Analysis and annotation of Affymetrix exon arrays using Bioconductor and exonmap

Bioc 2008

Crispin J Miller
Cancer Research UK
Applied Computational Biology Group, Paterson Institute for Cancer Research, Manchester, UK

bioinformatics.picr.man.ac.uk

Contents

Preamble

In this tutorial, we will consider how to use the Bioconductor package, *exonmap*, to help interpret Affymetrix Exon array data.

Exonmap provides detailed probe-level annotation of Affymetrix Exon arrays against the human genome, using data stored in the annotation database and genome browser, X:Map (http://xmap.picr.man.ac.uk). X:Map contains mappings between individual probes (and probesets) and the Ensembl build of the genome. Ensembl is then used to provide mappings between these and the exons, transcripts and genes they target.

Citing exonmap

If you find exonmap useful in your work, please cite:

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.

and also Bioconductor:

Gentleman, R. C., V. J. Carey, et al. (2004). "Bioconductor: open software development for computational biology and bioinformatics." Genome Biol 5(10): R80.

The following describes X:Map in more detail:

Yates T, Okoniewski MJ, Miller CJ. X:Map: annotation and visualization of genome structure for Affymetrix exon array analysis. Nucleic Acids Res. 2008 Jan;36 (Database issue):D780-6.

Getting more information

Every function in exonmap has a manual page that can be viewed by typing either *help(<function-name>)*, or *?<function-name>*.

In addition, the package has a vignette: to get at this, type *openVignette()*.

For more detailed installation instructions, see INSTALL.txt in the subdirectory *inst/doc/* in package download.

Agenda

INTRODUCTION	
	What are exon arrays?
	What are the challenges?
	How does exonmap approach them?
FIRST STEPS	
	Package installation and configuration
	Normalizing and pre-processing
FILTERING PROBESETS – 1	
	Identifying significant genes
EXPLORING KNOWN GENES OF INTEREST	
	Plotting and visualizing data

BREAK

FILTERING PROBESETS – 2	
	Cross hybridizing probesets
	Exonic, intronic, intergenic targeting probesets
MAPPING DATA	
	X.to.Y functions
	ESTs, Genescan predictions
SPLICING INDEX	
PUTTING IT ALL TOGETHER	
	A worked example
PLOTS AND VISUALIZATIONS	
	X:MapBridge

Introduction

Exon arrays differ from most other microarrays because they feature multiple probesets per transcript: the aim is to target every known and predicted exon in the entire genome with its own individual probeset. This presents a number of challenges, due to the complexity and coverage of the arrays (discussed in e.g. [1-4]).

For a start, since each array contains about 6 million probes, initial normalization and preprocessing can take time and/or significant memory. A number of R and Bioconductor packages exist to make this job easier for large arrays, including *XPS*, *aroma.affymetrix* and *oligo*. An alternative is to use the Affymetrix Power Tools (APT) software from by Affymetrix to generate a tab delimited output file, and then to load this into Bioconductor for further analysis. This is also useful if you are going to be using detection calls as part of your analysis.

For this tutorial, as you will see below, we will start with an already preprocessed *ExpressionSet*. This is a comparison between two cell lines, MCF7 (breast tumour derived) and MCF10A (normal epithelial). It is described in more detail in [2].

Another challenge arises from the increasing amount of detailed annotation required to interpret the arrays. Since each probeset reports on a fragment of a transcript (or multiple transcripts, if they share overlapping exons) it is no longer sufficient simply to provide a simple mapping between transcript or gene identifier; more fine grained mappings are required to place each probeset against their target sequence and to locate them relative to their target exons. In addition, some probesets contain probes capable of targeting multiple sites in the transcribed genome. It is useful to be aware of these, since their signal is potentially influenced by a number of alternate transcripts, and generally to filter them out of the data prior to analysis.

The strategy taken with X:Map is to first map each probe to the entire genome sequence (we do this because there is substantial transcription outside known genes), and to record these hits in a relational database. These database tables are associated with a local copy of Ensembl [5] and stored procedures are used to provide a clean API against which to program (and also for speed). The *exonmap* package then makes these data available in Bioconductor [6].

Installation

Configuration directory

Exonmap needs access to an installation of the X:Map database. This can be installed locally on the same machine running R, or on a separate database server and connected to remotely. We will not go into details about the X:Map installation here – this is described within the database installation instructions which can be found *within* the database tarballs at http://xmap.picr.man.ac.uk/download, but we will instead focus on the exonmap configuration and setup.

Exonmap makes use of a local config directory in which it stores a number of relatively small cache files, and also looks in to find out how to connect to X:Map.

The location of the config directory should be specified by the environment variable *R_XMAP_CONF_DIR*.

Although it is possible to do this during your R session:

```
> Sys.setenv(R_XMAP_CONF_DIR="~/.exonmap")
> library(exonmap)
```

this is not ideal, and you will almost certainly want to set this up permanently. How this is done is platform specific, and may well also depend on how your local systems' administrator prefers to configure your machine. For this reason, details will vary and you should contact your local expert if in doubt.

If *exonmap* can't find the *R_XMAP_CONF_DIR* environment variable, it will complain with a warning and will try to find a directory called *.exonmap* in your home directory.

databases.txt

Exonmap uses a file, *databases.txt*, in the config directory to specify what instances of the X:Map database are available. This is described in detail in the package installation instructions, and a sample file is provided as part of the tutorial. X:Map can support multiple versions of the same database, useful when working on a number of projects at once - it is possible to have an older version of the database to support work on an ongoing paper will also offering the most recent version to support a newer project.

It is also possible to offer access to different servers all providing the same database version. This is useful for example if you want to have a local installation on a laptop, but also to be able to access a more powerful database server when connected to a network.

Note that this differs from previous versions of the package.

db.local

If the directory *db.local* exists within the config directory, exonmap will try to build a small local data file that supports its filtering functions. There will be one file per database, and it is placed within the *db.local* directory. Use of this data makes large filtering queries run many times faster. It is almost certainly a good idea to create *db.local*. A file for each database version will be created, and these files will persist between sessions.

To check everything is OK:

```
library(exonmap)
xmapConnect()
symbol.to.gene("TP53")
```

Initial preprocessing

We will not go into much detail about preprocessing here, since *exonmap* focuses mainly on aspects further down the analysis pipeline. Instead we will make use of some already RMA[7] processed expression data in the file *"mcf7.mcf10a.R"* provided with this tutorial.
(We will avoid processing the data here).

This was produced using the following:

```
> library(exonmap)
> raw.data <- read.exon()
> raw.data@cdfName <- "exon.pmcdf"
> x <- rma(raw.data)
```

read.exon loads exon data from the current local directory, and makes use of a whitespace delimited file (by default called *'covdesc'*) to define which arrays to read, and also to provide additional annotation data. This is described in more detail in the package vignette, but is, essentially, a column of cel file names, followed by one or more columns corresponding to experimental factors (each with a heading). These end up in the resultant *eSet*'s annotation slot:
(Before running this make sure you are in the tutorial directory by using *setwd("<dir>").*)

```
library(exonmap)
load("mcf7.mcf10a.R")
pData(x)
```

Note also that we are explicitly setting the *cdfName* of the *ExpressionSet*. This is to make use of a custom CDF environment that contains only the pm probes on the array – it can be downloaded from http://xmap.picr.man.ac.uk.

Filtering probesets based on expression

The approach we take with exonmap is to try to use as many of the existing strategies and algorithms for dealing with expression arrays as possible. Thus we normalize, preprocess and filter the probesets on the array in order to generate a list of 'interesting' probesets.

Typically these will be both statistically significant, and have a reasonable level of change between sample groups, correlations, etc. The fundamental point is that at this stage, we are not treating the arrays as any different from any other type of Affymetrix array.

Exonmap provides some simple utility functions (written mainly to make it easier to provide documentation and examples) that provide pairwise comparisons between two sets of arrays. We will use them here to generate a list of probesets to explore during this tutorial:

```
r <- pc(x,"group",c("a","b"))
plot(fc(r),log10(tt(r)),pch=20)
keep <- (abs(fc(r)) > 4) & (tt(r) < 0.001)
ps   <- names(fc(r))[keep]
length(ps)
ps[1:100]
```

You will almost certainly want to substitute a more sophisticated analysis here – for example, you might want to use *limma* to generate a list using FDR moderated p-scores:

```
> library(limma)

pairwise <- function(x,group,gp1,gp2,fc=log2(1),p.value=0.001) {
  pd <- pData(x)
  grp <- pd[, colnames(pd) == group]
  c1 <- grp == gp1
  c2 <- grp == gp2
  design      <- as.matrix(cbind(as.numeric(c1),as.numeric(c2)))
  colnames(design) <- c(gp1,gp2)
  fit         <- lmFit(x,design)
  ct <- paste(gp1,"-",gp2,sep="")
  cont.matrix <- makeContrasts(contrasts=ct,levels=design)
  fit2        <- contrasts.fit(fit,cont.matrix)
  fit2        <- eBayes(fit2)
  result      <- decideTests(fit2,lfc=fc,p.value=p.value)
  keep        <- result@".Data" != 0
  r <- cbind(p.adjust(fit2$"p.value"[keep]),fit2$"coeff"[keep])
  rownames(r) <- names(result@".Data"[keep,])
  colnames(r) <- c("adj.p.value","fold.change")
  r
}

topN <- function(x,N=20,type=c("both","up","down")) {
  type <- match.arg(type)
  i    <- 1:dim(x)[1]
  o    <- order(x[,2],decreasing=T)
  i    <- i[o]

  if(type == "both") N <- floor(N/2)
  u    <- i[1:N]
  d    <- i[length(i)- N + 1:N]
  keep <- switch(type,
       both=c(u,d),
       up=u,
       down=d)
  x[keep,,drop=FALSE]
}
```

```
r  <- pairwise(x,"group","a","b",fc=4)
ps.2 <- rownames(r)
```

(Not surprisingly, at this fold change threshold – 16-fold, this gives very similar results to the unadjusted comparison above).

A note on FDR and multiple testing

One of the issues with exon arrays is that the large amount of probesets means that many apparently significant probesets can be generated simply by chance. The alternative, of making the significance threshold more significant (either explicitly, or via an FDR or multiple testing correction approach) has the disadvantage of losing additional True Positives.

The strategy we use here, of first finding significant probesets and then mapping to genes, makes this less of an issue than it might appear at first instance (see below), however, it is also sometimes possible to restrict the analysis to a smaller set of more relevant probesets in the first place.

Approximately 50% of the probesets on an exon array target outside known protein coding exons, and if you know that downstream analysis is likely to focus only on the well-characterized genes in the genome, it might be appropriate to ignore all these additional probesets from the outset. The filtering functions described in the section 'Filtering based on annotation' can help with this.

Exploring known genes

At this point, we have a list of significant probesets, identified by a simple pairwise comparison between replicate groups. In the next section we will begin to explore these in more detail, but generally, we already have a number of genes likely to be of interest in a given experiment. Exonmap can be used to generate plots of these genes directly:

```
library(exonmap)
xmapConnect()
g <- symbol.to.gene("TP53")
gene.graph(g,x,gps=list(1,2,3,4,5,6),gp.col=c("red","red","red","orange","orang
e","orange"))
```

When you run this, *xmapConnect()* will prompt you to choose a database to connect to – if you already know which one, you can specify it directly (e.g. *xmapConnect("human")*), and you will not be presented with a menu.

The function *symbol.to.gene()* then connects to the database and retrieves the gene identifier(s) that map to the specified symbol(s). This is our first use of *exonmap* and a fair amount is going on in the background; the aim is to hide all the database connectivity and so on from the user and present as simple an interface as possible.

Finally, *gene.graph()* again uses exonmap to find the intron/exon structure of the specified gene and the probesets that target it, and then uses this information to plot the expression data for that gene.

A selection of different plot functions exist; we will return to plotting later on.

Filtering based on annotation

Affymetrix have placed probes on the array with a variety of levels of confidence based on the amount of supporting experimental evidence (Affymetrix refer to these as groupings as 'core', 'extended' and 'full'). In many cases we are most interested in changes to the most well-characterized exons within the dataset – changes to intergenic or intronic probesets while potentially interesting are probably harder to deal with and we may wish to leave these to later.

Exonmap provides a set of filtering functions to partition a probeset list based on the region of the genome they target:

```
e <- exonic(ps)
```

will return a list containing only exon targeting probesets. Similarly,

```
i  <- intronic(ps)
ig <- intergenic(ps)
```

will filter for intronic and integenic probesets, respectively.

Note that by default these functions will also remove multi-targeted probesets (see below).

Multi-target (cross-hybridizing) probesets

An important aspect of high-density arrays is that occasionally some probes can hybridize to multiple places in the transcribed genome. This issue is becoming even more prominent as increasing amounts of transcription are being found outside known protein coding regions. For this reason, X:Map was built by searching against the entire genome (rather than just known transcripts) and recording the locations where each probe matches. The function *multitarget()* makes it possible to identify any probesets that are found to hit at more than one location:

```
multitarget(ps)
```

All these functions can take the parameter *exclude=FALSE*, in which case they return anything that **doesn't** match the criterion, so for example,

```
multitarget(ps,exclude=TRUE)
```

will return any probesets that aren't multi-targeted.

We can also test probesets based on annotation – i.e. the following functions exist, and should be self-explanatory:

```
is.exonic(ps)
is.intronic(ps)
is.intergenic(ps)
is.multitarget(ps)
```

The functions:

```
select.probewise(ps,"exonic")  # or intronic, intergenic, multitarget
exclude.probewise(ps,"exonic") # or intronic, intergenic, multitarget
```

also exist, and provide an alternative way of getting at the same functionality.

Behind the scenes

The function *probeset.stats()* reveals a little bit more about how all of this is implemented:

```
probeset.stats(ps[40:50])
```

```
    probeset_id probeset hitScore exonScore geneScore
40        17795  2336556        1         1         1
41        17797  2336558        1         1         1
42        17799  2336560        1         1         1
43        17805  2336566        1         4         1
44        18471  2337413      150         0         0
45        19635  2338842        1         0         1
46        20443  2339802        1         1         1
47        20452  2339811        1         1         1
48        20455  2339814        1         1         1
49        20469  2339828        1         4         1
50        20554  2339932        1         1         1
>
```

X:Map stores a hit table for each probeset, that scores each probeset according to the number of genome, exon and gene hits it has (hitScore, exonScore and geneScore, respectively). These are calculated by multiplying the scores for each individual probe within a probeset together. This means that:

- If a probeset hits the genome once, and only once, with each of its probes, the hitScore will be 1 * 1 * 1 *1 = 1.
- If one or more of its probes doesn't hit the genome, the hitScore will be 0.
- If all probes hit at least once, but one or more hit more than once, the hitScore will be > 1.

thus, probeset #44 is multitarget (and intronic), #40 is exonic, and #45 is intronic. Exon and gene scores are calculated similarly. The scores are computed this way because they can be done quickly on the database server in SQL – and they provide a quick way of filtering the data:

- Multi-target: hitScore > 0
- Exonic: exonScore > 0
- Intronic: exonScore ==0 & geneScore > 0

Happily, this is hidden from the user; in most cases it is not necessary to think about…

Mapping functions

As well as filtering to include or exclude probesets from a list based on where they hit the genome, it is also possible to map between probes, probesets and their target genes, exons and transcripts. For example,

```
ex <- probeset.to.exon(e[1:100])
ex
```

will take the first 100 probesets in our probeset list and return the exons they target (remember, *e* is the vector of exonic targeting probesets we created in the previous section). Similar functions exist for transcripts and genes:

```
tr <- probeset.to.transcript(e[1:100])
g  <- probeset.to.gene(e[1:100])
```

…and also for mapping between levels:

```
exon.to.transcript(ex)
transcript.to.gene(tr)
```

We can also map back in the opposite direction – in fact the functions are all of the form x.to.y. Thus we have:

```
probe.to.probeset,          probeset.to.probe
probeset.to.exon,           probeset.to.transcript,    probeset.to.gene
exon.to.probeset,           exon.to.transcript,        exon.to.gene
transcript.to.probeset,     transcript.to.exon,        transcript.to.gene
gene.to.probeset,           gene.to.exon,              gene.to.transcript
```

All these functions take the parameter *as.vector*, which is *TRUE* by default. If *FALSE*, then a *data.frame* is returned with some additional annotation data besides the database identifier. Each of these functions also has a parameter, *unique*. When *TRUE* (as it is by default) duplicate entries will be removed from the results.

Finally, the function *symbol.to.gene* (seen above in the first graph example) provides a translation between gene symbols and Ensembl IDs.

Note that *probeset.to.probe*, by default, strips our multitarget probesets. This is because a small number of probesets contain probes that hit the genome at a large number of sites. This can result in extremely large tables coming back from the database, and can be slow. Since in most cases these probesets aren't interesting or useful, we filter them out by default.

Getting gene, exon and probeset data together

Often, we want to compute some overall statistics for a gene (such as the splicing index, see below). To do this we need to know details about both its constituent exons and its target probesets and we want to put these together alongside our expression data. The

functions *gene.to.exon.probeset* and *gene.to.exon.probeset.expr* do this in one go. They do most of their work on the database server, making them relatively fast.

```
gene.to.exon.probeset(g[1:2])
gene.to.exon.probeset.expr(x,g[1:2])
```

Getting more information

The functions *gene.details, transcript.details* and *exon.details* each give additional annotation data.

```
gene.details(g)
```

Note that *probeset.details* does not exist, since a probeset is little more than a name with which to group a set of probes together. The function *probeset.to.probe* with *as.vector=FALSE* will produce a table of individual probe hit-locations for the specified probeset(s).

Range queries

X:Map also supports queries to find out which genes/transcripts/exons are located in a particular region of the genome:

```
genes.in.range(7400000,7600000,1,"17")
transcripts.in.range(7400000,7600000,1,"17")
probesets.in.range(7400000,7600000,1,"17")
```

A simple example

We are now in a position to generate a gene list from our example cell line data. Each entry will correspond to a gene where at least one, non-multitarget, exon targeting probeset is found to be statistically significant and differentially expressed.

We will use the simple pairwise comparison for this, but we could (and should) of course use a more statistically robust approach.

```
library(exonmap)
xmapConnect()
# load("mcf7.mcf10a.R")
r <- pc(x,"group",c("a","b"))
keep <- (abs(fc(r)) > 4) & (tt(r) < 0.001)
ps   <- names(fc(r))[keep]

ps  <- exonic(ps) # also gets rid of multi-targeted probesets
g   <- probeset.to.gene(ps)
gds <- gene.details(g)
```

Strategies for partitioning the data

This produces a list with about ~880 genes in it. While useful, clearly we will want to do some more analysis to partition this list further. One simple approach is simply to divide

the data according to the variance across each gene. We already know that at least one exon in each gene in the list has at least a 2^4-fold difference in expression, so genes from this list with low variance in their fold-changes have large and consistent changes across their exons.

```
gepe <- gene.to.exon.probeset.expr(x,g[1:100]) #just do 100 genes
fc.var <- function(g) {
       fcs <- apply(g,1,function(a) {
             mean(as.numeric(a[1:3])) - mean(as.numeric(a[4:6]))
       })
       var(fcs)
}

l  <- split(gepe,gepe$"gene")
vs <- sapply(l,fc.var)
o  <- order(vs,decreasing=FALSE)
g.ordered <- names(l)[o]
```

We can generate a summary plot for a set of genes using the *gene.strip* function. This is similar to a heatmap, with each row corresponding to a gene, and genes are plotted in exon order and coloured by, for example, fold change. Here we look at the 40 least varying genes:

```
gene.strip(g.ordered[1:40],x,list(1:3,4:6),type="mean-fc")
```

… and the 40 most varying:

```
gene.strip(rev(g.ordered)[1:40],x,list(1:3,4:6),type="mean-fc")
```

An alternate way of partitioning the data is to use the splicing index. This is described in the Affymetrix whitepaper 'Alternative transcript analysis methods for exon arrays' [8].

```
s <- si(x,g,list(1:3,4:6))
```

si returns a list, one element for each gene. Each entry in the list is a *data.frame* with the splicing index for each (exon targeting) probeset, its accompanying p-value and the gene level average. We can partition the data into two groups – genes that are, on average, up-regulated, and those that are down-, and then sort the genes within these two groups according to the maximum splicing index. Finally, we plot the results.

```
keep <- intersect(names(s),g)
s <- s[keep]
max.si <- sapply(s, function(a) {
 max(abs(a$si))
})

gene.av <- sapply(s, function(a) {
 max(a$gene.av)
})


up<-gene.av>0

o<-order(max.si[up],decreasing=TRUE)
gene.strip(g[up][o[1:40]],x,list(1:3,4:6),type="mean-fc",
main="up",probes.min=0)

o<-order(max.si[!up],decreasing=TRUE)
gene.strip(g[!up][o[1:40]],x,list(1:3,4:6),type="mean-fc",
main="down",probes.min=0)
```

More plots

We can also plot genes to show their gene structure:

```
plotGene("ENSG00000151914",x,list(1:3,4:6),
         type="mean-fc", main="up",probes.min=0)
```

Note that both *plotGene* and *plot.graph* allow their *xlim* to be specified, allowing you to zoom in to a particular region of a gene. This allows you to see the location of each individual probe within a probeset.

A mechanism has also been developed for exporting data from X:Map and displaying it directly in the X:Map browser. This is still under development but will be released shortly.
(The first two parameters are the start and end points of the region, then the strand (forward: 1 ; reverse: -1).

References

1. Abdueva D, Wing MR, Schaub B, Triche TJ: **Experimental comparison and evaluation of the Affymetrix exon and U133Plus2 GeneChip arrays**. *PLoS ONE* 2007, **2**(9):e913.
2. Okoniewski MJ, Hey Y, Pepper SD, Miller CJ: **High correspondence between Affymetrix exon and standard expression arrays**. *Biotechniques* 2007, **42**(2):181-185.
3. 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.
4. Yates T, Okoniewski MJ, Miller CJ: **X:Map: annotation and visualization of genome structure for Affymetrix exon array analysis**. *Nucleic Acids Res* 2008, **36**(Database issue):D780-786.
5. Flicek P, Aken BL, Beal K, Ballester B, Caccamo M, Chen Y, Clarke L, Coates G, Cunningham F, Cutts T *et al*: **Ensembl 2008**. *Nucleic Acids Res* 2008, **36**(Database issue):D707-714.
6. Gentleman RC, Carey VJ, Bates DM, Bolstad B, Dettling M, Dudoit S, Ellis B, Gautier L, Ge Y, Gentry J *et al*: **Bioconductor: open software development for computational biology and bioinformatics**. *Genome Biol* 2004, **5**(10):R80.
7. Irizarry RA, Hobbs B, Collin F, Beazer-Barclay YD, Antonellis KJ, Scherf U, Speed TP: **Exploration, normalization, and summaries of high density oligonucleotide array probe level data**. *Biostatistics* 2003, **4**(2):249-264.
8. Affymetrix: **Alternative transcript analysis methods for exon array analysis**. *Affymetrix whitepaper* 2007.