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 <- 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:

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

```
GRangesList of length 2
$ctcf
GRanges with 450096 ranges and 0 elementMetadata values
         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]      -   |
     [6]    chr10     [3012969, 3012992]      -   |
     [7]    chr10     [3012978, 3013001]      -   |
     [8]    chr10     [3013071, 3013094]      -   |
     [9]    chr10     [3018464, 3018487]      -   |
     ...      ...                    ...    ... ...
[450088]    chr12 [121220518, 121220541]      +   |
[450089]    chr12 [121222581, 121222604]      +   |
[450090]    chr12 [121235756, 121235779]      +   |
[450091]    chr12 [121236634, 121236657]      +   |
[450092]    chr12 [121239376, 121239399]      +   |
[450093]    chr12 [121245849, 121245872]      +   |
[450094]    chr12 [121245895, 121245918]      +   |
[450095]    chr12 [121246344, 121246367]      +   |
[450096]    chr12 [121253499, 121253522]      +   |

...
<1 more element>


seqlengths
 chr10 chr11 chr12
    NA    NA    NA
```

`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 with 450096 ranges and 0 elementMetadata values
         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]      -   |
     [6]    chr10     [3012969, 3012992]      -   |
     [7]    chr10     [3012978, 3013001]      -   |
     [8]    chr10     [3013071, 3013094]      -   |
```

```
     [9]    chr10     [3018464, 3018487]      -   |
     ...      ...                    ...    ... ...
[450088]    chr12 [121220518, 121220541]      +   |
[450089]    chr12 [121222581, 121222604]      +   |
[450090]    chr12 [121235756, 121235779]      +   |
[450091]    chr12 [121236634, 121236657]      +   |
[450092]    chr12 [121239376, 121239399]      +   |
[450093]    chr12 [121245849, 121245872]      +   |
[450094]    chr12 [121245895, 121245918]      +   |
[450095]    chr12 [121246344, 121246367]      +   |
[450096]    chr12 [121253499, 121253522]      +   |

seqlengths
 chr10 chr11 chr12
    NA    NA    NA
```

The mouse genome

The data we have refer to alignments to a genome, and only makes sense in that context. Bioconductor has genome packages containing the full sequences of several genomes. The one relevant for us is

```
> library(BSgenome.Mmusculus.UCSC.mm9)
> seqlengths(cstest) <- seqlengths(Mmusculus)
```

We will only make use of the chromosome lengths, but the actual sequence will be needed for motif finding, etc.

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)
> fraglen[!is.na(fraglen)]

   chr10    chr11    chr12
191.7425 184.5548 196.0883
```

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 with 450096 ranges and 0 elementMetadata values
         seqnames                 ranges strand   |
            <Rle>              <IRanges>  <Rle>   |
     [1]    chr10     [3012760, 3012959]      -   |
     [2]    chr10     [3012765, 3012964]      -   |
     [3]    chr10     [3012768, 3012967]      -   |
     [4]    chr10     [3012779, 3012978]      -   |
     [5]    chr10     [3012787, 3012986]      -   |
     [6]    chr10     [3012793, 3012992]      -   |
     [7]    chr10     [3012802, 3013001]      -   |
     [8]    chr10     [3012895, 3013094]      -   |
     [9]    chr10     [3018288, 3018487]      -   |
     ...      ...                    ...    ... ...
[450088]    chr12 [121220518, 121220717]      +   |
[450089]    chr12 [121222581, 121222780]      +   |
[450090]    chr12 [121235756, 121235955]      +   |
[450091]    chr12 [121236634, 121236833]      +   |
[450092]    chr12 [121239376, 121239575]      +   |
[450093]    chr12 [121245849, 121246048]      +   |
[450094]    chr12 [121245895, 121246094]      +   |
[450095]    chr12 [121246344, 121246543]      +   |
[450096]    chr12 [121253499, 121253698]      +   |

seqlengths
         chr1         chr2         chr3 ...  chrY_random chrUn_random
    197195432    181748087    159599783 ...     58682461      5900358
```

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

SimpleRleList 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

SimpleRleViewsList of length 35
$chr1
Views on a 197195432-length Rle subject

views: NONE

$chr2
Views on a 181748087-length Rle subject

views: NONE

$chr3
Views on a 159599783-length Rle subject

views: NONE

...
<32 more elements>
```

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

```
> viewSums(islands)

SimpleIntegerList 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 200 200 200 200 200 200 200 ... 200 200 400 200 200 200 200
...
<25 more elements>

> viewMaxs(islands)

SimpleIntegerList 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"]] 8 1 1 1 1 1 1 1 2 2 1 1 1 1 2 1 ... 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	68335	13337	3090	958	388	236	151	129	116	83
chr11	72202	16015	4394	1383	603	346	244	170	161	125
chr12	59277	11486	2628	782	343	174	136	102	70	73
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	68385	14775	2500	618	299	223	175	171	130	130
chr11	72264	18090	3568	911	472	288	264	231	204	186
chr12	59338	12681	2112	482	249	194	142	113	109	99
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. 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.

```
> 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
```

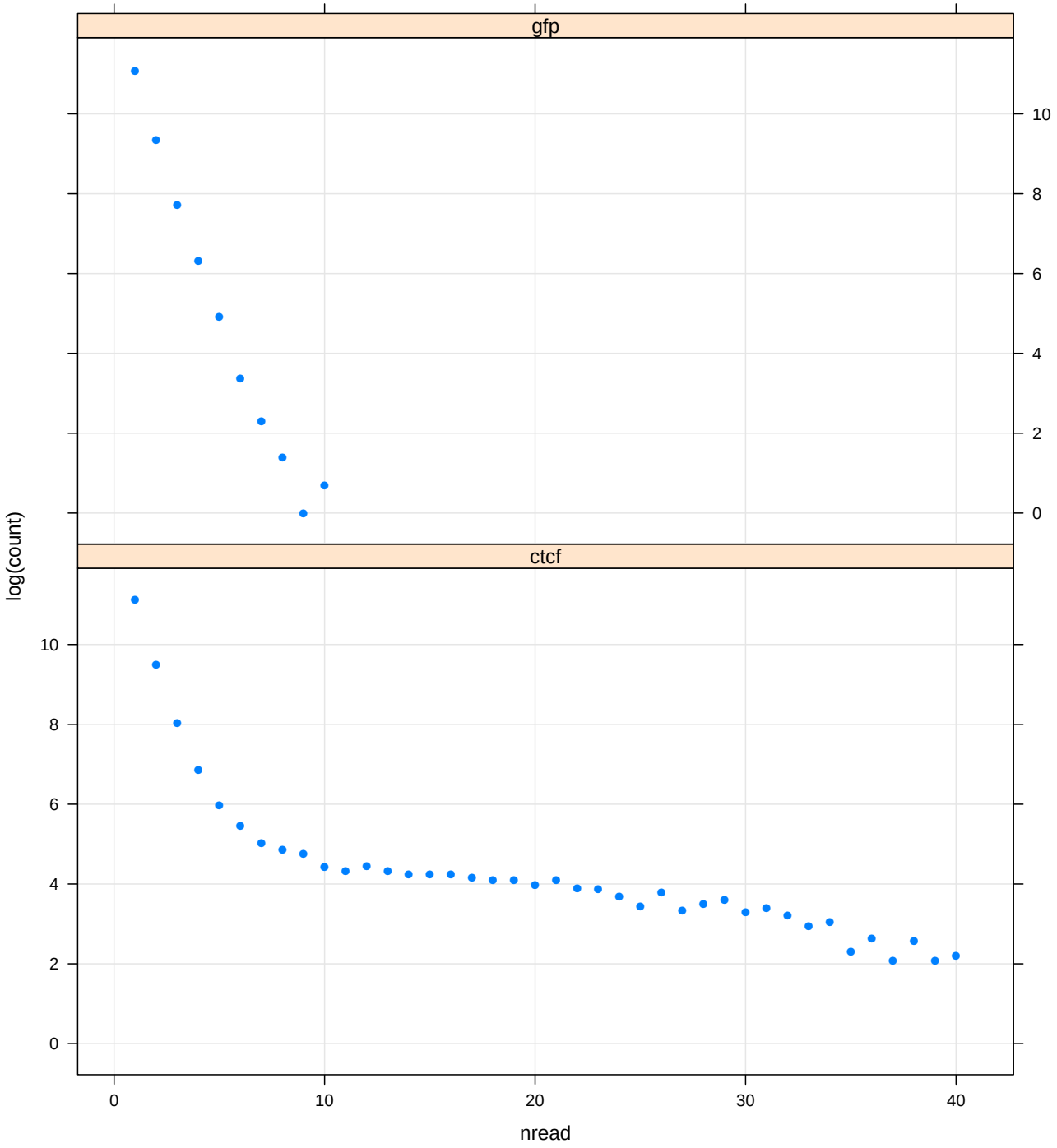
We can now use it to summarize the full dataset, flattening the returned *DataFrameList* with the `stack` function.

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

DataFrame with 4130 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
6      ctcf         chr6         1         0
7      ctcf         chr7         1         0
```

```
8      ctcf          chr8         1         0
9      ctcf          chr9         1         0
...     ...           ...       ...       ...
4122    gfp  chr7_random        31         0
4123    gfp  chr8_random        31         0
4124    gfp  chr9_random        31         0
4125    gfp chr13_random        31         0
4126    gfp chr16_random        31         0
4127    gfp chr17_random        31         0
4128    gfp  chrX_random        31         0
4129    gfp  chrY_random        31         0
4130    gfp chrUn_random        31         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"))
```

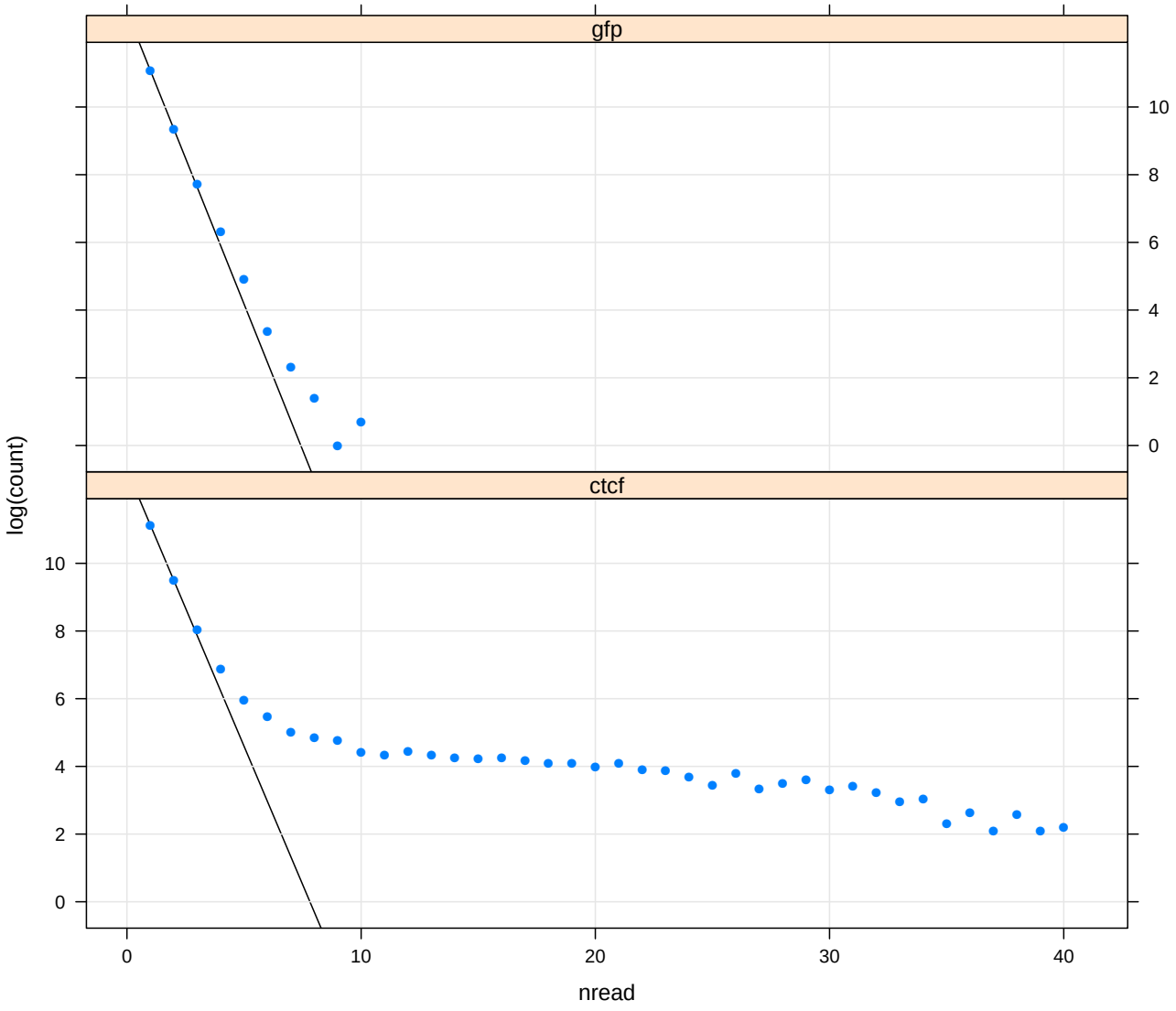


If reads were sampled randomly from the genome, then the null distribution number of reads per island would have a geometric distribution; that is,

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

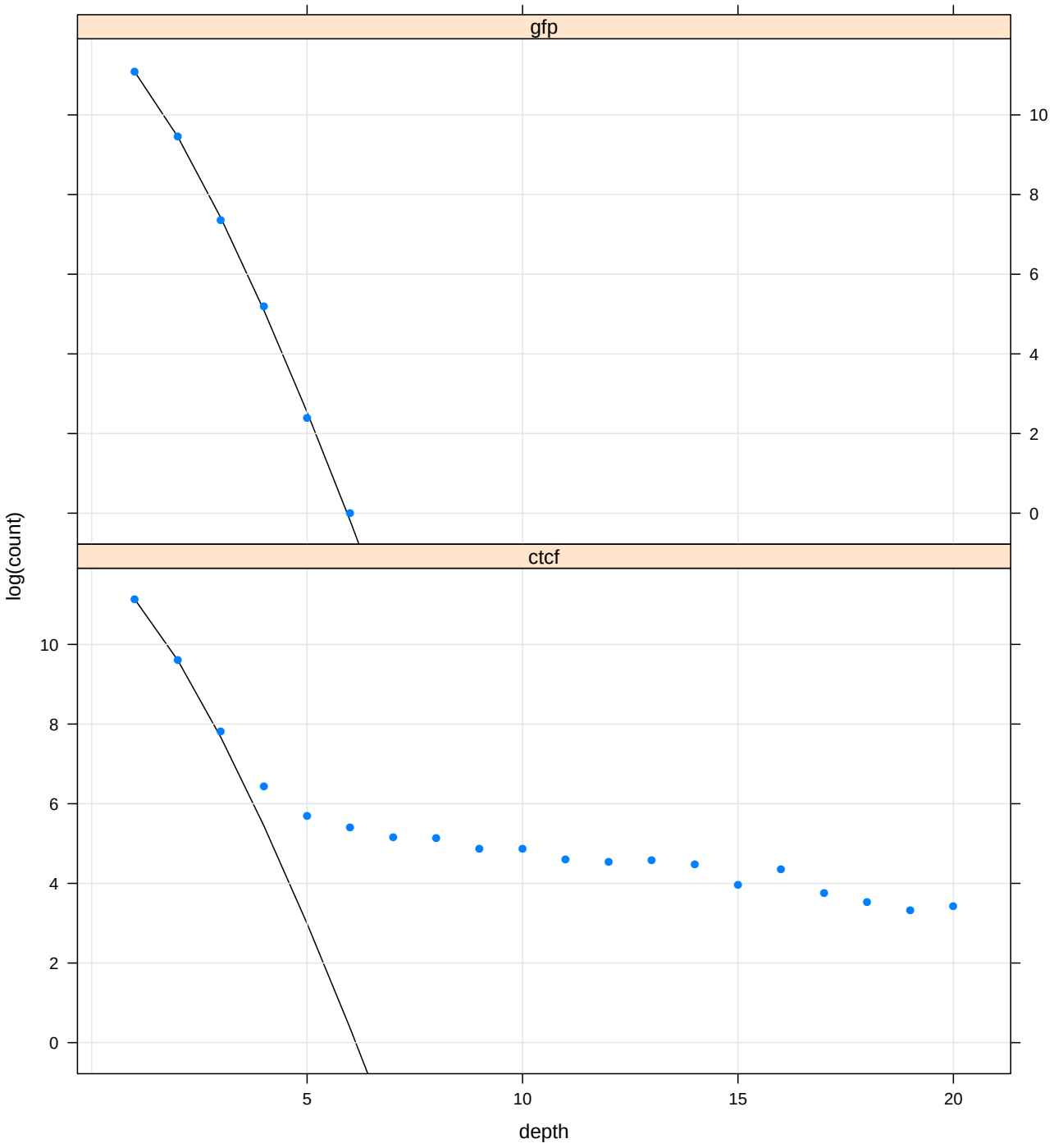
In other words, $\log P(X = k)$ is linear in k. Although our samples are not random, the islands with just one or two reads may be representative of the null distribution.

```
> 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 <- 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, ...)
+        })
```



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] 6.984781
```

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
```

```
SimpleRleViewsList of length 35
$chr1
Views on a 197195432-length Rle subject

views: NONE

$chr2
Views on a 181748087-length Rle subject

views: NONE

$chr3
Views on a 159599783-length Rle subject

views: NONE

...
<32 more elements>
```

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

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

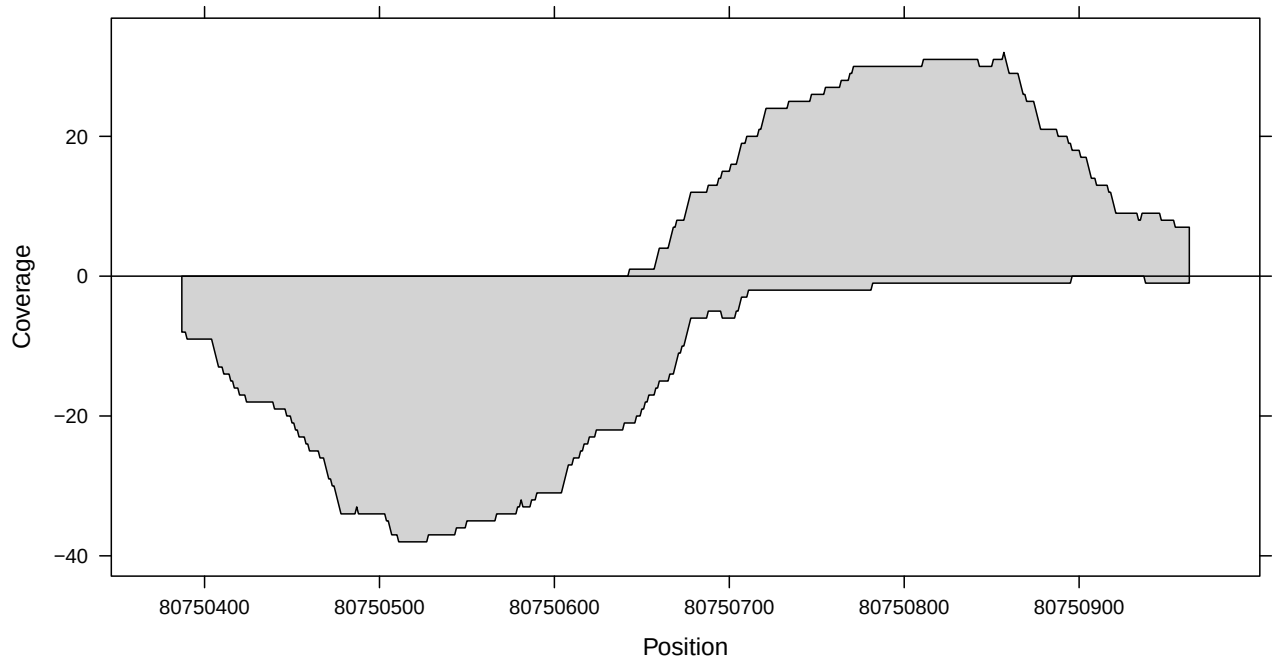
The result is a *RangedData* object with two columns: the view maxs and the view sums. Beyond that, this object is often useful as a scaffold for adding additional statistics.
It is meaningful to ask about the contribution of each strand to each peak, as the sequenced region of pull-down fragments would 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, peaks.ctcf)
> peaks.neg <- Views(cov.neg, peaks.ctcf)
> wpeaks <- tail(order(peak.depths$chr10), 4)
> wpeaks
```

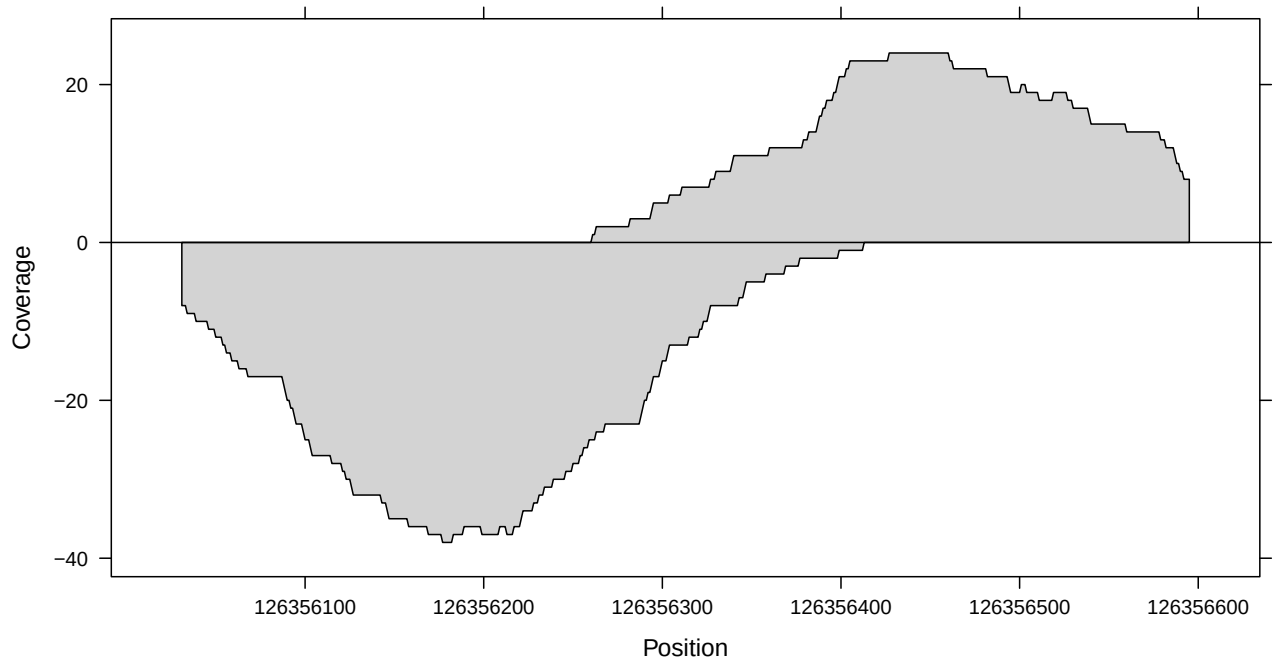
```
[1] 1407 2127  886 1179
```

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

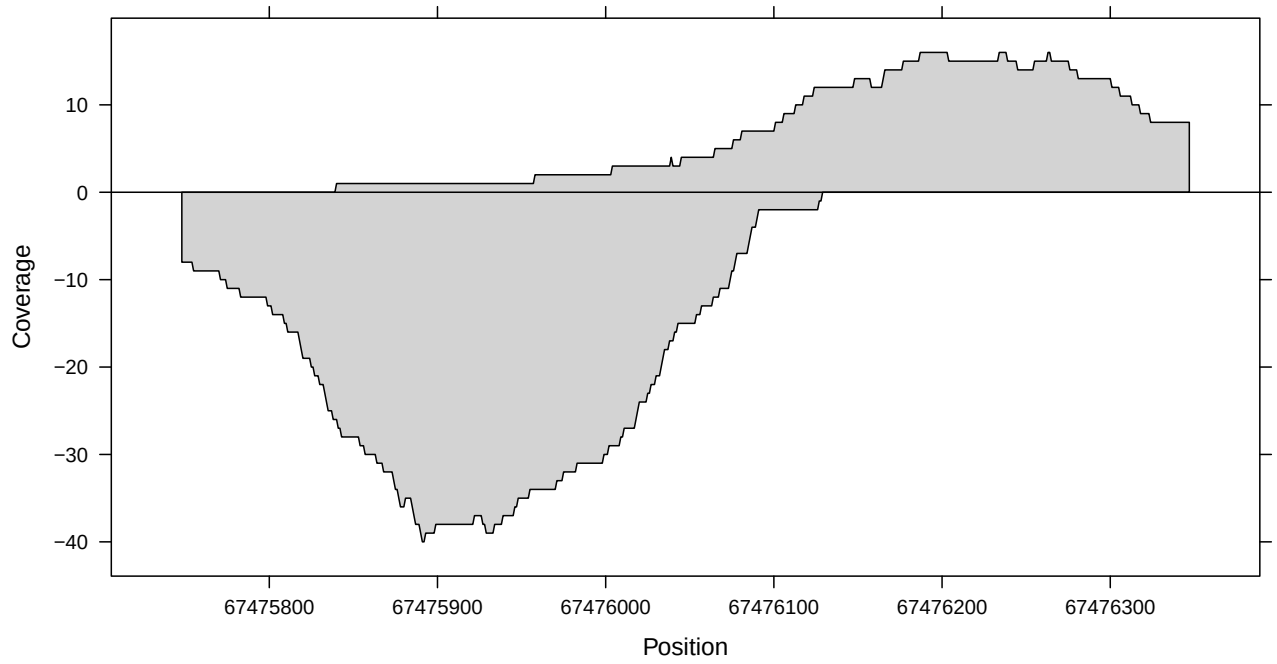
```
> coverageplot(peaks.pos$chr10[wpeaks[1]], peaks.neg$chr10[wpeaks[1]])
```



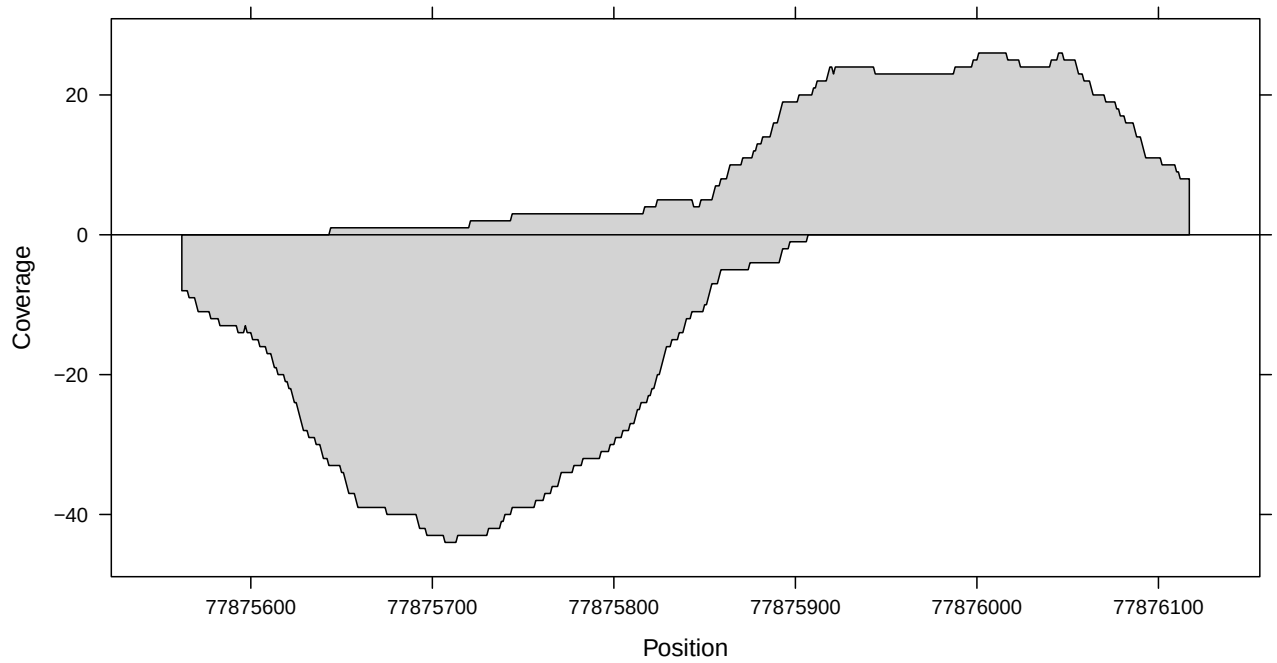

```
> coverageplot(peaks.pos$chr10[wpeaks[2]], peaks.neg$chr10[wpeaks[2]])
```



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



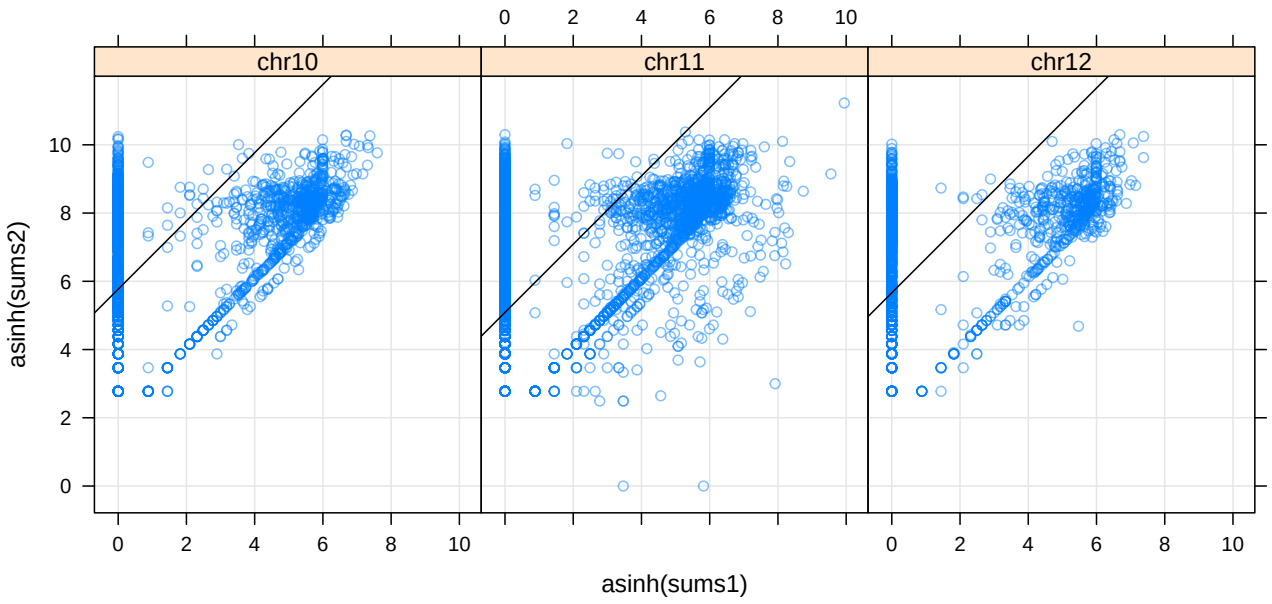

```
> coverageplot(peaks.pos$chr10[wpeaks[4]], peaks.neg$chr10[wpeaks[4]])
```



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.

```
> 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. Below, we 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).

```
> db <- makeTranscriptDbFromUCSC("mm9")
> gregions <- transcripts(db)
> gregions
```

```
GRanges with 49409 ranges and 2 elementMetadata values
           seqnames               ranges strand   |     tx_id     tx_name
              <Rle>            <IRanges>  <Rle>   | <integer> <character>
    [1]         chr1   [4797974, 4832908]      +   |        14  uc007afg.1
    [2]         chr1   [4797974, 4836816]      +   |        15  uc007afh.1
    [3]         chr1   [4847895, 4887985]      +   |        16  uc007afi.1
    [4]         chr1   [4848119, 4880877]      +   |        17  uc007afj.1
    [5]         chr1   [5073254, 5089858]      +   |        20  uc007afm.1
    [6]         chr1   [5073254, 5152630]      +   |        21  uc007afn.1
    [7]         chr1   [5578574, 5592947]      +   |        22  uc007afo.1
    [8]         chr1   [5578574, 5592947]      +   |        23  uc007afp.1
    [9]         chr1   [5586599, 5592947]      +   |        24  uc007afq.1
    ...          ...                 ...    ... ...       ...         ...
[49401] chrY_random [42903555, 42936135]      -   |     49280  uc009vhr.1
[49402] chrY_random [43008001, 43010364]      -   |     49281  uc009vhs.1
[49403] chrY_random [44315018, 44341235]      -   |     49285  uc009vhw.1
[49404] chrY_random [46075785, 46078095]      -   |     49288  uc009vhz.1
[49405] chrY_random [48319175, 48321522]      -   |     49293  uc009vie.1
[49406] chrY_random [48388361, 48390745]      -   |     49294  uc009vif.1
[49407] chrY_random [50814754, 50817087]      -   |     49296  uc009vih.1
[49408] chrY_random [52063165, 52089373]      -   |     49298  uc009vij.1
[49409] chrY_random [52063165, 52089373]      -   |     49299  uc009vik.1

seqlengths
      chr1       chr2       chr3 ...  chrX_random  chrY_random
 197195432  181748087  159599783 ...      1785075     58682461
```

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 %in% promoters
> xtabs(~ inPromoter + change, peakSummary)
```

```
          change
inPromoter down flat
     FALSE    3 6971
     TRUE     0  776
```

Or somewhere upstream or in a gene:

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

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 2.12.0 RC (2010-10-11 r53293)
Platform: i386-pc-mingw32/i386 (32-bit)

locale:
```

```
[1] LC_COLLATE=C
[2] LC_CTYPE=English_United States.1252
[3] LC_MONETARY=English_United States.1252
[4] LC_NUMERIC=C
[5] LC_TIME=English_United States.1252

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

other attached packages:
 [1] BSgenome.Mmusculus.UCSC.mm9_1.3.16 GenomicFeatures_1.2.0
 [3] chipseq_1.0.0                      ShortRead_1.8.0
 [5] Rsamtools_1.2.0                    lattice_0.19-13
 [7] BSgenome_1.18.0                    Biostrings_2.18.0
 [9] GenomicRanges_1.2.0                IRanges_1.8.0

loaded via a namespace (and not attached):
 [1] Biobase_2.10.0     DBI_0.2-5          RCurl_1.4-4.1      RSQLite_0.9-2
 [5] XML_3.2-0.1        biomaRt_2.6.0      grid_2.12.0        hwriter_1.2
 [9] rtracklayer_1.10.0 tools_2.12.0
```