Analysis of RNA-Seq data with Bioconductor

Kasper Daniel Hansen <khansen@jhsph.edu>
Department of Biostatistics
Johns Hopkins Bloomberg School of Public Health

based on material co-developed with
James Bullard (UC Berkeley)
Margaret Taub (Johns Hopkins)

FHCRC
November 18-20, 2009

Data

```
> require(yeastRNASeq)
> require(ShortRead)
> require(Genominator)
> data(yeastAligned)
```

yeastRNASeq has data from Lee et al (PLoS Gen 2009).
500,000 reads from 4 lanes.
A *wild-type* (wt) and a *mutant* (mut) strain of *S. cerevisiae* were sequenced.
Reads were aligned using `Bowtie`

```
> sapply(yeastAligned, length)
```

```
mut_1_f mut_2_f  wt_1_f  wt_2_f
 423318  420848  410349  430264
```

```
> yeastAligned[[1]]
```

Data

```
class: AlignedRead
length: 423318 reads; width: 26 cycles
chromosome: Scchr05 Scchr15 ... Scchr08 Scchr13
position: 541317 885627 ... 488228 667296
strand: - + ... - +
alignQuality: NumericQuality
alignData varLabels: similar mismatch
```

Annotation

There are (at least) two standard ways to obtain annotation using Bioconductor. One uses the biomaRt package to query Ensembl and the other uses the rtracklayer package to query UCSC.
Which annotation to use is a *biological* question and there is no universal answer. There are differences between Ensembl and UCSC.
Be careful that the annotation you are using corresponds to the genome you have used for mapping the reads.
(Cautionary tale)

biomaRt in one slide

The vignette is great, with lots of useful examples. Some important functions include:

- `listMarts()`: Displays the marts (basically databases) which are available for query
- `useMart()`: Sets up a connection to a particular mart
- `listDatasets()`: Lists the different databases available in a mart
- `listAttributes()`, `listFilters()`: Give a lists of the fields available for query (or filter) in the database you are interested in, providing a name (used for the query) and a brief description
- `getBM()` submits your query; you specify the attributes you want, and it allows you to select subsets of the results using the `filters` and `values` arguments

There are many more functions in biomaRt designed to ease the execution of common searches.

Using biomaRt

Ensembl is transcript-centric. But watch out!

```
> require(biomaRt)
> mart <- useMart("ensembl", "scerevisiae_gene_ensembl")
> attributes.gene <- c("ensembl_gene_id", "chromosome_name",
+     "start_position", "end_position", "strand",
+     "gene_biotype")
> attributes.tr <- c("ensembl_gene_id", "ensembl_transcript_id",
+     "ensembl_exon_id", "chromosome_name", "start_position",
+     "end_position", "strand", "gene_biotype",
+     "exon_chrom_start", "exon_chrom_end", "rank")
> yAnno.gene <- getBM(attributes = attributes.gene,
+     mart = mart)
> yAnno.tr <- getBM(attributes = attributes.tr,
+     mart = mart)
```

Let us look at the objects

```
> head(yAnno.gene, 2)
```

Using biomaRt

```
  ensembl_gene_id chromosome_name start_position
1         YHR055C             VIII         214535
2         YPR161C              XVI         864445
  end_position strand   gene_biotype
1       214720     -1 protein_coding
2       866418     -1 protein_coding
```

```
> head(yAnno.tr, 2)
```

```
  ensembl_gene_id ensembl_transcript_id ensembl_exon_id
1         YHR055C                YHR055C       YHR055C.1
2         YPR161C                YPR161C       YPR161C.1
  chromosome_name start_position end_position strand
1            VIII         214535       214720     -1
2             XVI         864445       866418     -1
    gene_biotype exon_chrom_start exon_chrom_end rank
1 protein_coding           214535         214720    1
2 protein_coding           864445         866418    1
```

```
> dim(yAnno.gene)
```

Using biomaRt

```
[1] 7124    6

> dim(yAnno.tr)

[1] 7547   11

> subset(yAnno.gene, ensembl_gene_id == "YPR098C")

  ensembl_gene_id chromosome_name start_position
7         YPR098C              XVI         728945
  end_position strand   gene_biotype
7       729526     -1 protein_coding

> subset(yAnno.tr, ensembl_gene_id == "YPR098C")
```

Using biomaRt

```
  ensembl_gene_id ensembl_transcript_id ensembl_exon_id
7         YPR098C                YPR098C       YPR098C.1
8         YPR098C                YPR098C       YPR098C.2
  chromosome_name start_position end_position strand
7             XVI         728945       729526     -1
8             XVI         728945       729526     -1
    gene_biotype exon_chrom_start exon_chrom_end rank
7 protein_coding           729480         729526    1
8 protein_coding           728945         729383    2
```

```
> length(unique(yAnno.tr$ensembl_transcript_id))
```

```
[1] 7124
```

We will use `yAnno.tr`.

```
> yAnno <- yAnno.tr
```

Using rtracklayer

There are two (possibly relevant) tables at UCSC.

```
> require(rtracklayer)
> session <- browserSession()
> genome(session) <- "sacCer2"
> ucsc.sgd <- getTable(ucscTableQuery(session, "sgdGene"))
> ucsc.ens <- getTable(ucscTableQuery(session, "ensGene"))
```

Let us have a quick look

```
> head(ucsc.sgd, 1)
```

```
  bin    name chrom strand txStart  txEnd cdsStart cdsEnd
1  73 YAL012W  chrI      +  130801 131986   130801 131986
  exonCount exonStarts exonEnds proteinID
1         1    130801,  131986,    P31373
```

```
> head(ucsc.ens, 1)
```

Using rtracklayer

```
  bin    name chrom strand txStart  txEnd cdsStart cdsEnd
1  73 YAL012W  chrI      +  130801 131986   130801 131986
  exonCount exonStarts exonEnds score   name2 cdsStartStat
1         1    130801,  131986,     0 YAL012W         cmpl
  cdsEndStat exonFrames
1       cmpl         0,
```

```
> subset(ucsc.sgd, name == "YPR098C")
```

```
     bin    name  chrom strand txStart  txEnd cdsStart
5756 590 YPR098C chrXVI      -  728944 729526   728944
     cdsEnd exonCount     exonStarts       exonEnds
5756 729526         2 728944,729479, 729383,729526,
     proteinID
5756    Q06089
```

```
> subset(ucsc.ens, name == "YPR098C")
```

Using rtracklayer

```
     bin    name  chrom strand txStart  txEnd cdsStart
6104 590 YPR098C chrXVI      -  728944 729526   728944
     cdsEnd exonCount     exonStarts       exonEnds score
6104 729526         2 728944,729479, 729383,729526,     0
       name2 cdsStartStat cdsEndStat exonFrames
6104 YPR098C         cmpl       cmpl       2,0,
```

No information about which genes are "verified", "uncharacterized" or "dubious". I also could not find this information in Ensembl. One might need to obtain annotation directly from SGD (a database specific to *S. Cerevisiae*) in order to retrieve this.

Computing on annotation

I have found IRanges to be very powerful and efficient when I want to compute on annotation (overlaps, set operations etc.) Some examples of this later.
It is quite common that some amount of post-processing of the annotation needs to be done.
Selecting the right annotation and processing it appropriately is a challenge. Especially for analysis of alternative splicing.

Summarizing at the gene level

We will now try to count the number of reads starting in each genomic region (gene). We want to get an end result like this

	mut_1	mut_2	wt_1	wt_2
YHR055C	0	0	0	0
YPR161C	38	39	35	34
YOL138C	31	33	40	26
YDR395W	55	52	47	47
YGR129W	29	26	5	5
YPR165W	189	180	151	180

We will discuss two approaches, one using IRanges and one using Genominator.

Counting

The (standard) RNA-Seq assay does not retain strand information, so strand will be ignored in the following.
We ignore (for now) the fact that in *S. Cerevisiae* genes often overlap each other on different strands. We also ignore splicing (although that is less of an issue in this organism).
We will count the number of reads whose 5' end falls within a genomic region. This is just one way of counting. (Discuss).

Using IRanges to represent our annotation

First, we need to match the chromosome names as Ensembl uses roman numerals and the Bowtie index used names.

```
> chrMap <- levels(chromosome(yeastAligned[[1]]))
> names(chrMap) <- c(as.character(as.roman(1:16)),
+     NA)
> head(chrMap)
```

```
       I        II       III        IV         V        VI
"Scchr01" "Scchr02" "Scchr03" "Scchr04" "Scchr05" "Scchr06"
```

```
> yAnno$chrom <- chrMap[yAnno$chromosome]
> yAnno <- yAnno[!is.na(yAnno$chrom), ]
```

Now, we construct a set of IRanges which represent our genes. Since the IRanges class only includes a start, a stop and a width, we need one IRanges object for each chromosome. We ignore strand, and create (essentially) a list with a component for each chromsome. We end up with a `RangesList` object.

Using IRanges to represent our annotation

```
> annoByChr <- split(yAnno, yAnno$chrom)
> annoIR <- lapply(annoByChr, function(d) {
+     IRanges(start = d$exon_chrom_start, end = d$exon_chrom_end
+ })
> annoIR <- do.call(RangesList, annoIR)
> annoIR

SimpleRangesList of length 16
$Scchr01
IRanges of length 132
       start    end width
[1]      335    649   315
[2]    80711  81952  1242
[3]      538    792   255
[4]   101566 105873  4308
[5]   113615 114616  1002
[6]   224554 224853   300
[7]    68717  69526   810
[8]   151099 151168    70
```

Using IRanges to represent our annotation

```
[9]   147596 151008  3413
...      ...    ...   ...
[124] 139221 139256    36
[125] 181135 181172    38
[126] 181205 181248    44
[127] 218540 219136   597
[128] 166268 166340    73
[129]  99306  99869   564
[130] 182516 182597    82
[131] 142369 142470   102
[132] 218131 218334   204

...
<15 more elements>
```

I could also have used the `GenomicData` class to represent the annotation, but the list format will be convenient later.

Representing our reads as IRanges

Next, we convert our aligned reads into IRanges as well. We need to do this separately for each lane of our data, so we write a function, and then use `lapply`. Again, we need a separate IRanges object for each chromosome, and we will arrange these into a RangesList. We end up with a `list` (4 lanes) of `RangesList` (the chromosomes).

```
> toRangesList <- function(aln) {
+     alignedByChr <- split(aln, chromosome(aln))
+     rngs <- lapply(alignedByChr, function(alnChr) {
+         IRanges(start = position(alnChr), width = 1)
+     })
+     do.call(RangesList, rngs)
+ }
> alnAsRanges <- lapply(yeastAligned, toRangesList)
> alnAsRanges[[1]]
```

Representing our reads as IRanges

```
SimpleRangesList of length 17
$Scchr01
IRanges of length 6675
         start    end width
[1]      63800  63800     1
[2]     142444 142444     1
[3]     166719 166719     1
[4]     184345 184345     1
[5]     140143 140143     1
[6]     142444 142444     1
[7]     148771 148771     1
[8]      40605  40605     1
[9]      71997  71997     1
...        ...    ...   ...
[6667]   67288  67288     1
[6668]  142388 142388     1
[6669]  126234 126234     1
[6670]  125797 125797     1
[6671]   72368  72368     1
```

Representing our reads as IRanges

```
[6672] 143587 143587      1
[6673] 142461 142461      1
[6674]  86297  86297      1
[6675]  64327  64327      1

...
<16 more elements>
```

Counting reads that fall in our genes

Finally, we use the as.table and findOverlaps in order to compute the counts within each gene. Unfortunately, this operation doesn't yet do the right thing with names, so we have to add the names back at the end; not elegant.

```
> exonNames <- split(yAnno$ensembl_exon_id, yAnno$chrom)
> oCounts <- sapply(alnAsRanges, function(aln) {
+     do.call(c, lapply(findOverlaps(annoIR, aln),
+         as.table))
+ })
> rownames(oCounts) <- do.call(c, exonNames)
> head(oCounts)
```

```
             mut_1_f mut_2_f wt_1_f wt_2_f
YAL069W.1          0       0      0      0
YAL034C.1         73      69    124    140
YAL068W-A.1        0       0      0      0
YAL024C.1         16      16     15     14
YAL020C.1         27      27     38     37
YAR070C.1          0       0      0      0
```

Counting reads that fall in our genes

If the assay had been stranded we could have added another layer to our lists, representing the two strands.
Note that the `oCounts` represents counts per exon. We could get counts per gene by using (for example) `tapply`.

Genominator overview

The Genominator package has methods for dealing with genomic data, including

- Import and manage/transform the data.
- Retrieving and summarizing data over annotation.
- Analysis tools for short read data.

In terms of short read data, we identify each read with its genomic location (of its 5' end). A consequence of this, is that information such as possible SNPs in the reads are discarded.
Right now, the package does not deal with paired-end data and reads mapped to junctions.
We (and collaborators) have been using the package internally for about one year and have completed several analyses using it.
We find it fast and flexible enough to use as a basis for custom analysis (at some level). We have analyzed datasets of 400M+ reads.
It has been on Bioconductor for about 1 week, so we are still ironing out some issues and adding capability.
Internally, Genominator uses an SQLite backend. This has certain consequences. One is that disk speed is suddenly *very* important.

Genominator overview

The main functionality of the package is to perform operations like

$$f(\text{ data}, \text{annotation})$$

Importing data

```
> library(Genominator)
> chrMap <- levels(chromosome(yeastAligned[[1]]))
> chrMap
 [1] "Scchr01" "Scchr02" "Scchr03" "Scchr04" "Scchr05"
 [6] "Scchr06" "Scchr07" "Scchr08" "Scchr09" "Scchr10"
[11] "Scchr11" "Scchr12" "Scchr13" "Scchr14" "Scchr15"
[16] "Scchr16" "Scmito"
> eData <- importFromAlignedReads(yeastAligned,
+     chrMap = chrMap, filename = "my.db", tablename = "raw",
+     overwrite = TRUE, deleteIntermediates = FALSE)
> head(eData)
```

Importing data

	chr	location	strand	mut_1_f	mut_2_f	wt_1_f	wt_2_f
1	1	3888	1	1	NA	NA	NA
2	1	3970	1	NA	NA	1	NA
3	1	3988	1	NA	1	NA	NA
4	1	4101	-1	NA	NA	NA	1
5	1	4242	1	1	NA	NA	NA
6	1	4271	-1	1	NA	NA	NA
7	1	4400	1	NA	NA	NA	1
8	1	4428	1	1	NA	NA	NA
9	1	4447	1	NA	NA	NA	1
10	1	4553	-1	NA	1	NA	NA

(last two arguments to `importFromAlignedReads` are usually not needed).
Internally, chromosomes (and strands) are stored as integers. The `chrMap` argument states how this conversion happens.

ExpData objects

ExpData objects are essentially a pointer to a table in a database (with a few additional twists), which exists externally to R. They are either created as a return value of some functions, or instantiated through their constructor:

```
> eData2 <- ExpData(db = "my.db", tablename = "mut_1_f",
+     mode = "w")
> head(eData2, 3)
```

```
  chr location strand mut_1_f
1   1     3888      1       1
2   1     4242      1       1
3   1     4271     -1       1
```

In general, they should not be saved.

ExpData, simple examples

```
> getRegion(eData, chr = 1, strand = 0, start = 10000,
+     end = 12000)

  chr location strand mut_1_f mut_2_f wt_1_f wt_2_f
1   1    10974     -1        1      NA     NA     NA
2   1    11562      1       NA       1     NA     NA

> laneCounts <- summarizeExpData(eData)
> laneCounts

mut_1_f mut_2_f  wt_1_f  wt_2_f
 423318  420848  410349  430264

> summarizeExpData(eData, fxs = "MAX")

mut_1_f mut_2_f  wt_1_f  wt_2_f
    231     191     109     107
```

`fxs` is limited to functions understood by SQLite.

ExpData and annotation

The real interest is in combining `ExpData` with annotation. In order to do so, we need to post-process the Ensembl annotation. We also drop some columns, mainly for display reasons.

```
> chrMap <- c(as.character(as.roman(1:16)), "MT",
+     "2-micron")
> yAnno$chr <- match(yAnno$chromosome, chrMap)
> yAnno$start <- yAnno$exon_chrom_start
> yAnno$end <- yAnno$exon_chrom_end
> yAnno <- yAnno[, c("ensembl_gene_id", "ensembl_exon_id",
+     "chr", "strand", "start", "end", "gene_biotype")]
> rownames(yAnno) <- yAnno$ensembl_exon_id
> head(yAnno, 2)
```

```
           ensembl_gene_id ensembl_exon_id chr strand  start
YHR055C.1         YHR055C       YHR055C.1   8     -1 214535
YPR161C.1         YPR161C       YPR161C.1  16     -1 864445
               end   gene_biotype
YHR055C.1 214720 protein_coding
YPR161C.1 866418 protein_coding
```

`ExpData` and annotation

In Genominator an annotation object has to have columns `chr`, `strand`, `start`, `end` as well as rownames. Each row corresponds to a genomic region (ie. a set of consecutive bases).

summarizeByAnnotation

This is a core function in Genominator.

```
> exonCounts <- summarizeByAnnotation(eData, yAnno,
+     ignoreStrand = TRUE)
> head(exonCounts, 3)
```

```
           mut_1_f mut_2_f wt_1_f wt_2_f
YHR055C.1        0       0      0      0
YPR161C.1       38      39     35     34
YOL138C.1       31      33     40     26
```

We can do other kind of summarization (limited to SQL commands)

```
> head(summarizeByAnnotation(eData, yAnno, bindAnno = TRUE,
+     fxs = "COUNT"), 3)
```

summarizeByAnnotation

```
           ensembl_gene_id ensembl_exon_id chr strand  start
YHR055C.1          YHR055C       YHR055C.1   8     -1 214535
YPR161C.1          YPR161C       YPR161C.1  16     -1 864445
YOL138C.1          YOL138C       YOL138C.1  15     -1  61325
              end    gene_biotype mut_1_f mut_2_f wt_1_f
YHR055C.1 214720 protein_coding       0       0      0
YPR161C.1 866418 protein_coding      11      15     13
YOL138C.1  65350 protein_coding      13      15     15
           wt_2_f
YHR055C.1      0
YPR161C.1     12
YOL138C.1     12
```

It is possible to aggregate exons into genes by (argument name might change) using a "meta identifier", which tells Genominator which regions belong in the same group.

```
> geneCounts <- summarizeByAnnotation(eData, yAnno,
+     ignoreStrand = TRUE, meta.id = "ensembl_gene_id")
> head(geneCounts, 3)
```

summarizeByAnnotation

	mut_1_f	mut_2_f	wt_1_f	wt_2_f
HRA1	7	14	4	12
LSR1	389	401	50	60
NME1	181	170	8	6

splitByAnnotation

The function summarizeByAnnotation lets the database handle most of the work. That is fast, and runs in bounded memory. But it is also inflexible.
We can also retrieve data in a convenient form using splitByAnnotation. Beware that the return object may be quite big.

```
> exonSplit <- splitByAnnotation(eData, yAnno[1:100,
+     ], ignoreStrand = TRUE)
> exonSplit2 <- splitByAnnotation(eData, yAnno[1:100,
+     ], expand = TRUE, ignoreStrand = TRUE)
> exonSplit3 <- splitByAnnotation(eData, yAnno[1:100,
+     ], expand = TRUE, addOverStrands = TRUE, ignoreStrand = TR
```

expand'ing can be very convenient, but the return object is very big.

applyMapped

After you have used splitByAnnotation you might want to use a function that depends both on the data and on the annotation (for example involving the exon length). A use case is

```
> countsPerBase <- applyMapped(exonSplit, yAnno,
+     FUN = function(map, anno) {
+         colSums(map, na.rm = TRUE)/(anno$end -
+             anno$start + 1)
+     })
```

Here, applyMapped takes care of matching the mapped reads with the right annotation region.

Union-intersection gene models

It is not always wise to rely directly on annotation. We have been advocating so-call "union-intersection" (UI) models for summarizing at the gene level. Essentially these models consist of all bases of a gene that are present in every transcript and not in any other gene (either strand). For *S. Cerevisia* this boils down to making sure that we don't have overlap with another gene on either strand (this is a really issue for this organism).
We provide a helper function for constructing these models. Because the interface to the functionality is not yet finalized, we need to access the function using `Genominator:::`.

```
> yAnnoUI <- Genominator:::makeUIgenes(yAnno, gene.id = "ensembl
+     transcript.id = "ensembl_transcript_id", verbose = TRUE)
> subset(yAnnoUI, ensembl_gene_id == "YAL005C")
> subset(yAnno, ensembl_gene_id == "YAL005C")

> save(yAnnoUI, file = "data/yAnnoUI.rda")
```

We will use these UI gene models in the following.

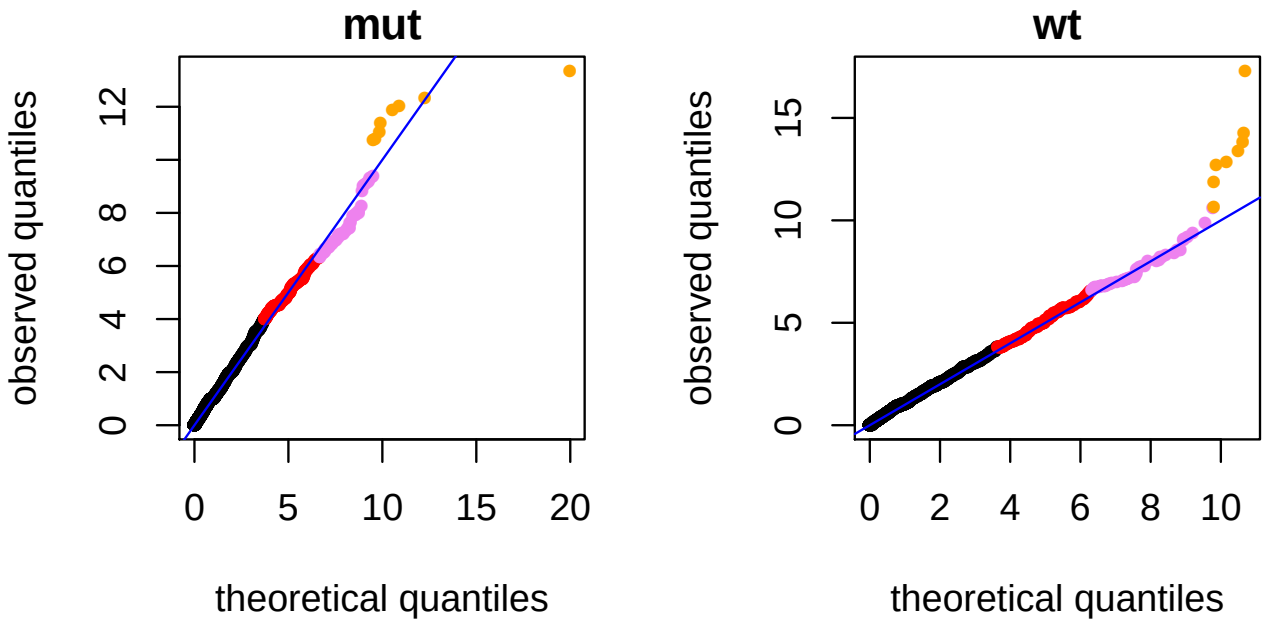
Analysis

We will now obtain the gene level counts and do a poison goodness-of-fit analysis as has been standard in several papers.
Note here that for each of the two samples ("wt" and "mut") we have 2 *technical* replicates in one lane each. "Technical" in this context means that the exact same "content" was deposited on the two lanes.

```
> geneCountsUI <- summarizeByAnnotation(eData, yAnnoUI,
+     ignoreStrand = TRUE, meta.id = "ensembl_gene_id")
```

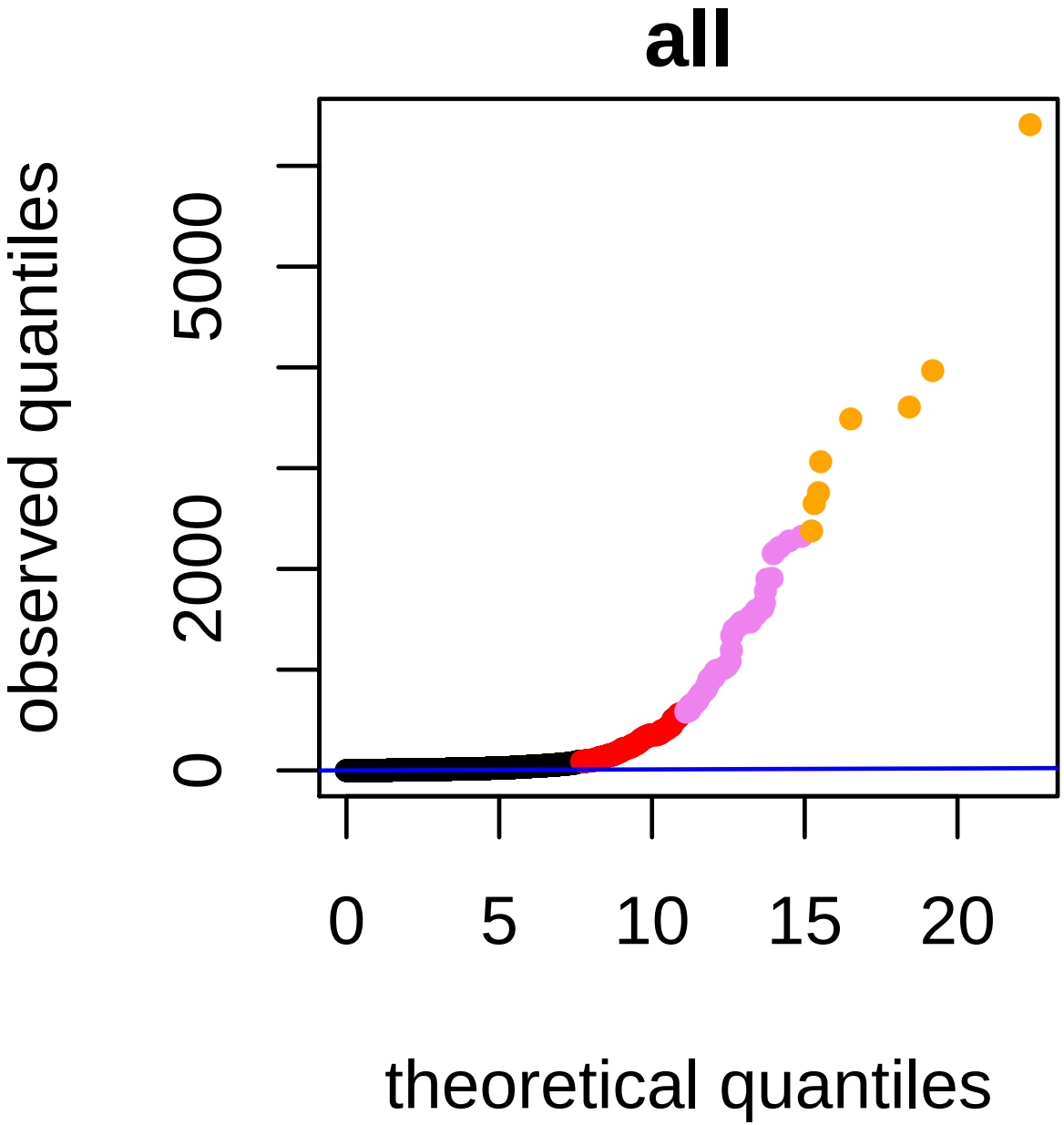
```
> plot(regionGoodnessOfFit(geneCountsUI, groups = rep(c("mut",
+     "wt"), times = c(2, 2))), chisq = TRUE)
```

Analysis




```
> plot(regionGoodnessOfFit(geneCountsUI, groups = rep("all",
+     4)), chisq = TRUE)
```

Analysis


all
observed quantiles
0
2000
5000
0
5
10
15
20
theoretical quantiles

Normalization

In the literature, it is standard to compute RPKMs. This is a form of normalization. It attempts to normalize between lanes taking the total sequencing effort into account and it attempts to normalize between genes taking the gene length into account.
It does not really handle the between gene normalization well (see Oshlack 2009).
Its attempt to normalize between lanes it is a form of *global* normalization: just dividing by the total number of reads (what the total number is, differs).
We have shown that in general it is much better to use upper-quartile normalization. This essentially uses the upper quartile of the read counts instead of the total number of reads.

```
> notZero <- which(rowSums(geneCountsUI) != 0)
> upper.quartiles <- apply(geneCountsUI[notZero,
+     ], 2, function(x) quantile(x, 0.75))
> uq.scaled <- upper.quartiles/sum(upper.quartiles) *
+     sum(laneCounts)
> laneCounts
```

Normalization

```
mut_1_f mut_2_f  wt_1_f  wt_2_f
 423318  420848  410349  430264
```

```
> uq.scaled
```

```
 mut_1_f  mut_2_f   wt_1_f   wt_2_f
453755.9 449554.5 378130.0 403338.6
```

```
> sum(laneCounts)
```

```
[1] 1684779
```

```
> sum(uq.scaled)
```

```
[1] 1684779
```

One we have the upper quartiles, the LR statistic is pretty standard (the call could be made faster with a bit of work)

Normalization

```
> groups <- factor(rep(c("mut", "wt"), times = c(2,
+     2)))
> pvalues <- apply(geneCountsUI[notZero, ], 1, function(y) {
+     fit <- glm(y ~ groups, family = poisson(),
+         offset = log(uq.scaled))
+     fit0 <- glm(y ~ 1, family = poisson(), offset = log(uq.sca
+     anova(fit0, fit, test = "Chisq")[2, 5]
+ })
```

The p-values could now be corrected for multiple testing using for example the mt.rawp2adjp function from the multtest package.

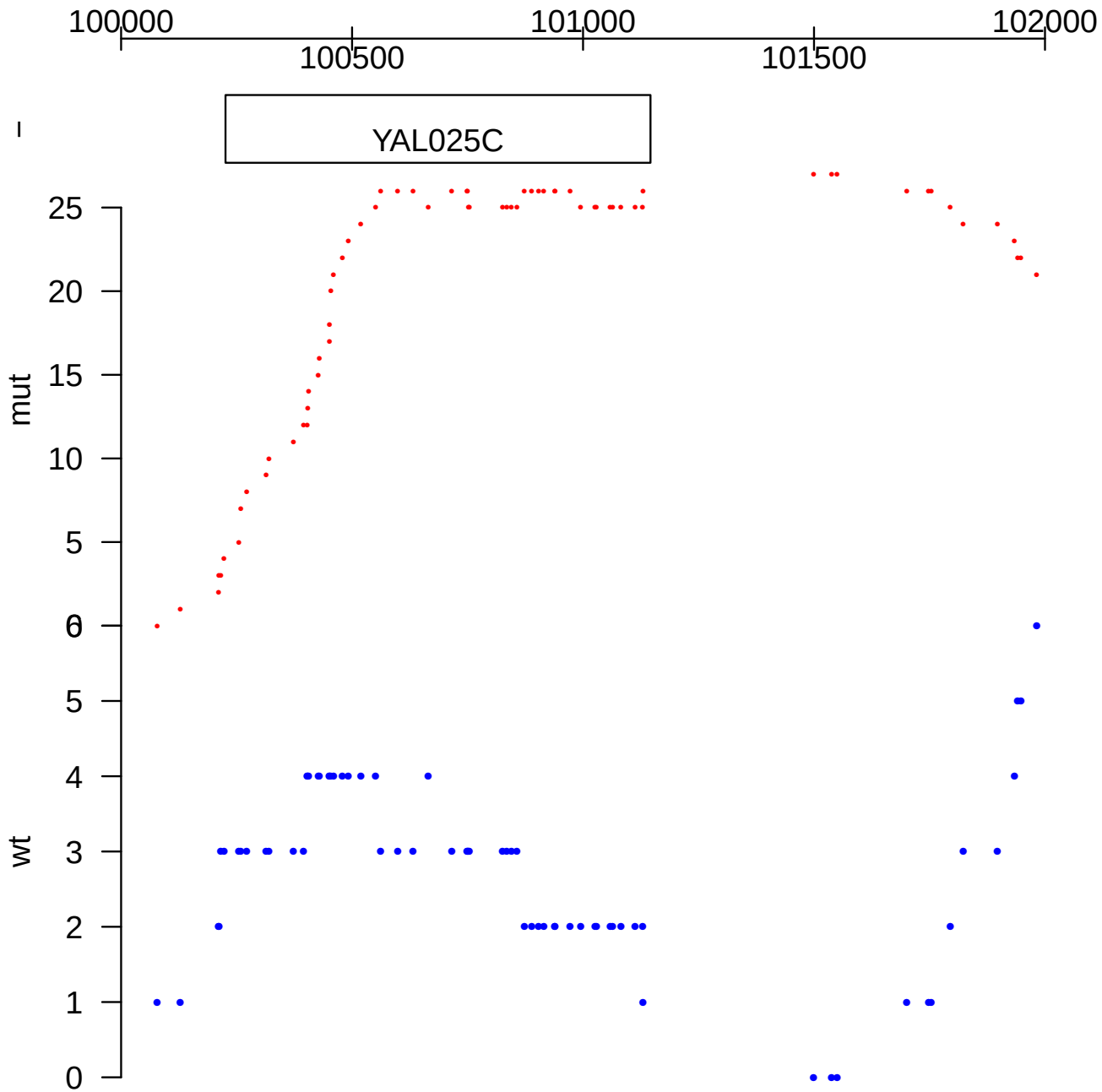
```
> library(multtest)
> adj <- mt.rawp2adjp(pvalues, proc = "BH")
> adj <- adj$adjp[order(adj$index), ]
> rownames(adj) <- names(pvalues)
```

Plotting

We are working on an interface to GenomeGraphs that retrieves data from the backend and plots it.

```
> annoFactory <- Genominator:::makeAnnoFactory.AnnoData(cbind(yA
+     feature = "gene"), featureColumnName = "feature",
+     groupColumnName = NULL, idColumnName = "ensembl_gene_id",
+     dp = DisplayPars(plotId = TRUE, idColor = "black"))
> rp <- Genominator:::makeRegionPlotter(list(mut = list(expData
+     what = "mut", fxs = Genominator:::makeConvolver(26),
+     dp = DisplayPars(lwd = 0.2, color = "red")),
+     wt = list(expData = eData2, what = "wt", fxs = Genominator
+         dp = DisplayPars(lwd = 0.3, color = "blue",
+             type = "p"))), annoFactory = annoFactory)
> rp(1, 1e+05, 102000)
```

Plotting


100000
101000
102000
100500
101500
YAL025C
mut
wt
25
20
15
10
5
0
5
4
3
2
1
0

Session Info

- R version 2.10.0 Patched (2009-11-17 r50465), `i386-apple-darwin9.8.0`
- Locale: `en_US.UTF-8/en_US.UTF-8/C/C/en_US.UTF-8/en_US.UTF-8`
- Base packages: base, datasets, graphics, grDevices, grid, methods, stats, utils
- Other packages: Biobase 2.6.0, biomaRt 2.2.0, Biostrings 2.14.5, BSgenome 1.14.1, DBI 0.2-4, GenomeGraphs 1.6.0, Genominator 1.1.1, IRanges 1.4.6, lattice 0.17-26, multtest 2.2.0, RSQLite 0.7-3, ShortRead 1.4.0, yeastRNASeq 0.0.2
- Loaded via a namespace (and not attached): hwriter 1.1, MASS 7.3-3, RCurl 1.3-0, splines 2.10.0, survival 2.35-7, tools 2.10.0, XML 2.6-0