attached packages:
[2] lattice_0.20-29
[4] AnnotationDbi_1.29.0
[5] Biobase_2.27.0
[6] chipseq_1.17.0
[7] ShortRead_1.25.0
[8] GenomicAlignments_1.3.0
[9] Rsamtools_1.19.0
[10] BiocParallel_1.1.0
[11] BSgenome_1.35.0
[12] rtracklayer_1.27.0
[13] Biostrings_2.35.0
[14] XVector_0.7.0
[16] GenomeInfoDb_1.3.0
[17] IRanges_2.1.0
[18] S4Vectors_0.5.0
[19] BiocGenerics_0.13.0

loaded via a namespace (and not attached):
[1] BBmisc_1.7 BatchJobs_1.4 DBI_0.3.1
[4] RColorBrewer_1.0-5 Rcurl_1.95-4.3 RSQLite_0.11.4
[7] XML_3.98-1.1 base64enc_0.1-2 biomaRt_2.23.0
[10] bitops_1.0-6 brew_1.0-6 checkmate_1.4
[13] codetools_0.2-9 digest_0.6.4 fail_1.2
[16] foreach_1.4.2 grid_3.2.0 hwriter_1.3.2
[19] iterators_1.0.7 latticeExtra_0.6-26 sendmailR_1.2-1
[22] stringr_0.6.2 tools_3.2.0 zlibbioc_1.13.0