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

MichałJ Okoniewski, Tim Yates, Crispin J Miller

June 20, 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 exon, transcript and gene relationships. Most of this is hidden from the user; the package uses a series of functions (e.g. `probeset.to.exon`) that manage the underlying database queries.

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 web-based front end, at `http://xmap.picr.man.ac.uk`.

The X:MAP database and the exonmap package are described in more detail in (add citations).

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. If not, a brief introduction can be found at http://bioinf.picr.man.ac.uk/; a more detailed description is in the Affymetrix MAS manual at http://www.affymetrix.com.

Although this package is 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 Human Exon 1.0ST array and *mouseexonpmcdf* for Mouse Exon 1.0 ST arrays, available from http://bioinformatics.picr.man.ac.uk have been prepared by processing the ASCII CDF files from Affymetrix, using the *(*makecdfenv) and *(*altcdfenvs). 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 `exprSet` 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 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, for example, `?probeset.to.gene` for more details. All the functions produce, by default, a vector of the identifiers resulting from the specified mapping. More information can be generated by setting the parameter *vector.out* to *FALSE*, in which case, a data frame is returned. If *unique* is true, duplicates are removed before the result is returned.

```
> xmapDatabase("Human")
```

```
done.
```

```
> 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
```

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

6 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)
```

We can also find all of the probesets between two points by using *probesets.in.range*

```
> gds <- gene.details(sig.genes)
> x1 <- gds$seq_region_start
> x2 <- gds$seq_region_end
> chr <- gds$name
> strand <- gds$seq_region_strand
> ps <- apply(cbind(x1, x2, strand, chr), 1, function(a) {
+     probesets.in.range(a[1], a[2], a[3], a[4])
+ })
```

This 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)
```

```
ENSG00000137573 ENSG00000112559 ENSG00000160180 ENSG00000112299 ENSG00000198904
            523              73              27             154             187
ENSG00000128394 ENSG00000082175
            271             267
```

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).

```
> ps2 <- lapply(sig.genes, gene.to.exon.probeset)
```

This gives us back a list of dataframes, one for each gene:

```
> ps2[[1]]
```

```
              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
12 ENSG00000137573 ENSE00001191942      635092       3102464           4
13 ENSG00000137573 ENSE00001191942      635093       3102465           4
14 ENSG00000137573 ENSE00001191955      635071       3102443           4
15 ENSG00000137573 ENSE00001191962      635067       3102439           4
16 ENSG00000137573 ENSE00001191968      635066       3102438           4
17 ENSG00000137573 ENSE00001191977      635063       3102435           4
18 ENSG00000137573 ENSE00001191981      635062       3102434           4
19 ENSG00000137573 ENSE00001191988      635047       3102419           4
20 ENSG00000137573 ENSE00001191988      635048       3102420           4
21 ENSG00000137573 ENSE00001191991      635045       3102417           4
22 ENSG00000137573 ENSE00001191996      635042       3102414           4
23 ENSG00000137573 ENSE00001192007      635038       3102410           4
24 ENSG00000137573 ENSE00001192007      635039       3102411           4
25 ENSG00000137573 ENSE00001192013      635032       3102404           4
26 ENSG00000137573 ENSE00001226516      635075       3102447           4
28 ENSG00000137573 ENSE00001304123      635021       3102393           4
29 ENSG00000137573 ENSE00001307988      635014       3102386           4
30 ENSG00000137573 ENSE00001324480      635019       3102391           4
31 ENSG00000137573 ENSE00001328622      635009       3102381           4
32 ENSG00000137573 ENSE00001364471      635001       3102373           4
```

7 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
```

```
11       634998   3102369         1          0          1
12       634999   3102370         1          0          1
13       635000   3102371         1          0          1
14      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):

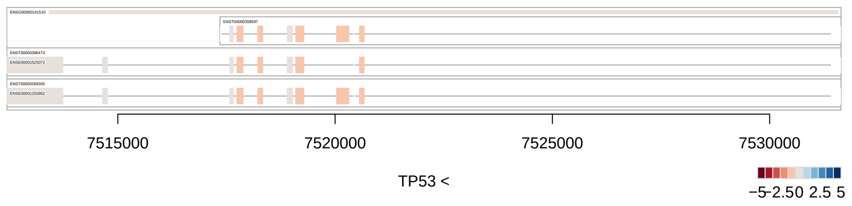
```
> sigs.nomt <- exclude.probewise(sigs, filter = "multitarget")
```

8 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 indexes into the expression data in *data*. So, for example,

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

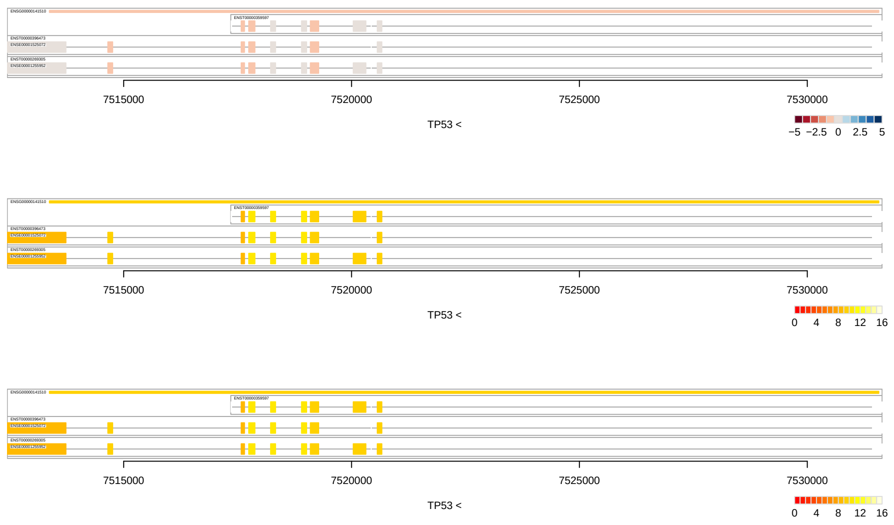


will compute the fold changes between arrays 1:3 and 4:6. All well behaving exon-matching probesets are found, and the mean value used to colour the plot. The process is repeated for each transcript and each exon. Transcripts aren't coloured, and the mean value for the gene is shown as a bar running 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`).

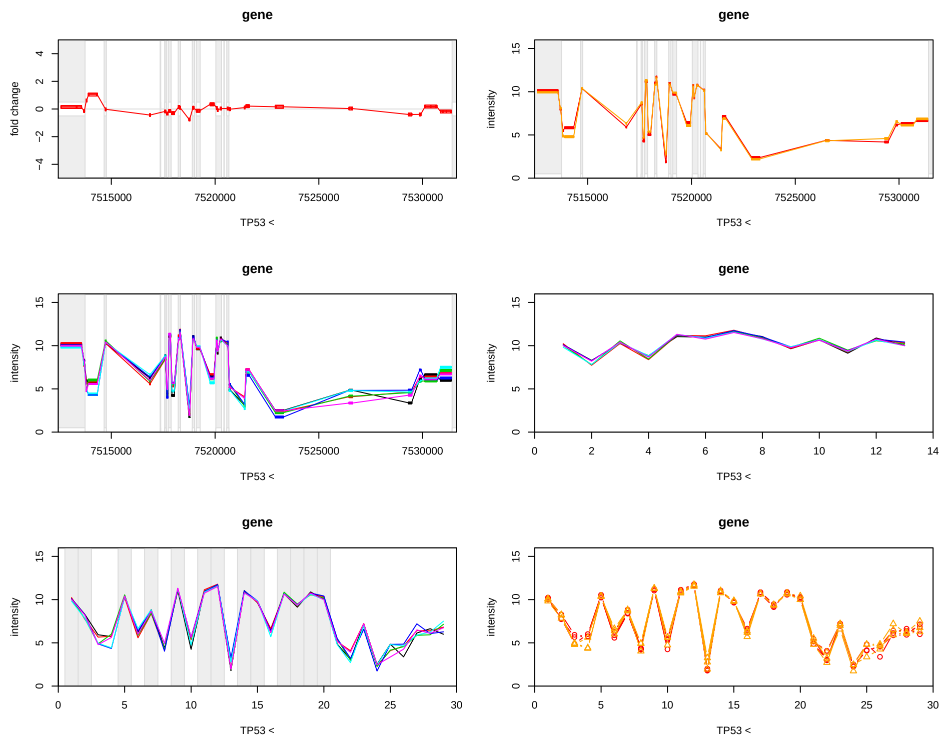
By default, exons with no matching probesets (following filtering for multi-targeting probesets) are coloured white.

```
> par(mfrow = c(3, 1))
> plotGene("ENSG00000141510", x.rma, gps = list(1:3, 4:6), type = "mean-fc",
+     scale.to.gene = TRUE)
> plotGene("ENSG00000141510", x.rma, gps = list(1:3), type = "mean-int",
+     col = heat.colors(16))
> plotGene("ENSG00000141510", x.rma, gps = list(4:6), type = "mean-int",
+     col = heat.colors(16))
```



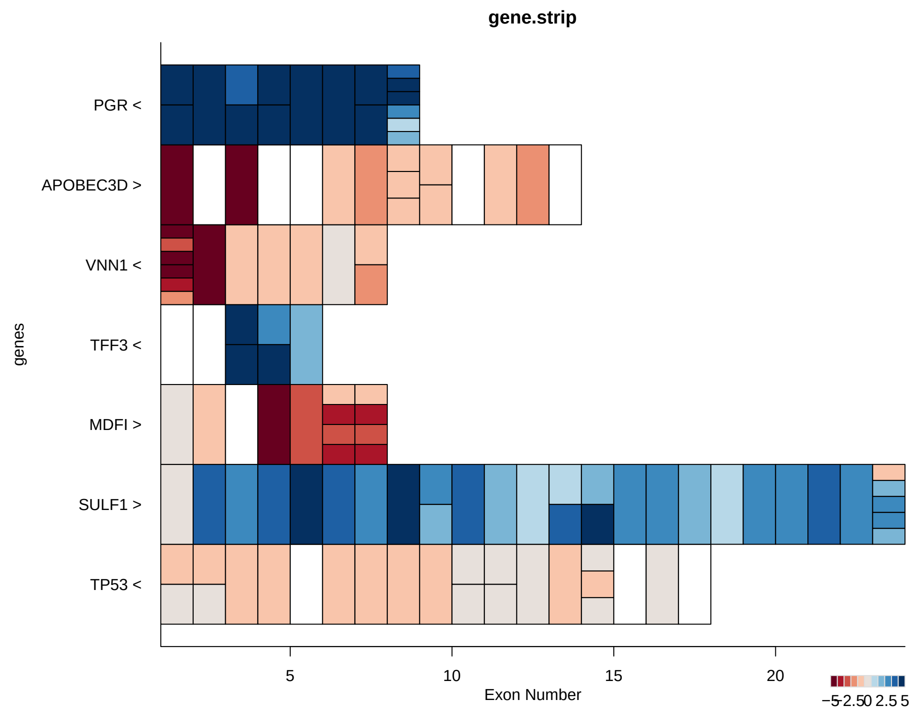
Another utility to visualize the expression of a gene is `?plot.gene.graph`. It creates a line-plot for a specified gene, including intronic probesets. For example:

```
> par(mfrow = c(3, 2))
> gene.graph("ENSG00000141510", x.rma, gps = list(1:3, 4:6), type = "mean-fc",
+     gp.col = "red")
> gene.graph("ENSG00000141510", x.rma, gps = list(1:3, 4:6), type = "mean-int",
+     gp.col = c("red", "orange"))
> gene.graph("ENSG00000141510", x.rma, gps = list(1, 2, 3, 4, 5,
+       6), type = "mean-int", gp.col = 1:6)
> gene.graph("ENSG00000141510", x.rma, gps = list(1, 2, 3, 4, 5,
+       6), type = "mean-int", gp.col = 1:6, by.order = TRUE)
> gene.graph("ENSG00000141510", 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("ENSG00000141510", 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), by.order = TRUE, show.introns = TRUE,
+       exon.bg.col = NA)
```



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

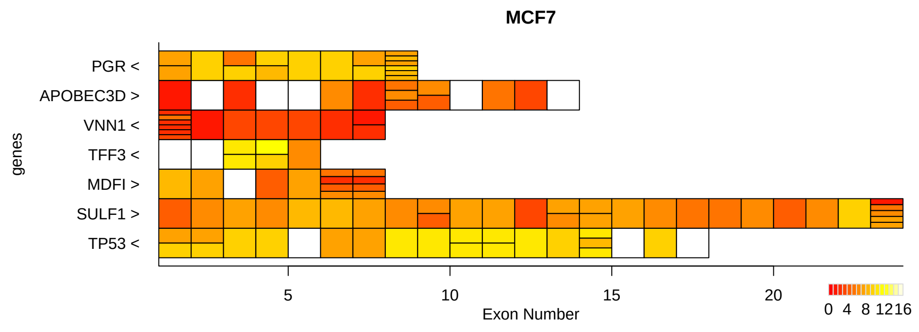
```
> all.genes <- probeset.to.gene(featureNames(x.rma))
> gene.strip(all.genes, x.rma, list(1:3, 4:6), type = "mean-fc")
```

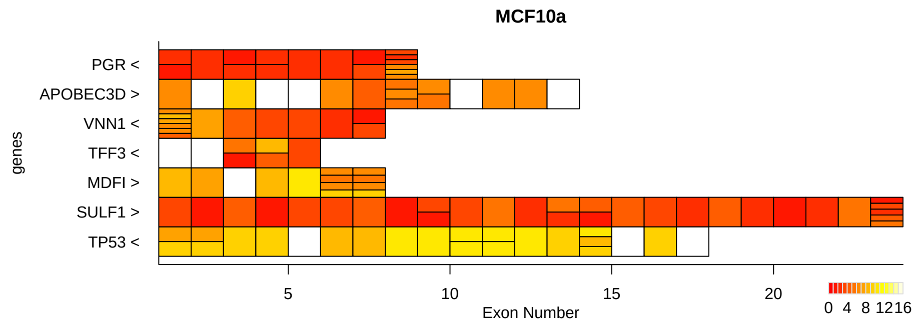


Here, each row corresponds to a gene, and each exon is plotted in exon-order along the X axis. The plot is coloured as before; exons for which a uniquely matching probeset cannot be found are, by default, coloured white. When multiple probesets hit the same exon, these are stacked vertically within that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

Alternatively, plots could be coloured by intensity:

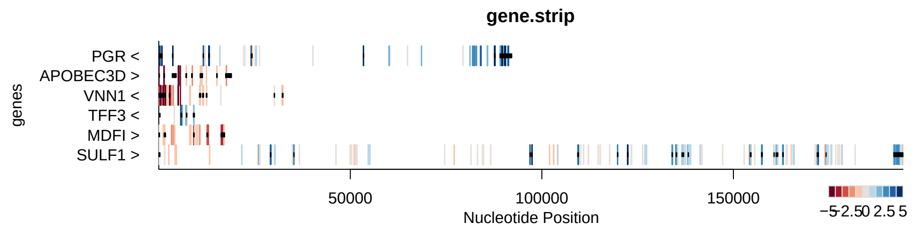
```
> par(mfcol = c(2, 1))
> gene.strip(all.genes, x.rma, list(1:3), type = "median-int",
+     col = heat.colors(16), main = "MCF7")
> gene.strip(all.genes, x.rma, list(4:6), type = "median-int",
+     col = heat.colors(16), main = "MCF10a")
```





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)
```



9 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 calculateed 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.

The two sets of arrays can be specified in two ways: First, by using numeric indices defining the appropriate columns in the expression data. This is doine by supplying these as a list to *members* (e.g. *members=list(1:3,4:6)* will calculate the splicing index between arrays 1,2,3 and 4,5,6. Alternatively, the annotation in the *pData* object from *x* can be used (e.g. *group="treatment",members=c("a","b")*, will compare between the arrays labelled "a", and "b" in the "treatment" column of *pData(x)*).

The implementation also calculates a *p.value* and *t.statistic* for each probeset; these are returned alongside the splicing index.

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",
+     c("a", "b"))
```

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