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.

```
> 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 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:

```
> qa_list <- lapply(sampleFiles, qa)
> report(do.call(rbind, qa_list))
> ## spend some time evaluating the QA report, then procede
> filter <- compose(chipseqFilter(), alignQualityFilter(15))
> cstest <- GenomicRangesList(lapply(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:

```
> data(cstest)
> cstest
```

```
GRangesList object of length 2:
$ctcf
GRanges object with 450096 ranges and 0 metadata columns:
           seqnames                 ranges strand
              <Rle>              <IRanges>  <Rle>
       [1]    chr10     [3012936, 3012959]      +
       [2]    chr10     [3012941, 3012964]      +
       [3]    chr10     [3012944, 3012967]      +
       [4]    chr10     [3012955, 3012978]      +
       [5]    chr10     [3012963, 3012986]      +
       ...      ...                    ...    ...
  [450092]    chr12 [121239376, 121239399]      -
  [450093]    chr12 [121245849, 121245872]      -
  [450094]    chr12 [121245895, 121245918]      -
  [450095]    chr12 [121246344, 121246367]      -
  [450096]    chr12 [121253499, 121253522]      -


...
<1 more element>
-------
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$ctcf
```

```
GRanges object with 450096 ranges and 0 metadata columns:
           seqnames                 ranges strand
              <Rle>              <IRanges>  <Rle>
       [1]    chr10     [3012936, 3012959]      +
       [2]    chr10     [3012941, 3012964]      +
       [3]    chr10     [3012944, 3012967]      +
       [4]    chr10     [3012955, 3012978]      +
       [5]    chr10     [3012963, 3012986]      +
       ...      ...                    ...    ...
  [450092]    chr12 [121239376, 121239399]      -
  [450093]    chr12 [121245849, 121245872]      -
  [450094]    chr12 [121245895, 121245918]      -
  [450095]    chr12 [121246344, 121246367]      -
  [450096]    chr12 [121253499, 121253522]      -
  -------
  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:

```
> fraglen <- estimate.mean.fraglen(cstest$ctcf, method="correlation")
> fraglen[!is.na(fraglen)]

chr10 chr11 chr12
  340   340   340
```

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:

```
> 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.

```
> 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*.

```
> 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.

```
> 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 200 200 200 200 200 200 600 ... 200 200 400 200 200 200 200
...
<25 more elements>

> 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 1 3 1 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>

> 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
```

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

Processing multiple lanes

Although data from one lane is often a natural analytical unit, we typically want to apply any procedure to all lanes. Here is a simple summary function that computes the frequency distribution of the number of reads.

```
> 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

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

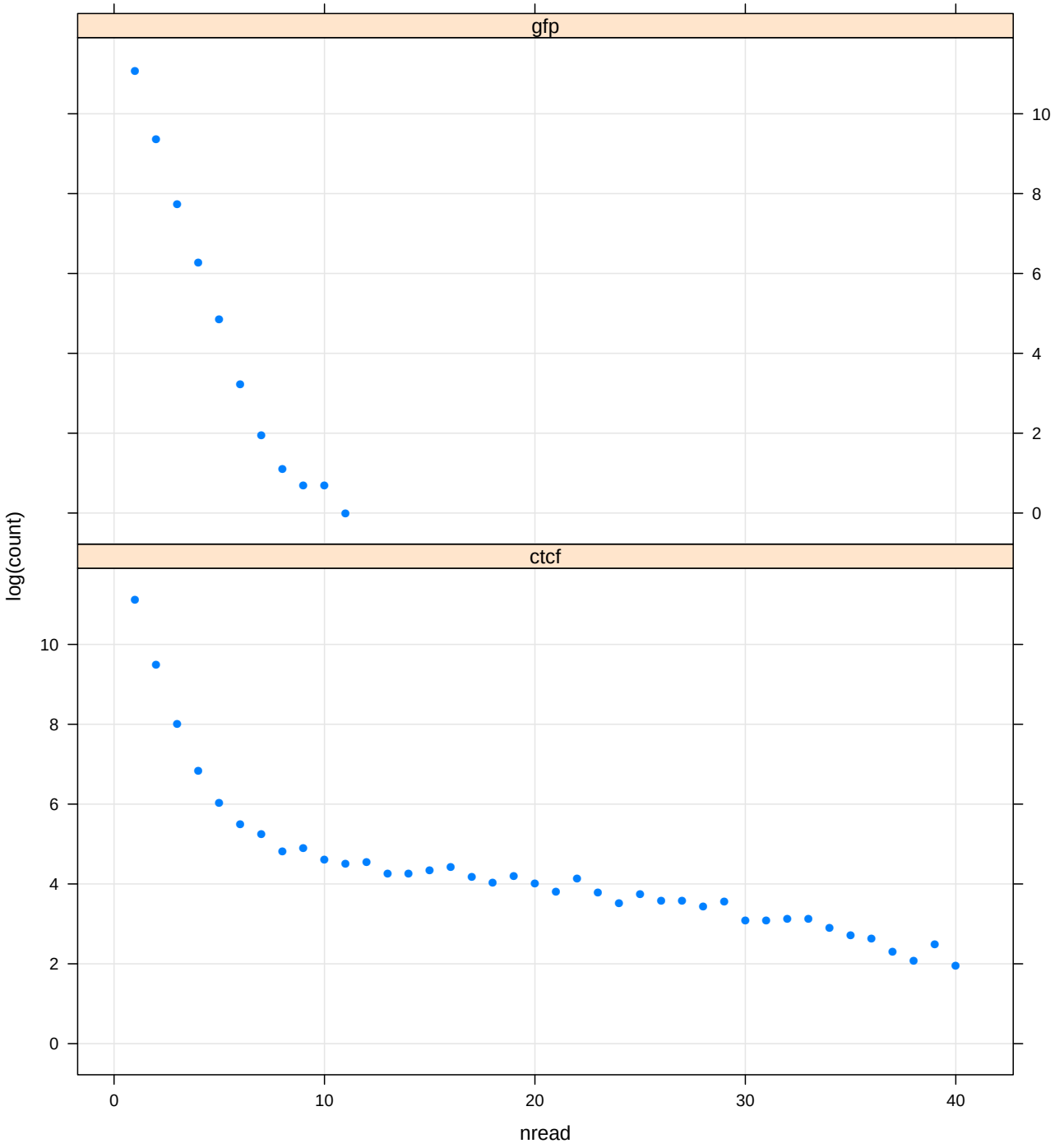
```
DataFrame with 6 rows and 3 columns
  chromosome     nread     count
    <factor> <integer> <integer>
1       chr1         1         0
2       chr2         1         0
3       chr3         1         0
4       chr4         1         0
5       chr5         1         0
6       chr6         1       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

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

```
> nread.islands <- DataFrameList(lapply(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      ctcf         chr1         1         0
2      ctcf         chr2         1         0
3      ctcf         chr3         1         0
4      ctcf         chr4         1         0
5      ctcf         chr5         1         0
...     ...          ...       ...       ...
4021    gfp chr16_random        34         0
4022    gfp chr17_random        34         0
4023    gfp  chrX_random        34         0
4024    gfp  chrY_random        34         0
4025    gfp chrUn_random        34         0
```

```
> 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"))
```

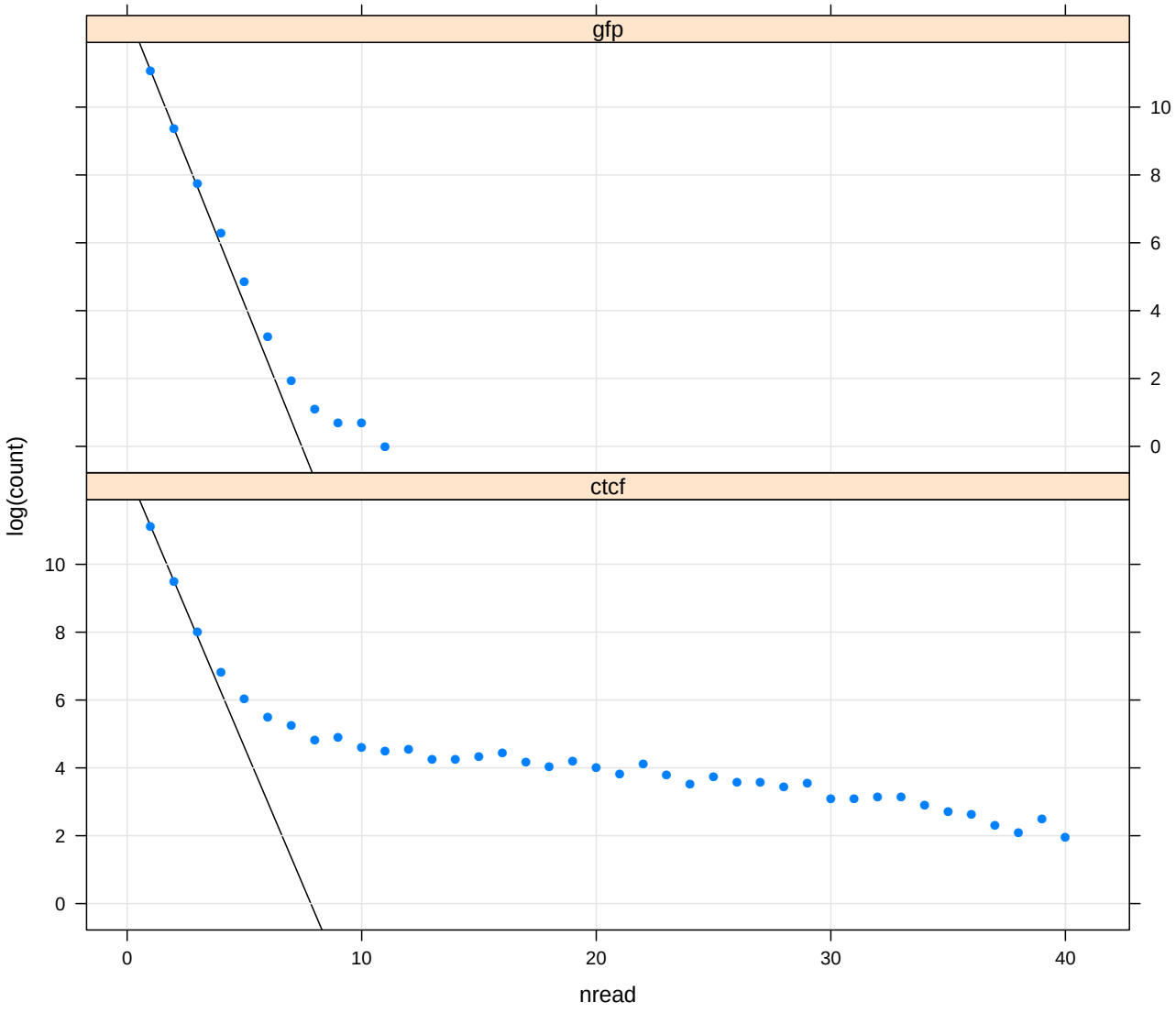


If reads were sampled randomly from the genome, then the null distribution number of reads per island would have a geometric distribution; that is,

$$P(X = k) = p^{k-1}(1 - p)$$

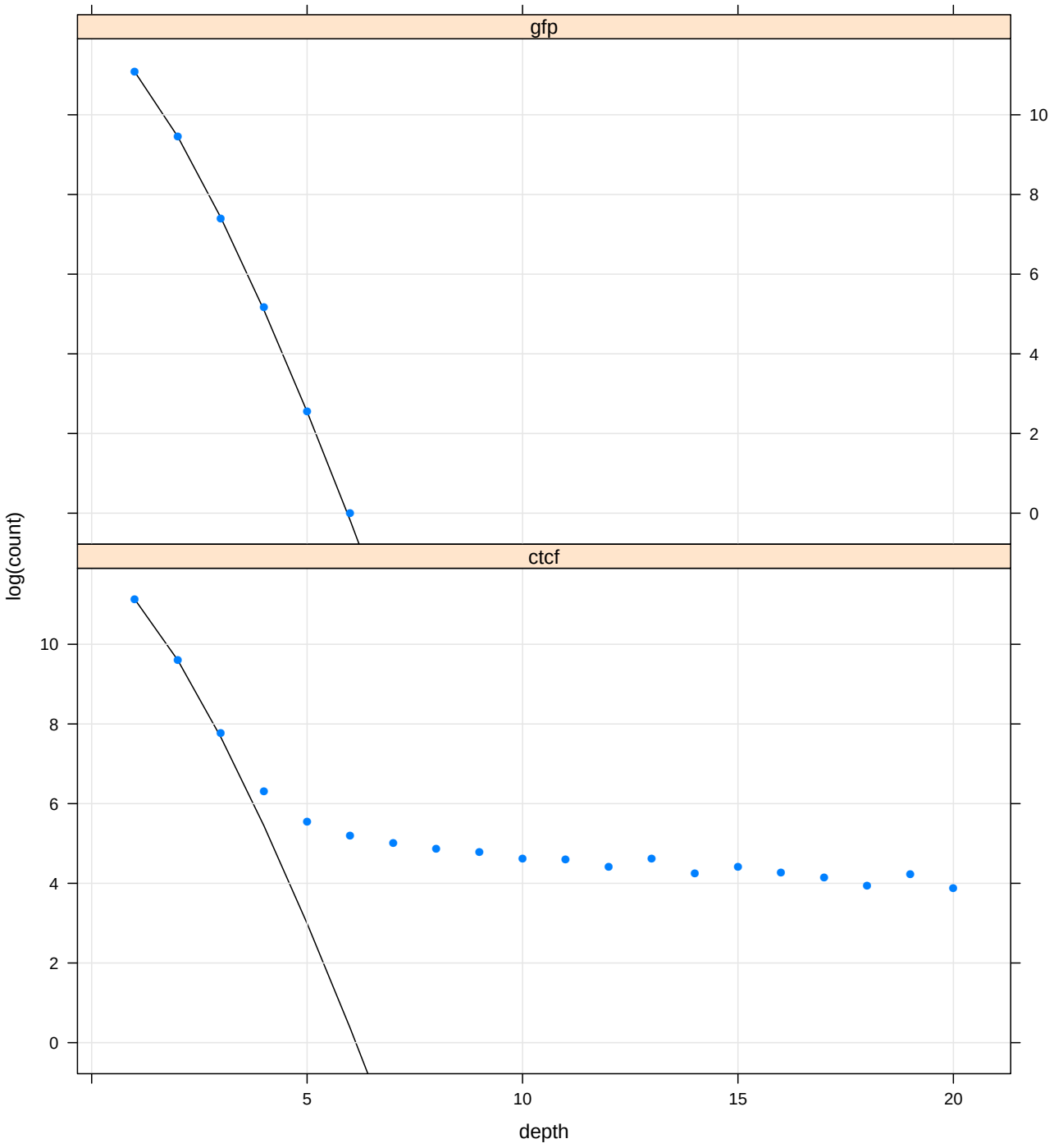
In other words, $\log P(X = k)$ is linear in k. Although our samples are not random, the islands with just one or two reads may be representative of the null distribution.

```
> 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.

```
> 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 <- DataFrameList(lapply(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:

```
> 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:

```
> 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:

```
> 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:

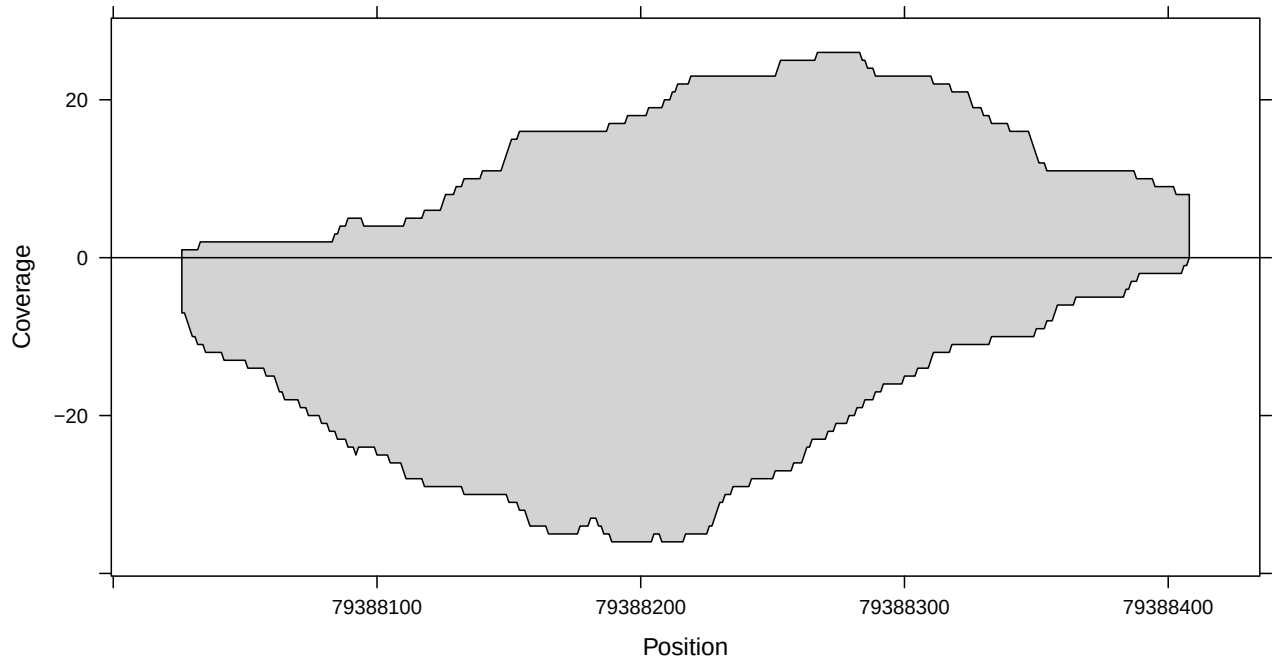
```
> peaks <- peakSummary(peaks.ctcf)
```

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:

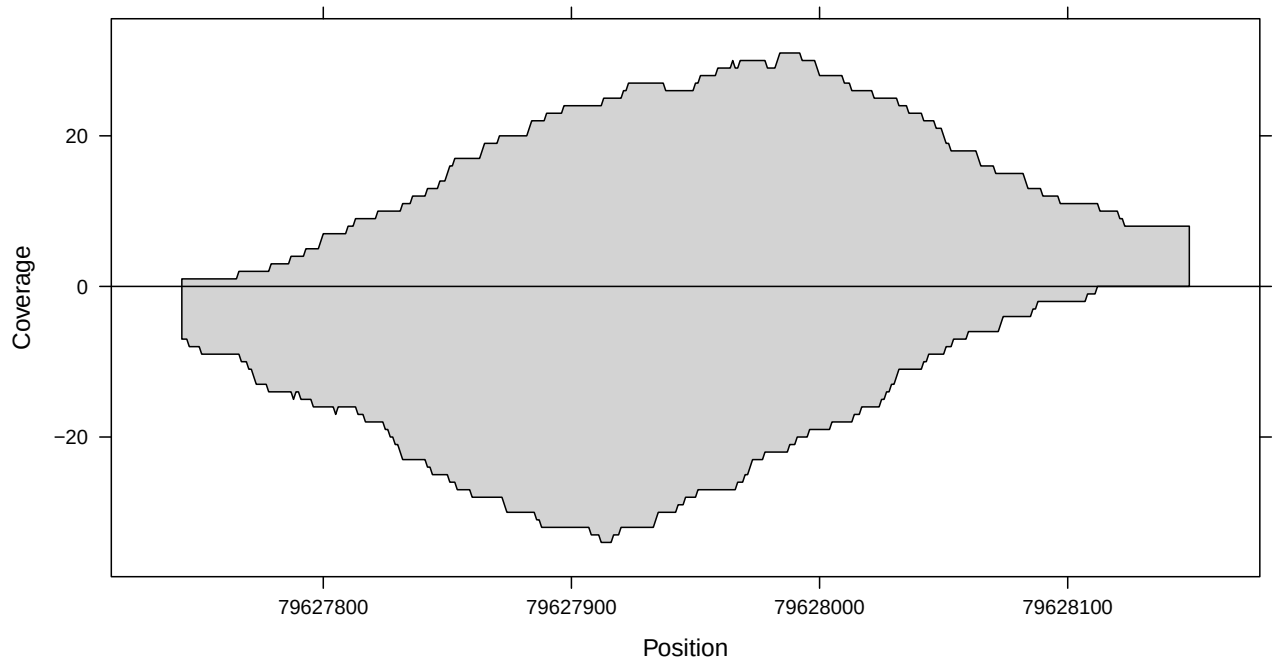
```
> peak.depths <- viewMaxs(peaks.ctcf)
> cov.pos <- coverage(ctcf.ext[strand(ctcf.ext) == "+"])
> cov.neg <- coverage(ctcf.ext[strand(ctcf.ext) == "-"])
> peaks.pos <- Views(cov.pos, ranges(peaks.ctcf))
> peaks.neg <- Views(cov.neg, ranges(peaks.ctcf))
> wpeaks <- tail(order(peak.depths$chr10), 4)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

[1]  971  989 1079  922

>
```

Below, we plot the four highest peaks on chromosome 10.

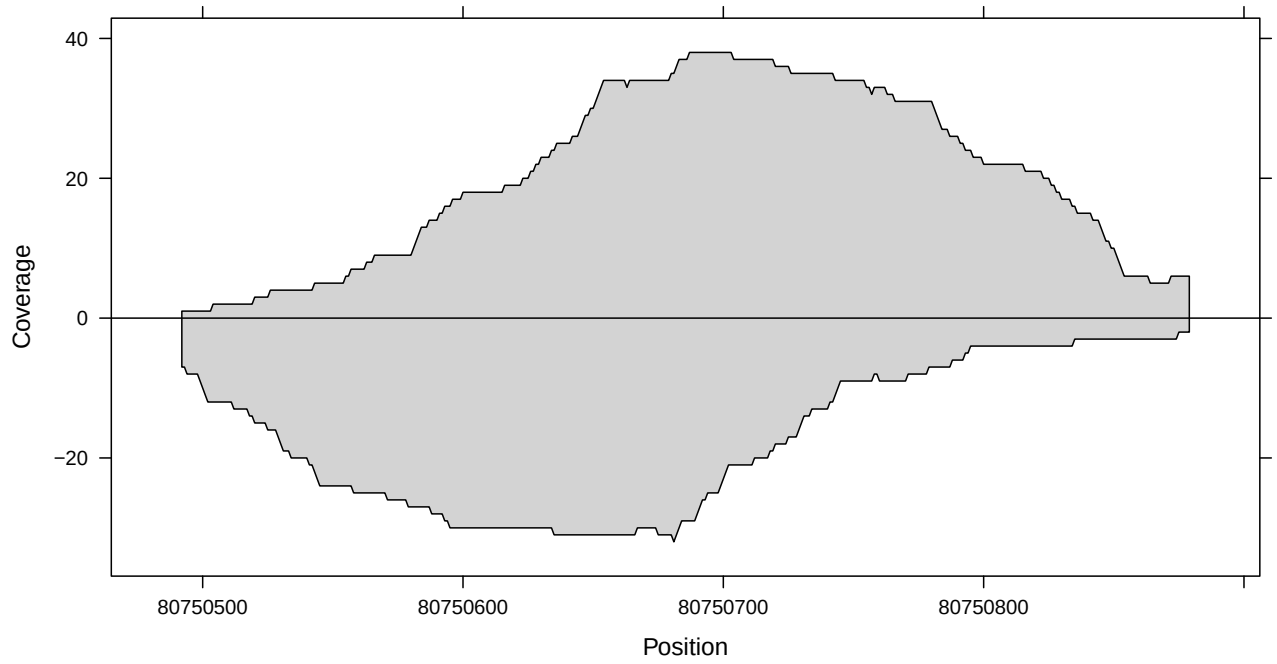
```
> coverageplot(peaks.pos$chr10[wpeaks[1]], peaks.neg$chr10[wpeaks[1]])
```



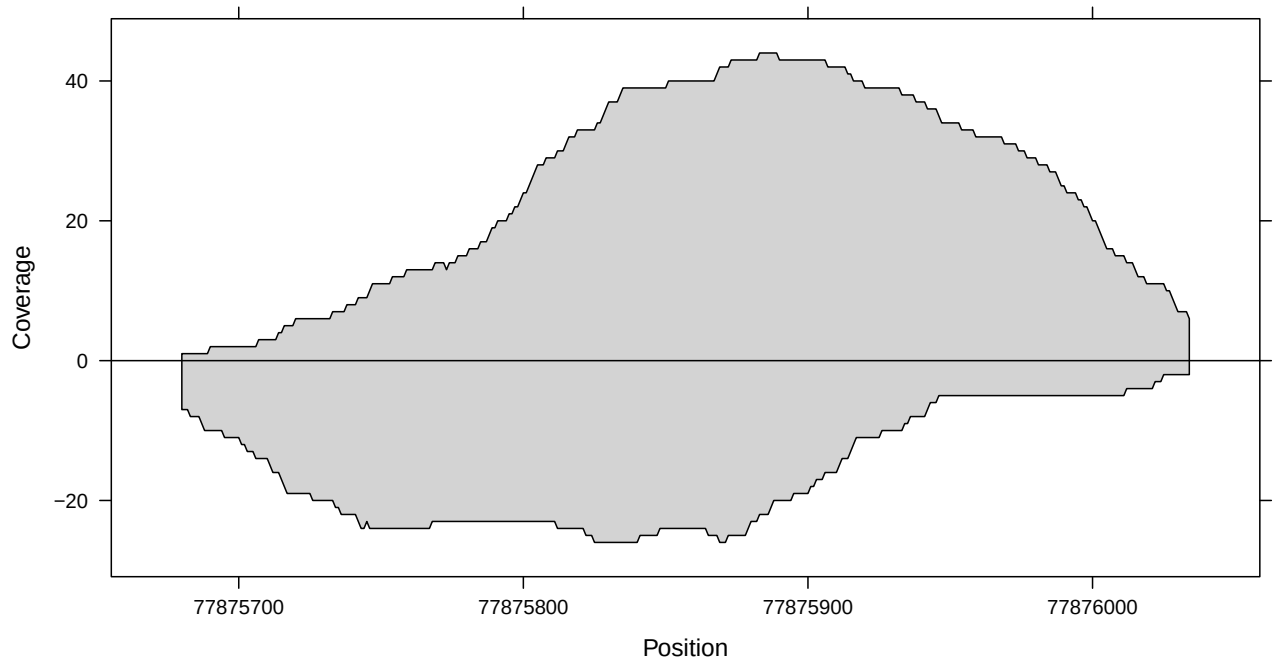

```
> coverageplot(peaks.pos$chr10[wpeaks[2]], peaks.neg$chr10[wpeaks[2]])
```



```
> coverageplot(peaks.pos$chr10[wpeaks[3]], peaks.neg$chr10[wpeaks[3]])
```



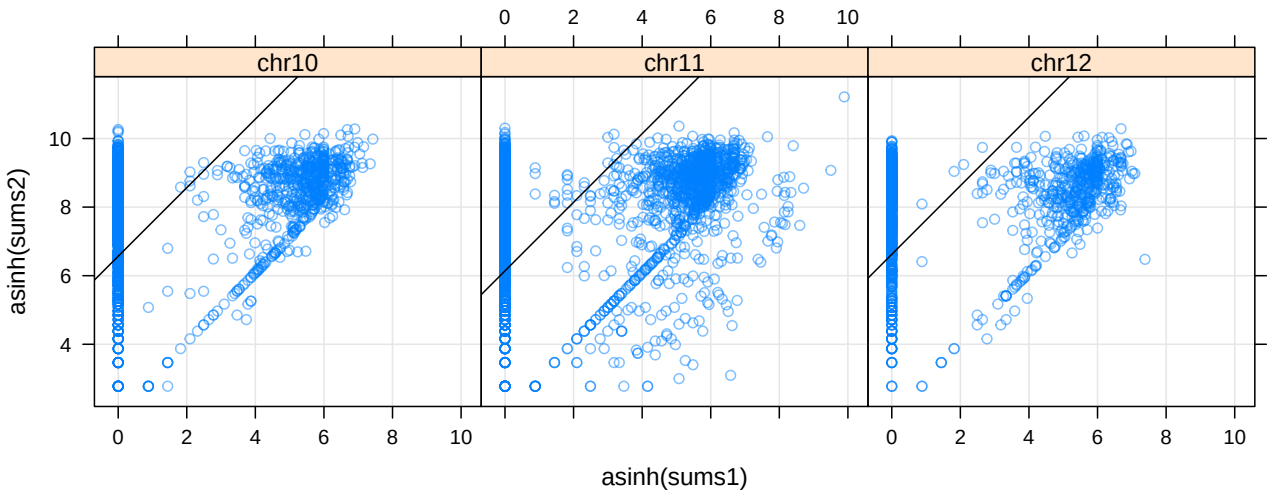

```
> coverageplot(peaks.pos$chr10[wpeaks[4]], peaks.neg$chr10[wpeaks[4]])
```



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.

```
> ## 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.

```
> 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"))
+         })
```

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 `makeTxDbFromUCSC`; here we use a library containing a recent snapshot.

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

GRanges object with 55419 ranges and 2 metadata columns:
              seqnames             ranges strand |     tx_id     tx_name
                 <Rle>          <IRanges>  <Rle> | <integer> <character>
      [1]         chr1 [4797974, 4832908]      + |         1  uc007afg.1
      [2]         chr1 [4797974, 4836816]      + |         2  uc007afh.1
      [3]         chr1 [4847775, 4887990]      + |         3  uc007afi.2
      [4]         chr1 [4847775, 4887990]      + |         4  uc011wht.1
      [5]         chr1 [4848409, 4887990]      + |         5  uc011whu.1
      ...          ...                ...    ... .       ...         ...
  [55415] chrUn_random [2204169, 2216886]      - |     55415  uc009sjw.2
  [55416] chrUn_random [2674945, 2678407]      - |     55416  uc009skb.2
  [55417] chrUn_random [2889607, 2891056]      - |     55417  uc009ske.1
  [55418] chrUn_random [3830796, 3837247]      - |     55418  uc009skg.1
  [55419] chrUn_random [4677114, 4677187]      - |     55419  uc009skh.1
  -------
  seqinfo: 35 sequences (1 circular) from mm9 genome
```

We can now estimate the promoter for each transcript:

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

And count the peaks that fall into a promoter:

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

          change
inPromoter down flat
     FALSE   21 5158
     TRUE     2  625
```

Or somewhere upstream or in a gene:

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

> sumtab <-
+     as.data.frame(rbind(total = xtabs(~ change, peakSummary),
```

```
+                           promoter = xtabs(~ change,
+                             subset(peakSummary, inPromoter)),
+                           upstream = xtabs(~ change,
+                             subset(peakSummary, inUpstream)),
+                           gene = xtabs(~ change, subset(peakSummary, inGene))))
> ##cbind(sumtab, ratio = round(sumtab[, "down"] /  sumtab[, "up"], 3))
```

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:

```
> 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:

```
> 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:

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

Version information

```
> sessionInfo()

R version 3.4.2 (2017-09-28)
Platform: x86_64-apple-darwin15.6.0 (64-bit)
Running under: OS X El Capitan 10.11.6

Matrix products: default
BLAS: /Library/Frameworks/R.framework/Versions/3.4/Resources/lib/libRblas.0.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/3.4/Resources/lib/libRlapack.dylib

locale:
```

```
[1] C/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

attached base packages:
[1] stats4    parallel  stats     graphics  grDevices utils     datasets
[8] methods   base

other attached packages:
 [1] TxDb.Mmusculus.UCSC.mm9.knownGene_3.2.2
 [2] lattice_0.20-35
 [3] GenomicFeatures_1.30.0
 [4] AnnotationDbi_1.40.0
 [5] chipseq_1.28.0
 [6] ShortRead_1.36.0
 [7] GenomicAlignments_1.14.0
 [8] SummarizedExperiment_1.8.0
 [9] DelayedArray_0.4.0
[10] matrixStats_0.52.2
[11] Biobase_2.38.0
[12] Rsamtools_1.30.0
[13] Biostrings_2.46.0
[14] XVector_0.18.0
[15] BiocParallel_1.12.0
[16] GenomicRanges_1.30.0
[17] GenomeInfoDb_1.14.0
[18] IRanges_2.12.0
[19] S4Vectors_0.16.0
[20] BiocGenerics_0.24.0

loaded via a namespace (and not attached):
 [1] Rcpp_0.12.13            compiler_3.4.2          RColorBrewer_1.1-2
 [4] prettyunits_1.0.2       progress_1.1.2          bitops_1.0-6
 [7] tools_3.4.2             zlibbioc_1.24.0         biomaRt_2.34.0
[10] digest_0.6.12           bit_1.1-12              RSQLite_2.0
[13] memoise_1.1.0           tibble_1.3.4            pkgconfig_2.0.1
[16] rlang_0.1.2             Matrix_1.2-11           DBI_0.7
[19] GenomeInfoDbData_0.99.1 rtracklayer_1.38.0      stringr_1.2.0
[22] hwriter_1.3.2           bit64_0.9-7             grid_3.4.2
[25] R6_2.2.2                XML_3.98-1.9            RMySQL_0.10.13
[28] latticeExtra_0.6-28     magrittr_1.5            blob_1.1.0
[31] assertthat_0.2.0        stringi_1.1.5           RCurl_1.95-4.8
```