Second-Generation Sequence Data Analysis with R and Bioconductor

Sean Davis

National Cancer Institute,
National Institutes of Health
Bethesda, MD, USA

sdavis2@mail.nih.gov

September 16, 2009

Abstract

This document has two main goals: (i) To document some common and simple use cases for sequence data analysis using R and Bioconductor and (ii) To provide a playground for experimental functions for publicly (and locally) available data.

1 Introduction

Second-generation sequencing technologies bring huge data volumes, complex experimental design, algorithmic and data visualization challenges, and data integration nightmares. While much of the technology is proprietary, the software to deal with the data has largely been left to the open-source community. The Bioconductor project is one such open-source community that is working on problems related to second-generation sequencing. The idea is to document workflows for as many use cases as possible, not with the goal of becoming the best tool for any particular workflow, but to provide a set of tools that are useful for sequence data analysis. These tools are being developed collaboratively within the Bioconductor community.

2 Use Cases

While there are a huge number of applications for second-generation sequencing, there are a few use cases that demonstrate the current functionality for dealing with sequence data from within Bioconductor. The use cases will necessarily be somewhat abridged and illustrative and are not meant to be "full analyses" by any means. In fact, the power and flexibilty of using Bioconductor instead of a "canned" software can really only be appreciated by extending these analyses somewhat beyond what is presented here.

Sequence analysis using R and Bioconductor relies on several "packages" that provide extended functionality beyond core R. We start by loading these packages into R:

```
> suppressMessages(library(ShortRead))
> suppressMessages(library(Rpressa))
```

2.1 Targeted Sequencing

Sequencing whole genomes is still generally prohibitive in terms of time and money for most labs. Therefore, various molecular biology methods have been developed to enrich the regions of the genome of most biological interest. In the use case presented here, capture probes of 120 base pairs were used for hybridization of exons in genes of interest. Approximately 27,000 120-mer probes were used for the experiment here. The targeted genes were chosen because they have a higher likelihood of being mutated in cancer or are in pathways that are of interest in cancer.

The first step is to simply load a lane of data:

```
> data(targeted)
> targeted
```

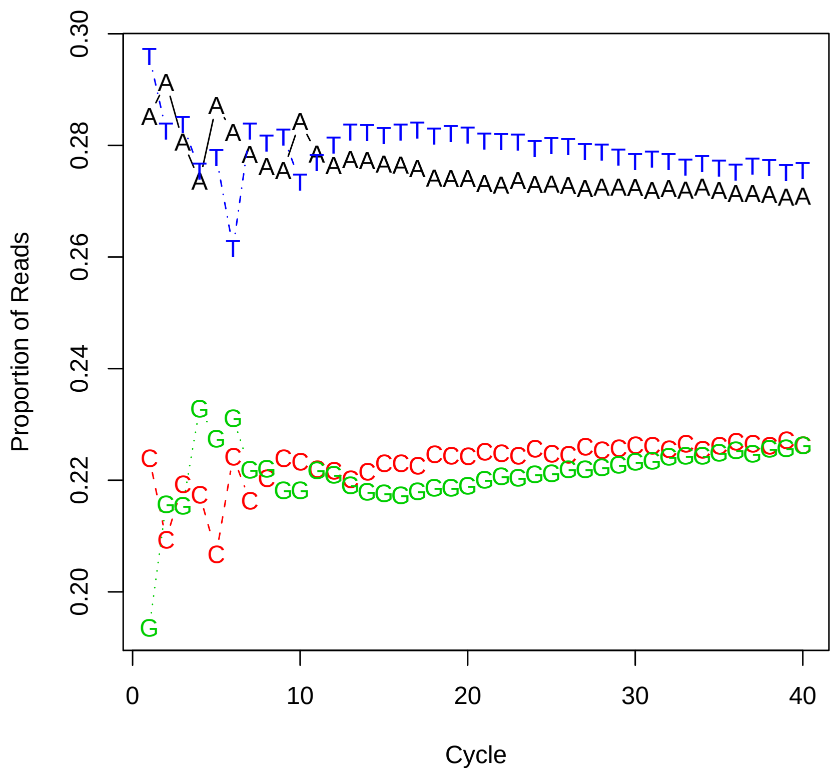
```
class: AlignedRead
length: 6349344 reads; width: 40 cycles
chromosome: QC QC ... QC QC
position: NA NA ... NA NA
strand: NA NA ... NA NA
alignQuality: NumericQuality
alignData varLabels: run lane ... filtering contig
```

The R object `targeted` contains data on 6349344 reads including those reads that were poor quality and those that did not align to the human genome. Getting the reads that align to the human genome is fairly easy.

```
> aln2 <- targeted[!is.na(position(targeted))]
```

The data in `aln2` are now those reads that align to the human genome; there are 4995174 such reads (78.67 % of the reads). There are a number of quality control functions and accessors that could be applied to these data. An interesting one is to look at the proportion of bases at each cycle of the read.

```
> library(lattice)
> abc <- (alphabetByCycle(sread(aln2))/length(aln2))[1:4, ]
> colnames(abc) <- 1:40
> abc <- t(abc)
> matplot(abc, type = "b", xlab = "Cycle", ylab = "Proportion of Reads",
+     pch = c("A", "C", "G", "T"))
```



It is interesting to look at sequencing coverage for the capture regions as a quality control measure. Loading the description of the capture regions from a bed-format file is quite straightforward using another Bioconductor package, *rtracklayer*.

```
> suppressMessages(library(rtracklayer))
> bedfile <- system.file("extdata/agilent27k.lot1.bed", package = "Rpressa")
> rl.capture <- ranges(import(bedfile))
> rl.capture
```

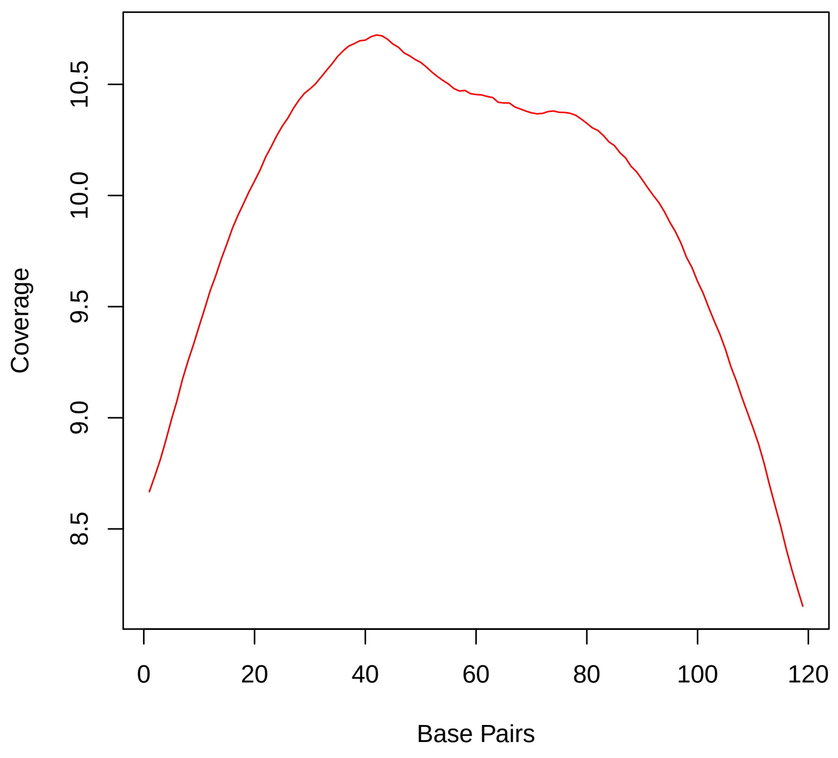
```
CompressedIRangesList: 25 elements
names(25): chr1 chr10 chr11 chr12 chr13 ... chr8 chr9 chrX chrX_random chrY
```

Calculating the number of times each base in the genome has been sequenced is also easily accomplished.

```
> cvg <- coverage(aln2)
> names(cvg) <- sub(".fa", "", names(cvg))
> cvg <- cvg[names(cvg) %in% names(rl.capture)]
> vcvg <- Views(cvg, rl.capture)
```

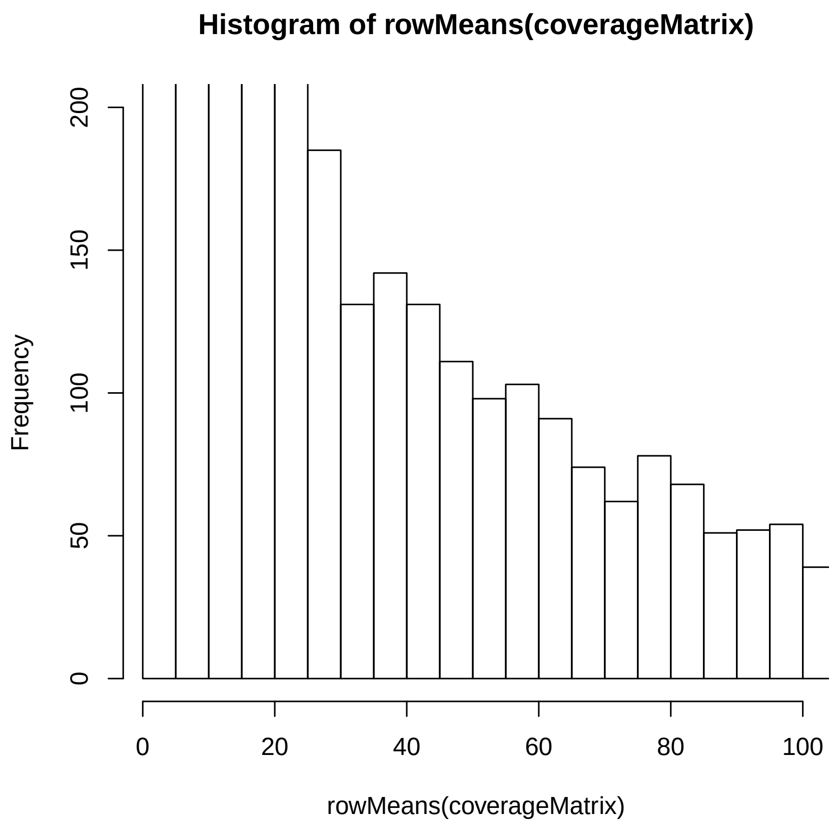
Now, `vcvg` is a view of the genomic coverage that contains the regions targeted by the targeting probes. How well are the targeted regions covered?

```
> coverageMatrix <- t(do.call(cbind, as.list(viewApply(vcvg, as.vector))))
> plot(colMeans(coverageMatrix), type = "l", col = "red", xlab = "Base Pairs",
+     ylab = "Coverage")
```



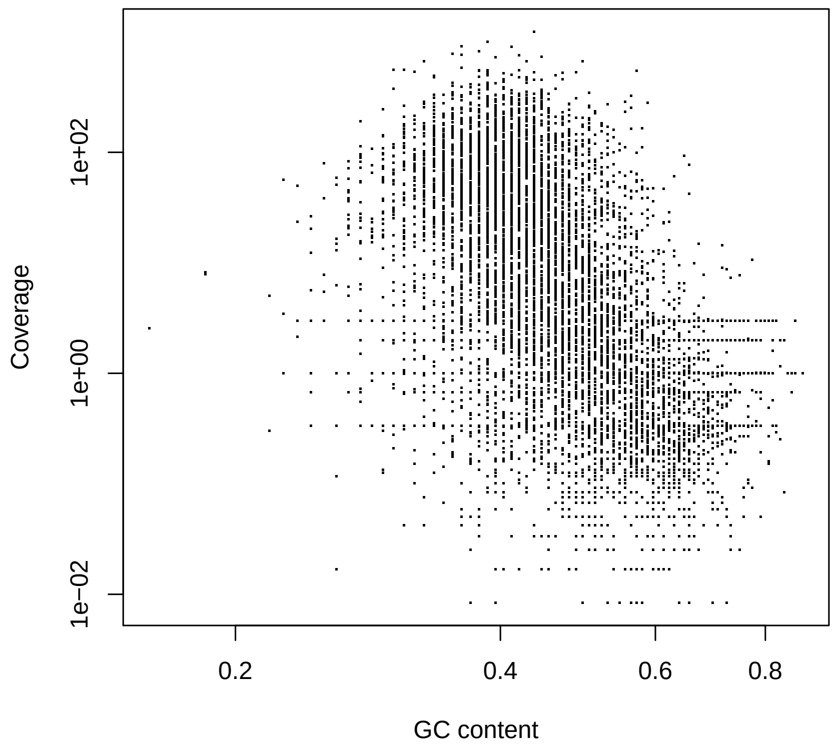
And how many regions have a mean coverage above a given threshold?

```
> hist(rowMeans(coverageMatrix), breaks = "scott", xlim = c(1,
+     100), ylim = c(0, 200))
```



In short, 7866 regions have mean coverage>1 while 3053 have coverage>10. Bioconductor also has data packages that contain the entire human genome sequence, but in a compact and random-accessible form for memory efficiency. With this information in hand, it might be interesting to look at the effect of the GC content of the capture oligos on genomic coverage.

```
> library(BSgenome)
> library(BSgenome.Hsapiens.UCSC.hg18)
> regionDNA <- DNAStringSet(getSeq(Hsapiens, rl.capture))
> x <- alphabetFrequency(regionDNA, as.prob = TRUE)
> avgCvg <- rowMeans(coverageMatrix)
> x <- x[avgCvg > 0, 2:3]
> plot(rowSums(x), avgCvg[avgCvg > 0], log = "xy", xlab = "GC content",
+     ylab = "Coverage", pch = ".")
```



2.2 ChIP-Seq

This section borrows heavily from the vignette for the *chipseq*.

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 readReads function, which in turn uses the readAligned function in the *ShortRead*. This step removed all duplicate reads and applied a quality score cutoff. The remaining data were reduced to a set of alignment start positions (including orientation).

```
> suppressMessages(library(chipseq))
> data(cstest)
> cstest
```

```
GenomeDataList: 2 elements
names(2): ctcf gfp
```

Extending Reads

The sequencer generally reads only the first n (where n is typically on the order of 36bp for ChIP-seq applications), but the typical insert size is on the order of 150-250bp. The TFBS of interest is somewhere in that fragment (ideally), but not necessarily at the beginning. Therefore, it is useful to extend the reads to the full length of the average fragment size. To make the chromosomes the correct size, the lengths of the chromosomes need to be used.

```
> library(BSgenome.Mmusculus.UCSC.mm9)
> mouse.chromlens <- seqlengths(Mmusculus)
> head(mouse.chromlens)

     chr1      chr2      chr3      chr4      chr5      chr6
197195432 181748087 159599783 155630120 152537259 149517037
```

We extend all reads to be 200 bases long. This is done using the extendReads() function, which can work on data from one chromosome in one lane.

```
> ext <- extendReads(cstest$ctcf$chr10, seqLen = 200)
> head(ext)

IRanges instance:
      start     end width
[1] 3012936 3013135   200
[2] 3012941 3013140   200
[3] 3012944 3013143   200
[4] 3012955 3013154   200
[5] 3012963 3013162   200
[6] 3012969 3013168   200
```

As with the targeted sequencing example, computing coverage can be a useful way of looking at the data. To keep things simple, the analysis is first restricted to chromosome 10.

```
> cov <- coverage(ext, width = mouse.chromlens["chr10"])
> cov

  'integer' Rle instance of length 129993255 with 288928 runs
  Lengths:  3012799 97 2 37 5 3 11 8 6 9 ...
  Values :  0 1 2 3 5 6 7 8 9 10 ...
```

The regions of interest are contiguous segments of non-zero coverage, which the *chipseq* package refers to as *islands.*

```
> islands <- slice(cov, lower = 1)
> islands

  Views on a 129993255-length Rle subject

views:
            start       end width
    [1]   3012800   3013270   471 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...]
    [2]   3018464   3018663   200 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...]
    [3]    3020766   3020965   200 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...]
    [4]    3023019   3023218   200 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...]
    [5]    3023240   3023439   200 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...]
    [6]    3032536   3032735   200 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...]
    [7]    3038377   3038576   200 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...]
    [8]    3040312   3040554   243 [1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 ...]
    [9]    3041098   3041297   200 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...]
    ...        ...       ...   ... ...
[87957] 129973175 129973447   273 [1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 ...]
[87958] 129974813 129975012   200 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...]
[87959] 129975575 129975774   200 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...]
[87960] 129978669 129978868   200 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...]
[87961] 129979209 129979571   363 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...]
[87962] 129980253 129980452   200 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...]
[87963] 129981957 129982156   200 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...]
[87964] 129982330 129982529   200 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...]
[87965] 129987020 129987219   200 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...]
```

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

```
> viewSums(head(islands))
```

```
[1] 2400  200  200  200  200  200
```

```
> viewMaxs(head(islands))
```

```
[1] 11  1  1  1  1  1
```

```
> nread.tab <- table(viewSums(islands)/200)
> depth.tab <- table(viewMaxs(islands))
> head(nread.tab, 10)
```

```
    1     2     3     4     5     6     7     8     9    10
68111 13350  3022   925   415   247   191   122   132   100
```

```
> head(depth.tab, 10)
```

```
    1     2     3     4     5     6     7     8     9    10
68159 14745  2388   547   256   180   150   129   120   102
```

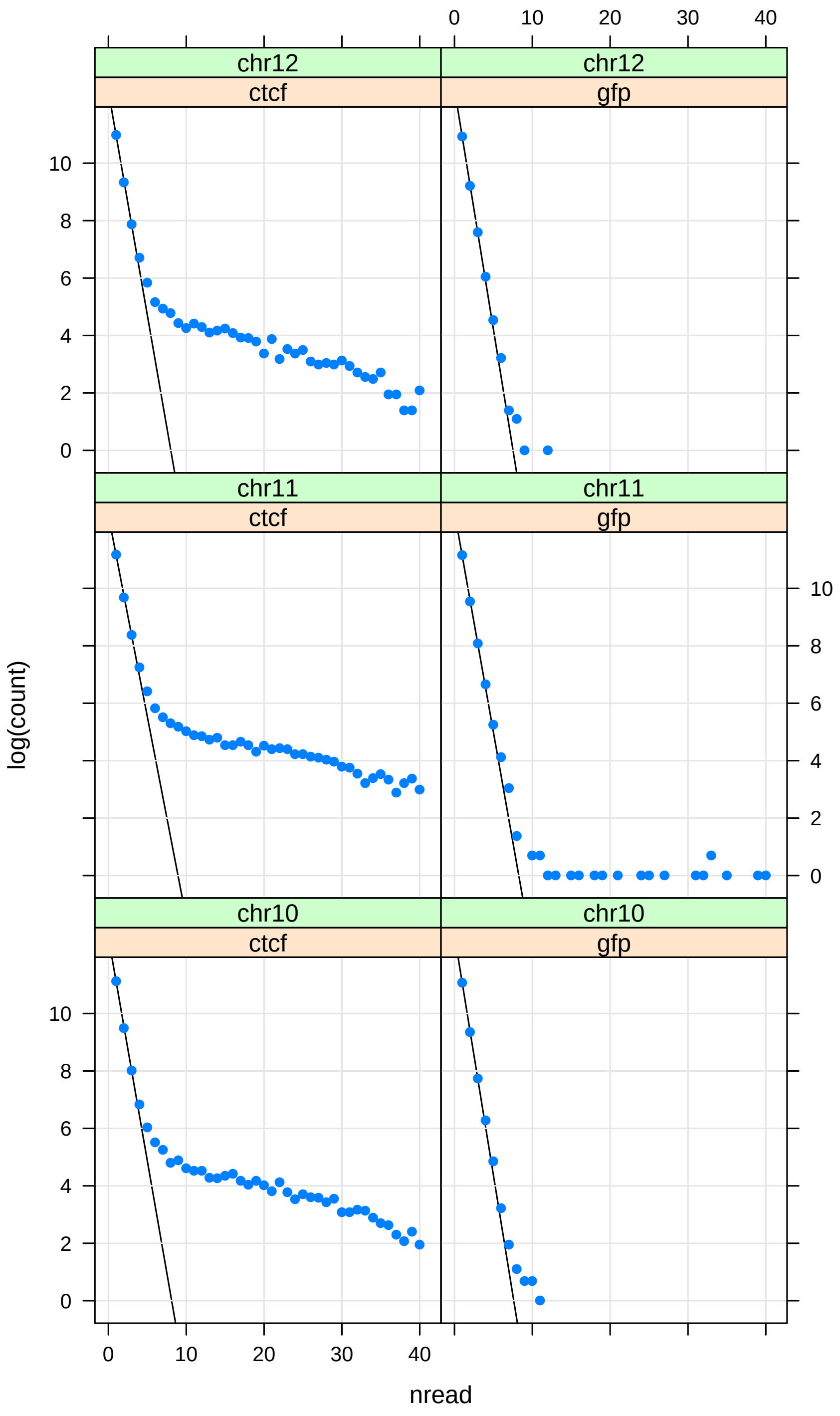
It is also possible to process all the data in all lanes simultaneously.

```
> islandReadSummary <- function(x) {
+     g <- extendReads(x, seqLen = 200)
+     s <- slice(coverage(g), lower = 1)
+     tab <- table(viewSums(s)/200)
+     ans <- data.frame(nread = as.numeric(names(tab)), count = as.numeric(tab))
+     ans
+ }
```

```
> nread.islands <- gdapply(cstest, islandReadSummary)
> nread.islands <- as(nread.islands, "data.frame")
> head(nread.islands)

  nread count chromosome sample
1     1 68111      chr10   ctcf
2     2 13350      chr10   ctcf
3     3  3022      chr10   ctcf
4     4   925      chr10   ctcf
5     5   415      chr10   ctcf
6     6   247      chr10   ctcf
```

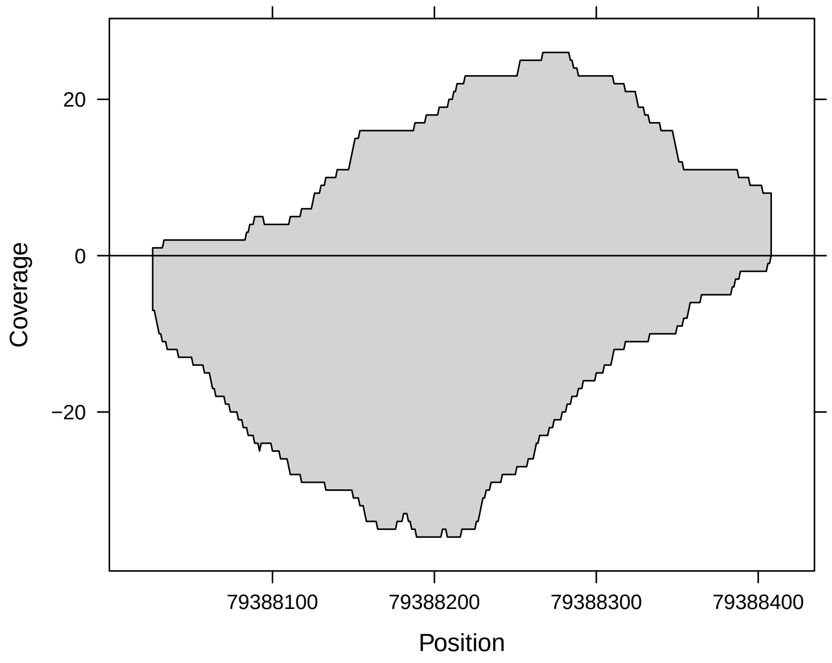
A simple plot of the log(count) versus the number of reads in each island is useful. If the reads were randomly distributed across the genome, the relationship should be linear. In the GFP lane, this is close to true for most of the data. However, for CTCF, there is obvious deviation from linear. Points to the right of the line in each plot are, then, "significant" in some sense and a threshold of 8 reads in an island looks like a good pick for finding islands of significance.

```
> xyplot(log(count) ~ nread | sample + chromosome, nread.islands,
+     subset = (nread <= 40), pch = 16, type = c("p", "g"), panel = function(x,
+         y, ...) {
+         panel.lmline(x[1:3], y[1:3], col = "black")
+         panel.xyplot(x, y, ...)
+     })
```

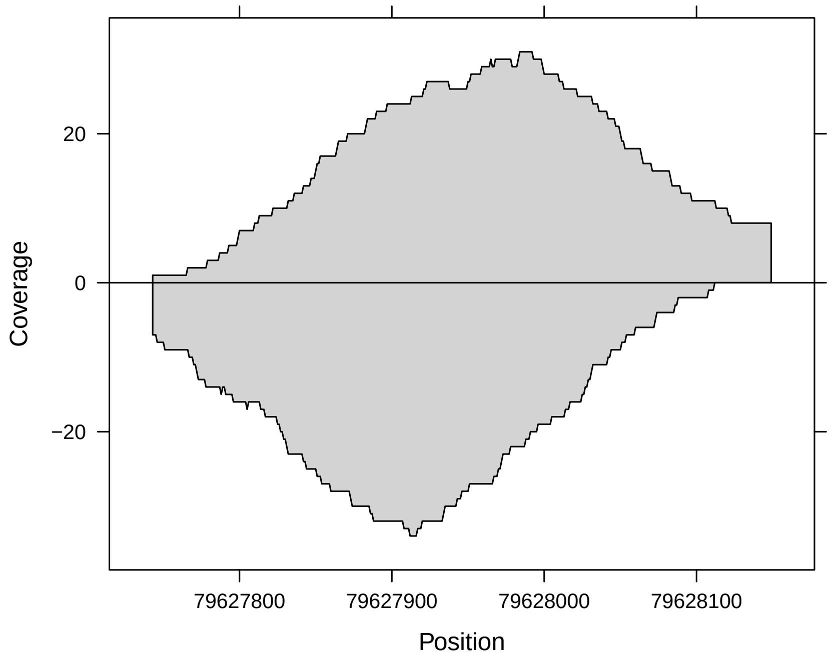


Finding peaks is a fairly simple procedure. Note that the threshold defined by the linear extrapolation in the plot is used.

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

  Views on a 129993255-length Rle subject

views:
           start       end width
   [1]   3012955   3013135   181 [ 8  8  8  8  8  8  8  8  9  9  9  9  9 ...]
   [2]   3234799   3234895    97 [ 8  8  8  8  8  8  8  8  8  8  8  8  8 ...]
   [3]   3270012   3270297   286 [8 8 8 9 9 9 9 9 8 8 8 8 8 8 8 8 8 8 8 8 8 ...]
   [4]   3277662   3277832   171 [ 8  8  8  8  8  8  8  8  8  8  8  8  8 ...]
   [5]   3277848   3277859    12 [8 8 8 8 8 8 8 8 8 8 8 8]
   [6]   3460857   3460973   117 [ 8  8  8  8  8  8  8  8  8  8  8  8  8 ...]
   [7]   3617850   3617983   134 [ 8  8  8  8  8  8  8  9  9  9  9 10 10 ...]
   [8]   3651712   3651992   281 [ 8  8  9  9  9 10 10 10 10 10 10 10 10 ...]
   [9]   4310402   4310712   311 [ 8  8  8  9  9  9  9  9  9  9  9 10 10 ...]
   ...       ...       ...   ... ...
[1747] 128986519 128986595    77 [8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 ...]
[1748] 128986604 128986610     7 [8 8 8 8 8 8 8]
[1749] 128986638 128986673    36 [8 8 8 8 8 8 8 9 9 9 9 8 8 8 8 9 9 9 8 8 8 ...]
[1750] 129058889 129058980    92 [8 8 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 ...]
[1751] 129530031 129530201   171 [ 8  8  9  9  9  9  9  9  9  9  9  9  9 ...]
[1752] 129533303 129533381    79 [8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 ...]
[1753] 129665351 129665586   236 [ 8  9  9  9  9  9  9  9  9 10 10 10 10 ...]
[1754] 129666784 129666947   164 [8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 ...]
[1755] 129750671 129750849   179 [ 8  8  8  8  8  8  9  9  9  9  9  9  9 ...]

> peak.depths <- viewMaxs(peaks)
> cov.pos <- coverage(extendReads(cstest$ctcf$chr10, strand = "+",
+     seqLen = 200), width = mouse.chromlens["chr10"])
> cov.neg <- coverage(extendReads(cstest$ctcf$chr10, strand = "-",
+     seqLen = 200), width = mouse.chromlens["chr10"])
> peaks.pos <- copyIRanges(peaks, cov.pos)
> peaks.neg <- copyIRanges(peaks, cov.neg)
> wpeaks <- tail(order(peak.depths), 4)
> wpeaks

[1]  971  989 1079  922

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



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


Coverage
Position