Description of exonmap: simple analysis and annotation tools for Affymetrix exon arrays

MichałJ Okoniewski, Tim Yates, Crispin J Miller

October 13, 2008

Contents

1 Introduction

The package *exonmap* is intended to support various forms of data analysis for Affymetrix Exon microarrays. It includes a variety of routines for translating between probesets, exons, genes and transcripts, and makes use of a relational database (X:Map) to define these relationships for the current genome assembly. X:Map is built using Ensembl and Affymetrix annotation data, along with custom probeset to genome mappings.

Genome mappings were generated by searching probe sequences against the entire human (or mouse) genome and building database tables representing their hit locations and hit specificity. These are placed alongside data describing the relationships between exons, transcripts and genes. Most of this is hidden from the user; the package uses a series of functions (e.g. `probeset.to.exon`) that provide mappings between character vectors of database identifiers, while managing the underlying database queries internally.

The package provides graphics routines for plotting individual genes, and for colouring them by expression level or fold-change, and functions are also provided to link to the X:Map genome browser at `http://xmap.picr.man.ac.uk`.

The X:Map database and the exonmap package are described in more detail in:

"Okoniewski MJ, Yates T, Dibben S, Miller CJ. An annotation infrastructure for the analysis and interpretation of Affymetrix exon array data. Genome Biol. 2007;8(5):R79."

2 Initial processing of exon array data

exonmap makes use of the *affy* package; a basic understanding of the library and its vignette is a good idea. We also assume that the reader knows how the Affymetrix system works. A good starting point is the Affymetrix MAS manual, which can be found at http://www.affymetrix.com.

Although this package is designed primarily to support annotation, it does contain some basic utility functions to make it easy to load and begin to explore exon array data. The following section exists simply to provide a quick route to a list of differentially expressed probesets; alternative strategies are of course possible, and you may choose to skip this section and use your own approach.

3 Reading in data and generating expression calls

The first thing you need to do is to get R to use the *exonmap* package by telling it to load the library:

```
> library(exonmap)
> library(affy)
```

R needs to know about the replicates in your experiment, so we must also load some descriptive data that says which arrays were replicates and also something about the different experimental conditions you were testing. This means that *exonmap* needs *two things*:

1. your .CEL files, and
2. a white-space delimited file describing the samples that went on them.

By default, this file is called *covdesc*. The first column should have no header, and contains the names of the .CEL files you want to process. Each remaining column is used to describe something in the experiment you want to study. For example you might have a set of chips produced by treating a cell line with two drugs. Your *covdesc* file might look like something like this:

	treatment
ctrl1.cel	n
ctrl2.cel	n
ctrl3.cel	n
a1.cel	a
a2.cel	a
a3.cel	a
b1.cel	b
b2.cel	b
b3.cel	b
ab1.cel	a.b
ab2.cel	a.b

This is similar to the approach taken by *simpleaffy*.

The easiest way to get going is to:

1. Create a directory, move all the relevant *CEL* files to that directory
2. Create a *covdesc* file and put it in the same directory
3. If using linux/unix, start R in that directory.
4. If using the Rgui for Microsoft Windows make sure your working directory contains the *Cel* files (use "File -> Change Dir" menu item).
5. Load the library.

Exon array CEL files may be read using the function `read.exon`. In all cases, an experiment description file (covdesc) must be present.

In addition, a CDF metadata package must be specified. Versions of CDF metadata for mouse and human exon arrays can be downloaded from http://xmap.picr.man.ac.uk. The CDF metadata cannot include control or backround probesets if you are going to process it with RMA or plier.

For example, to get started, you might run something like:

```
> raw.data <- read.exon()
> if (exists(raw.data)) {
+     raw.data@cdfName <- "exon.pmcdf"
+     x.rma <- rma(raw.data)
+ }
```

The CDF files *exon.pmcdf* (for the Human Exon 1.0ST array) and *mouseexonpmcdf* (for the Mouse Exon 1.0 ST array), available from http://bioinformatics.picr.man.ac.uk have been prepared by processing the ASCII CDF files from Affymetrix using the *(*makecdfenv) and *(*altcdfenvs) packages. They include PM probes only. Probesets representing genomic and antigenomic background and control probesets have also been removed.

4 Pairwise comparison of expression data

The function `pc` provides fast pairwise comparisons for `ExpressionSet` objects.

```
> data(exonmap)
> pc.exonmap <- pc(x.rma, "group", c("a", "b"))
```

`pc` produces an object of class PC that has two slots: *fc*, for the log2 fold change and *tt* containing a t-test p-value. For the purpose of this vignette, we use these to select significant probesets, although other more in-depth approaches are of course possible. For example:

```
> sigs <- names(fc(pc.exonmap))[(abs(fc(pc.exonmap)) > 1) & (tt(pc.exonmap) <
+     1e-04)]
> length(sigs)
```

```
[1] 31
```

5 Connecting to the database

The `xmapConnect()` function, called with no parameters, will offer a list of available databases, allowing you to select the appropriate one. Alternatively, the name of the database can be provided directly, if known (e.g. *xmapConnect("human")*).

6 Translation routines for genes, transcripts, exons and probesets

The X:Map database can be queried in a number of ways using translation functions. All of them have the form X.to.Y, where X and Y may be a vector of gene, transcript,

exon or probeset identifiers. See, `?mappings` for more details. All the functions produce, by default, a vector of identifiers. More information can be generated by setting the parameter *as.vector* to *FALSE*, in which case a data frame is returned. If *unique* is *FALSE*, duplicates are removed before the result is returned.

```
> xmapConnect("human")
> sig.exons <- probeset.to.exon(sigs)
> length(sig.exons)

[1] 20

> sig.transcripts <- probeset.to.transcript(sigs)
> length(sig.transcripts)

[1] 12

> sig.genes <- probeset.to.gene(sigs)
> length(sig.genes)

[1] 7

> sig.exons.data.frame <- probeset.to.exon(sigs, as.vector = FALSE)
```

(These numbers are so small because there are only 7 genes represented in the example dataset).

Each of these funtions, by default, removes probesets where one or more probes matches the genome at multiple sites (see the section 'Probeset filtering', below, for more details). Setting *mt.rm=FALSE* prevents this additional filtering.

7 More details

`exon.details`,`transcript.details` and `gene.details` can all be used to extract detailed annotation, given the appropriate set of identifiers.

```
> exon.details(sig.exons)
> transcript.details(sig.transcripts)
> gene.details(sig.genes)
```

Note that there is no corresponding function, *probeset.details*. This is because there is relatively little useful information to provide about a probeset since it is essentially a name used to group a set of probes together. The function *probeset.to.probe(v,as.vector=FALSE)* can be used to find the locations of each of the probeset's probes' genome hits.

8 Finding items at specified locations in the genome

A set of functions of the form *X.in.range* can be used to find all of the probeset between two points. For example:

```
> gds <- gene.details(sig.genes)
> x1 <- gds$seq_region_start
> x2 <- gds$seq_region_end
> chr <- gds$name
> strand <- gds$seq_region_strand
> ps <- mapply(probesets.in.range, x1, x2, strand, chr)
```

gives us back a list of character vectors, one for each gene, containing the probesets within that gene's region.

```
> length(ps)

[1] 7

> class(ps)

[1] "list"

> sapply(ps, length)

[1] 523  73  27 154 187 271 267

> ps[[1]][1:10]

 [1] "3102373" "3102369" "3102370" "3102371" "3889483" "3767204" "3431978"
 [8] "3102374" "3973017" "3818668"
```

9 Genes,exons and probesets in a single query

Often we want to find all the probesets hitting a gene's exons. The function *gene.to.exon.probeset* is designed to do this quickly (it is implemented as a stored procedure on the database server).

```
> gene.mappings <- gene.to.exon.probeset(sig.genes)
```

This gives us back a data.frame - and these are the first 10 rows:

```
> gene.mappings[1:10, ]
```

```
              gene            exon probeset_id probeset_name probe_count
1  ENSG00000137573 ENSE00000697174      635026       3102398           4
2  ENSG00000137573 ENSE00000697198      635033       3102405           4
3  ENSG00000137573 ENSE00000697200      635040       3102412           4
4  ENSG00000137573 ENSE00000980794      635058       3102430           4
5  ENSG00000137573 ENSE00000980794      635059       3102431           4
6  ENSG00000137573 ENSE00000980797      635065       3102437           4
7  ENSG00000137573 ENSE00000980803      635073       3102445           4
8  ENSG00000137573 ENSE00001191942      635088       3102460           4
9  ENSG00000137573 ENSE00001191942      635089       3102461           4
11 ENSG00000137573 ENSE00001191942      635091       3102463           4
```

The function `gene.to.exon.probeset.exprs` does the same, but also takes an `ExpressionSet` as an argument. It extracts the relevant rows from this and prepends ito the annotation data to generate a table containing both expression data and annotation.

10 Probeset filtering

Probesets can be filtered according to the number and quality of their matches to the genome. Match statistics can be displayed with `probeset.stats`.

```
> probeset.stats(ps[[1]][1:5])
```

```
  probeset_id probeset hitScore exonScore geneScore
1      635001  3102373        1         1         1
2      634998  3102369        1         0         1
3      634999  3102370        1         0         1
4      635000  3102371        1         0         1
5     1277261  3889483      101         0        25
```

The hit, exon and gene scores are calculated using all the probes in the probesets (usually 4), by finding how many times they match to the genome, to exons, and to genes - and then multiplying the minimum value for the probe within a probeset with the maximum. Thus the first probeset in the example is "exonic" as it matches the genome 1 time and only matches 1 gene and 1 exon. The second one is "intronic" because not all its probes hit an exon and the fifth one is a "multitarget" probeset because it includes at least one probe that matches many locations in the genome.

These four types of probesets can be selected or excluded from a probeset list using the `select.probewise` and `exclude.probewise` functions. For example, to find probesets that hit within genes, but outside regions annotated as exons by ensembl:

```
> select.probewise(sigs, filter = "intronic")
```

```
[1] "3388403"
```

In a similar way, a probeset list can be filtered to get rid of multiply targeting probesets (i.e. those annotated by X:Map to hit in more than one place on the genome):

```
> sigs.nomt <- exclude.probewise(sigs, filter = "multitarget")
```

The functions `is.exonic`,`is.intronic`,`is.intergenic` and `is.multitarget` provide a way of testing each probeset (rather than filtering the list) and the functions `exonic`,`intronic`,`intergenic` and `multitarget` also exist (for consistency) and provide exactly the same functionality as `select.probewise` and `exclude.probewise`.

```
> is.intronic(sigs)
```

```
3102398 3102391 3102439 3102381 3102419 3102412 3102447 3102405 2906895 2906891
  FALSE   FALSE   FALSE   FALSE   FALSE   FALSE   FALSE   FALSE   FALSE   FALSE
3933543 3933539 3933542 3933538 2974595 2974598 2974597 2974596 3945601 3388376
  FALSE   FALSE   FALSE   FALSE   FALSE   FALSE   FALSE   FALSE   FALSE   FALSE
3388408 3388379 3388382 3388414 3388375 3388407 3388403 3388393 3388380 3388412
  FALSE   FALSE   FALSE   FALSE   FALSE   FALSE    TRUE   FALSE   FALSE   FALSE
3388383
  FALSE
```

```
> intronic(sigs)
```

```
[1] "3388403"
```

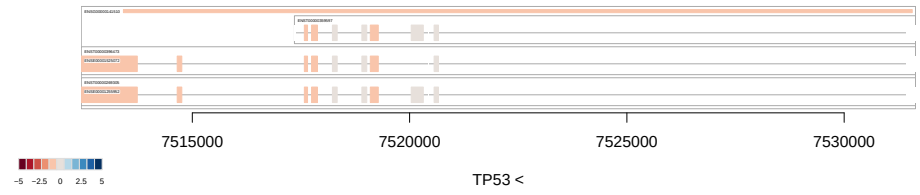
11 Plotting genes of interest

plotGene provides plots of the transcript and exon structure of a given gene, coloured by expression data.

In order to compute the values that go into the plot, one or more groups is supplied as a list, using the parameter *gps* (note that this has changed slightly since the last release of the package). Each element in *gps* is a vector of indices into the expression data in *data*.

So, for example,

```
> plotGene("ENSG00000141510", x.rma, gps = list(1:3, 4:6), type = "mean-fc")
```



will compute the fold changes between arrays 1:3 and 4:6.

Alternatively, the arrays to be compared can be defined using the *ExpressionSet*'s annotation data, in the same way as `pc`, so:

```
> plotGene("ENSG00000141510", x.rma, gps = c("a", "b"), group = "group",
+     type = "mean-fc")
```

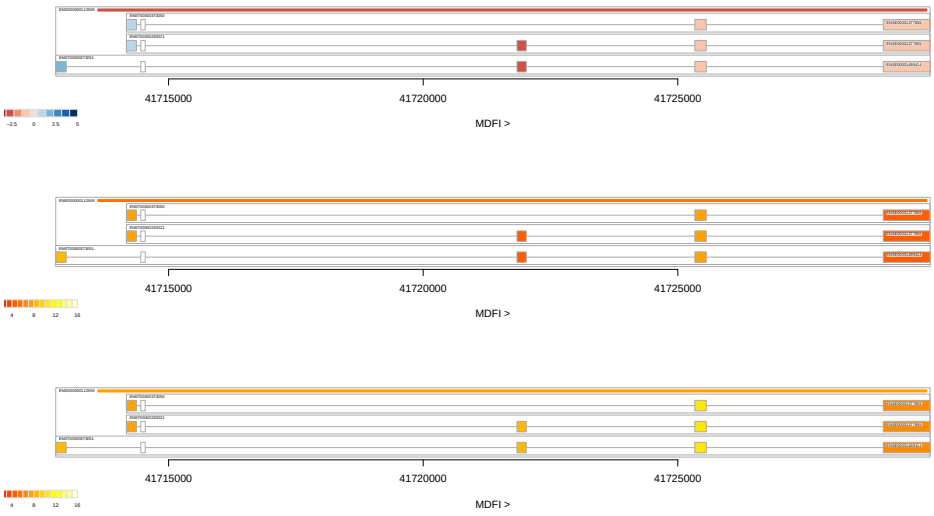
will also work.

All well behaving exon-matching probesets are found, and the mean value used to colour the plot. The process is repeated for each transcript and each exon. Transcripts aren't coloured, and the mean value for the gene is shown as a bar running across the top of the plot.

The approach to averaging can be changed and, raw intensities can plotted instead (see `?plotGene` for more details). It is also possible to pre-scale the colouring to the average for the gene (averaged over all the exons), so that data is coloured relative to the gene-average (using the parameter *scale.to.gene*).

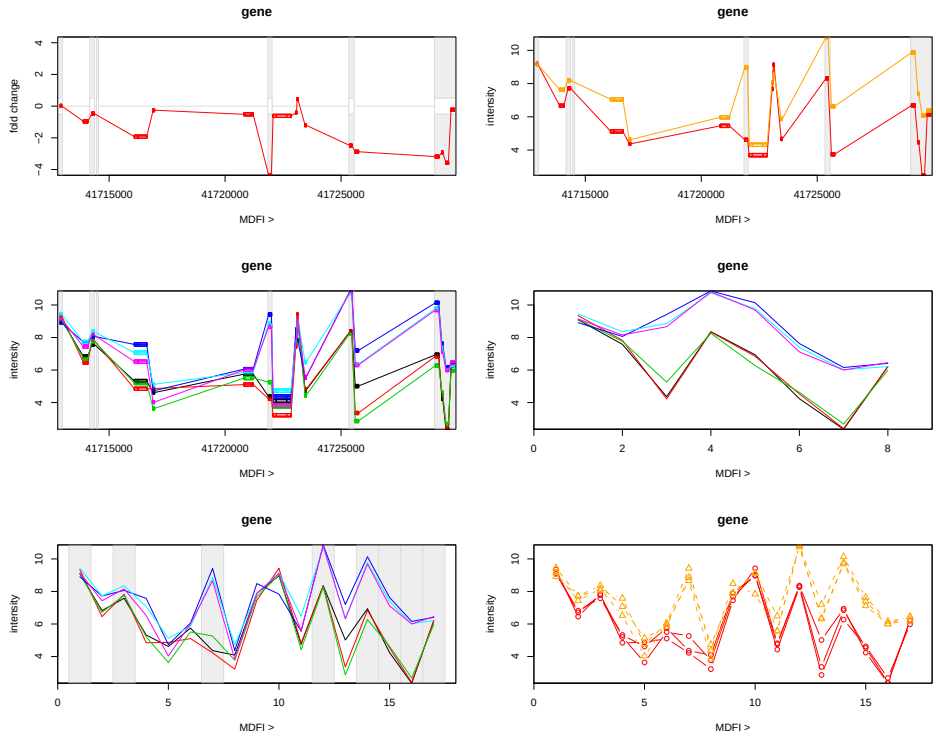
By default, exons with no matching probesets (following filtering for multi-targeting probesets) are coloured white.

```
> par(mfrow = c(3, 1))
> plotGene("ENSG00000112559", x.rma, gps = list(1:3, 4:6), type = "mean-fc",
+     scale.to.gene = TRUE)
> plotGene("ENSG00000112559", x.rma, gps = list(1:3), type = "mean-int",
+     col = heat.colors(16))
> plotGene("ENSG00000112559", x.rma, gps = list(4:6), type = "mean-int",
+     col = heat.colors(16))
```



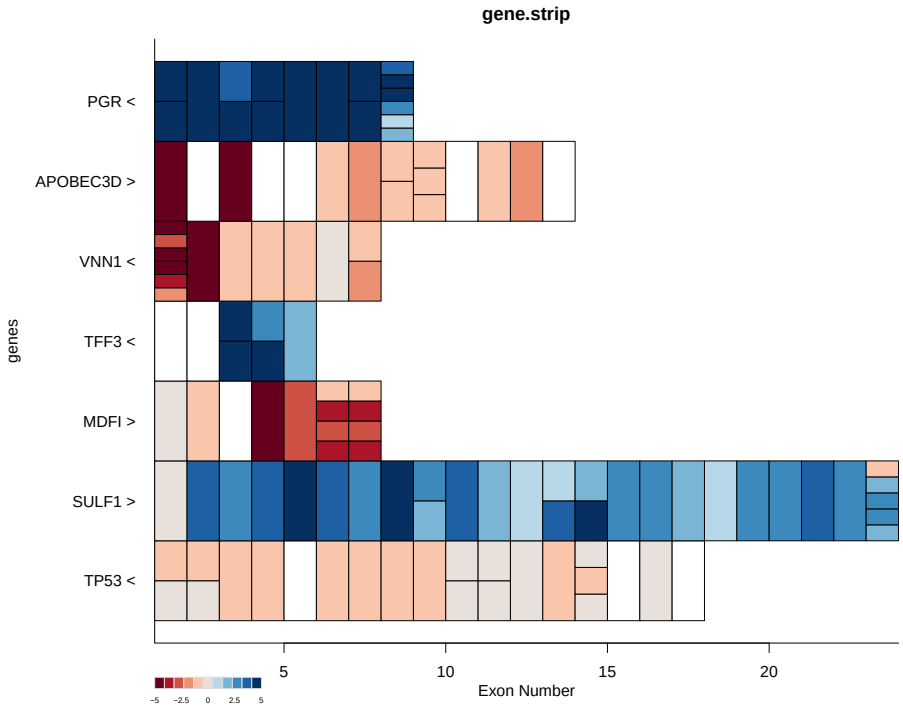
Another utility to visualize the expression of a gene is `gene.graph`. It creates a line-plot for a specified gene, including intronic probesets. For example:

```
> par(mfrow = c(3, 2))
> gene.graph("ENSG00000112559", x.rma, gps = list(1:3, 4:6), type = "mean-fc",
+     gp.col = "red")
> gene.graph("ENSG00000112559", x.rma, gps = list(1:3, 4:6), type = "mean-int",
+     gp.col = c("red", "orange"))
> gene.graph("ENSG00000112559", x.rma, gps = list(1, 2, 3, 4, 5,
+     6), type = "mean-int", gp.col = 1:6)
> gene.graph("ENSG00000112559", x.rma, gps = list(1, 2, 3, 4, 5,
+     6), type = "mean-int", gp.col = 1:6, by.order = TRUE)
> gene.graph("ENSG00000112559", x.rma, gps = list(1, 2, 3, 4, 5,
+     6), type = "mean-int", gp.col = 1:6, by.order = TRUE, show.introns = TRUE)
> gene.graph("ENSG00000112559", x.rma, gps = list(1, 2, 3, 4, 5,
+     6), type = "mean-int", gp.col = c(rep("red", 3), rep("orange",
+     3)), gp.pch = c(1, 1, 1, 2, 2, 2), gp.lty = c(1, 1, 1, 2,
+     2, 2), by.order = TRUE, show.introns = TRUE, exon.bg.col = NA)
```



Heatmap style plots can also be generated with the function `gene.strip`.

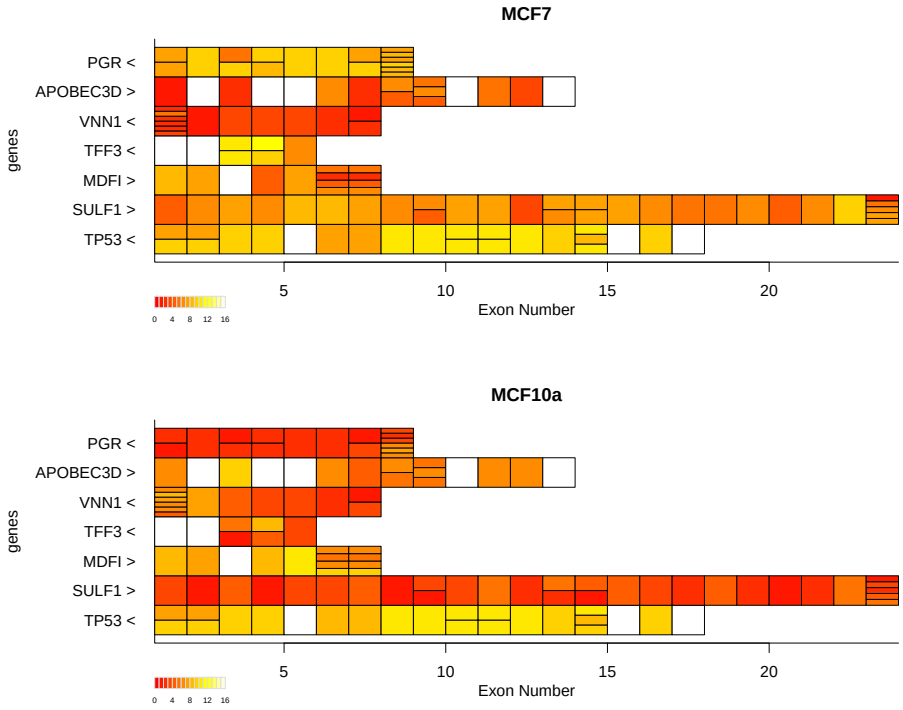
```
> all.genes <- probeset.to.gene(featureNames(x.rma))
> gene.strip(all.genes, x.rma, list(1:3, 4:6), type = "mean-fc")
```



Here, each row corresponds to a gene, and each exon is plotted in exon-order along the X axis. The plot is coloured as before; exons for which a uniquely matching probeset cannot be found are, by default, coloured white. When multiple probesets hit the same exon, these are stacked vertically within that exon's rectangle.

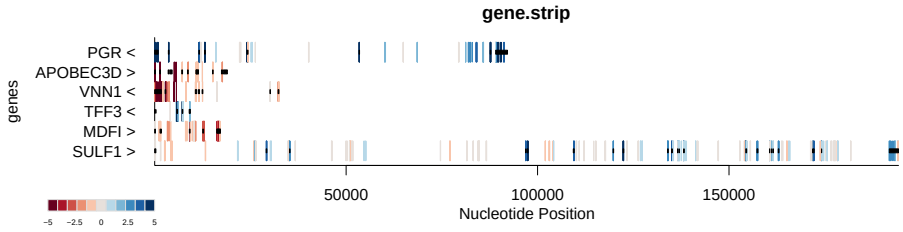
Alternatively, plots could be coloured by intensity:

```
> par(mfcol = c(2, 1))
> gene.strip(all.genes, x.rma, list(1:3), type = "median-int",
+     col = heat.colors(16), main = "MCF7")
> gene.strip(all.genes, x.rma, list(4:6), type = "median-int",
+     col = heat.colors(16), main = "MCF10a")
```



The parameter `show.introns` can be used to change the plotting behaviour so that introns are shown, and exons are positioned relative to their nucleotide position within the gene.

```
> gene.strip(sig.genes, x.rma, list(1:3, 4:6), type = "mean-fc",
+     show.introns = TRUE)
```



12 Splicing index and splicing ANOVA

Splicing index and splicing ANOVA have also been implemented as described in the Affymetrix white paper: "Alternative transcript analysis methods for exon arrays".

The splicing index gives a measure of the difference in expression level for each probeset in a gene between two sets of arrays, relative to the gene-level average in each set. This is calculated only for those probesets that are defined as exon targeting and

non-multitargetted (See *select.probewise* and *exclude.probewise* for more details of how this filtering is performed.

As before, the two sets of arrays to comapare can be specified either as a list: *...,gps=list(1:3,4:6),...*, or by using the annotation data: *...,gps=c('a','b'),group='groups',....*

The implementation also calculates a *p.value* and *t.statistic* for each probeset; these are returned alongside the splicing index, and the overall gene level average is also provided.

By default, the splicing index is calculated using the mean across genes and samples, specifing *median.gene=TRUE* will use the median instead. It is calculated using the unlogged data, unless *unlogged=FALSE*. This only affects the internal calculations - values in x are always assumed to be logged, and the splicing index is always returned on the log2 scale.

```
> si <- si(x.rma, c("ENSG00000141510", "ENSG00000082175"), group = "group",
+     gps = c("a", "b"))
```

`splanova` is an implementation of the MIDAS approach suggested by Affymetrix, also described in the above whitepaper. It produces an object with F-values and significance of alternative splicing, for each probeset and treatment in a multi-treatment experiment.

13 ESTs, in silico predictions etc...

Mappings can also be performed against the EST and prediction subsets of Ensembl. For this the other features database must be installed as well (see the Installation Instructions for more information). The subset to search against is defined by setting the *subset* parameter of the appropiate `X.to.Y` function:

```
> ps <- probesets.in.range(7497600, 7516800, -1, "17")
> ps <- multitarget(ps, exclude = TRUE)
> exonic(ps)

[1] "3743908" "3743909" "3743912"

> intronic(ps)

[1] "3743910" "3743911"

> probeset.to.exon(ps, subset = "core")

[1] "ENSE00001255952" "ENSE00001525072" "ENSE00001255878" "ENSE00001525080"

> probeset.to.exon(ps, subset = "est")
```

```
[1] "ENSESTEE00000225869" "ENSESTEE00000225868"
> probeset.to.exon(ps, subset = "prediction")
[1] 143942 143940
> probeset.to.transcript(ps, subset = "core")
[1] "ENST00000269305" "ENST00000396473"
> probeset.to.transcript(ps, subset = "est")
[1] "ENSESTTT00000065754" "ENSESTTT00000065756" "ENSESTTT00000065757"
> probeset.to.transcript(ps, subset = "prediction")
[1] 25802
> probeset.to.gene(ps, subset = "core")
[1] "ENSG00000141510"
> probeset.to.gene(ps, subset = "est")
[1] "ENSESTGG00000027012" "ENSESTGG00000027014"
```

Note that `probeset.to.gene` doesn't provide a mapping for the "prediction" subset, since it stores transcript, not gene predictions.

Predictions are also slightly different from the core and est subsections because they are represented by IDs, not strings (see the result of *probeset.to.transcript(ps,subset="prediction")*, for example.

14 Putting it all together

A typical workflow for analysing exon array data using exonmap might be to first identify differentially expressed probesets, using whatever standard techniques might be applied to previous generation arrays - for example, preprocessing using RMA, generating fold changes and filtering using an FDR moderated threshold.

This yields a set of significant probesets. Exonmap can then be used to map these to the exons they target - yielding a list of exons in which at least one probeset is significant.

These exons can then be mapped to genes, and visualised individually, or filtered using global methods such as the splicing index. Global plots can be generated, and interesting genes pursued further via X:Map.

This translates into relatively little code:- we pick it up after initial preprocessing, and in this example, use only the minimal data set supplied with the package. i.e. *x.rma* contains (a small set) of normalized and preprocessed expression data for two cell lines, *"a"* and *"b"*. We also do a trivial filtering based on fold change and unadjusted p-value, rather than something more sophisticated using, for example *limma*.

Then we:

1. Remove mulittarget probesets
2. Map to exons
3. Map to genes
4. Generate splicing index
5. Partition by average fold change for the genes
6. plot, ordered by splicing index
7. repeat plots with genes scaled to gene level average

```
> pData(x.rma)
```

```
                 sample group
ex1MCF7_r1.CEL        1     a
ex1MCF7_r2.CEL        2     a
ex1MCF7_r3.CEL        3     a
ex2MCF10A_r1.CEL      4     b
ex2MCF10A_r2.CEL      5     b
ex2MCF10A_r3.CEL      6     b
```

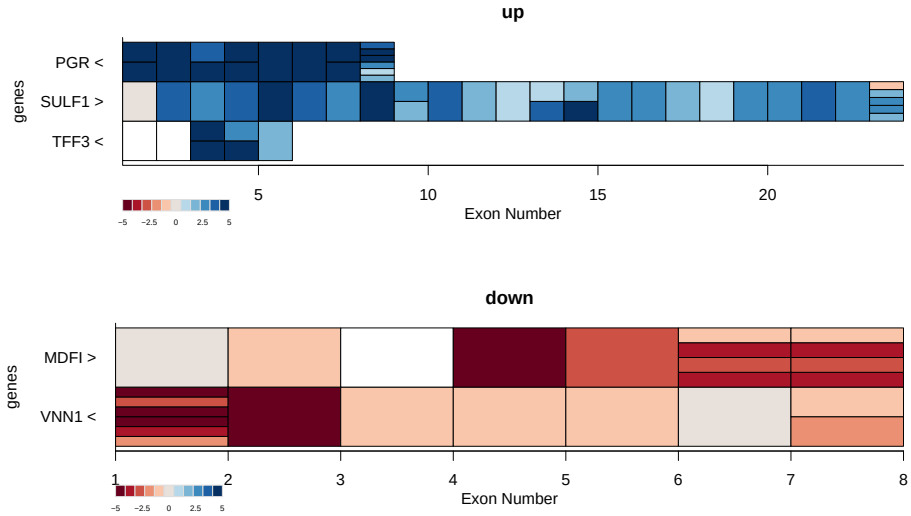
```
> pc.exonmap <- pc(x.rma, "group", c("a", "b"))
> sigs <- names(fc(pc.exonmap))[(abs(fc(pc.exonmap)) > 1) & (tt(pc.exonmap) <
+     1e-04)]
> sigs <- exonic(sigs)
> e <- probeset.to.exon(sigs)
> g <- exon.to.gene(e)
> r <- si(x.rma, g, gps = list(1:3, 4:6))
> g <- intersect(g, names(r))
> r <- r[g]
```

At this point we have a list of genes containing at least one significant exon taragetting probeset, and their splicing index.

We can then calculate the maximum splicing index and retrieve the average fold change for each gene. These are then used to partition the dataset into (on average) up- and down- regulated genes. We can sort each set by the maximum splicing index and plot separate gene.strip plots for each set.

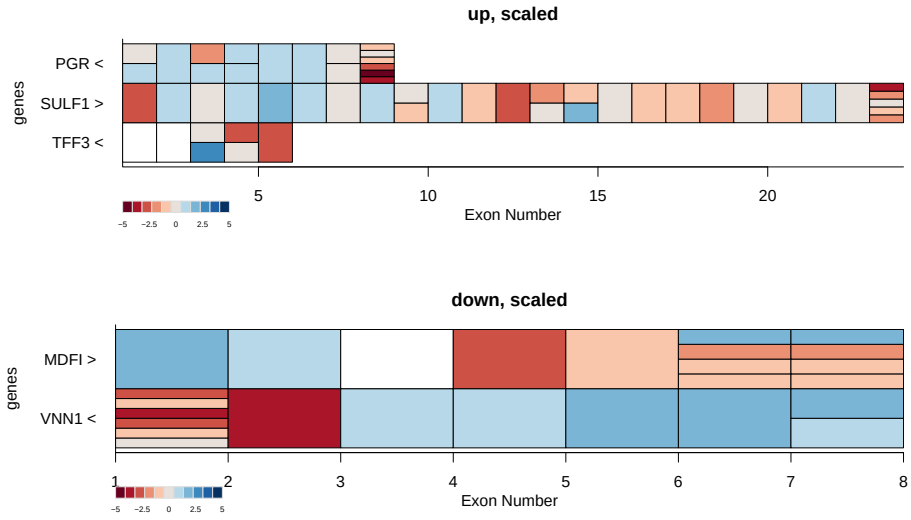
```
> max.si <- sapply(r, function(a) {
+     max(abs(a$si))
+ })
> gene.av <- sapply(r, function(a) {
```

```
+     max(a$gene.av)
+ })
> up <- gene.av > 0
> par(mfrow = c(2, 1))
> o <- order(max.si[up], decreasing = TRUE)
> gene.strip(g[up][o], x.rma, list(1:3, 4:6), type = "mean-fc",
+     main = "up")
> o <- order(max.si[!up], decreasing = TRUE)
> gene.strip(g[!up][o], x.rma, list(1:3, 4:6), type = "mean-fc",
+     main = "down")
```



It can also be useful to compare these plots to those produced when each gene is scaled to the gene-average fold change:

```
> par(mfrow = c(2, 1))
> o <- order(max.si[up], decreasing = TRUE)
> gene.strip(g[up][o], x.rma, list(1:3, 4:6), type = "mean-fc",
+     main = "up, scaled", scale.to.gene = TRUE)
> o <- order(max.si[!up], decreasing = TRUE)
> gene.strip(g[!up][o], x.rma, list(1:3, 4:6), type = "mean-fc",
+     main = "down, scaled", scale.to.gene = TRUE)
```


up, scaled
genes
PGR <
SULF1 >
TFF3 <
5
10
15
20
Exon Number
−5
−2.5
0
2.5
5

down, scaled
genes
MDFI >
VNN1 <
1
2
3
4
5
6
7
8
Exon Number
−5
−2.5
0
2.5
5