oligo - Primer

Benilton Carvalho

Contents

1 Data Import **3**

2 Preprocessing Expression Arrays **4**

2.1 Affymetrix Expression 4

2.2 Nimblegen Expression 10

2.2.1 Initialization of the environment 11

2.2.2 Exploring the feature-level data 12

2.2.3 RMA algorithm 14

2.2.4 Assessing differential expression 15

3 Obtaining Genotype Calls from SNP Arrays **18**

4 Preprocessing Exon Arrays **22**

5 Interfacing with ACME to Find Enriched Regions Using Tiling Arrays **28**

6 High Performance Computing Features **31**

6.1 Support to large datasets 31

6.2 Parallel computing 32

6.3 Parallel Computing on Multicore Machines 32

7 Session Info **33**

1 Data Import

To import data using the oligo package, the user must have data at the probe-level. This means that if Affymetrix data are to be imported, the user is expected to have CEL files; if Nimblegen data are used instead, then XYS files are to be available.

Once sets of such files are available, the user can two tools, depending on the array manufacturer, to import the data: `read.celfiles` - for CEL files; and `read.xysfiles` - for XYS files. To assist the user on obtaining the names of the CEL or XYS files, the package provides two functions, `list.celfiles` and `list.xysfiles`, which accept the same arguments as the `list.files` function defined in the R base package. The basic usage of the package tools to import CEL or XYS files present in the current directory consists in combining the `read.files` functions with their `list.files` counterparts, as shown below, in a hypothetical example:

```
R> library(oligo)
R> celFiles <- list.celfiles()
R> affyRaw <- read.celfiles(celFiles)
R> xysFiles <- list.xysfiles()
R> nimbleRaw <- read.xysfiles(xysFiles)
```

The oligo package will attempt to identify the annotation package required to read the data in. If this annotation package is not installed, oligo will try to download it from BioConductor. If the annotation is not available on BioConductor, the user should use the pdInfoBuilder package to create an appropriate annotation. In case oligo fails to identify the annotation package's name correctly, the user can use the *pkgname* argument available for both `read.celfiles` and `read.xysfiles`.

From this point on, this document provides examples on the usage of the oligo package using datasets available in the oligoData package.

Object Name	Description
affyExpressionFS	Latin Square - Affymetrix U95A
nimbleExpressionFS	Sample Expression Dataset Nimbeglen
affyExonFS	Exon Sample Dataset - Human
affySnpFS	HapMap samples on XBA Array
nimbleTilingFS	Sample ChIP-chip dataset

Table 1: Datasets used in this document.

2 Preprocessing Expression Arrays

2.1 Affymetrix Expression

The dataset used in this example corresponds to the Latin Square Data for Expression Algorithm Assessment on the Human Genome U95 platform, made available by Affymetrix on their website[1]. To be used with oligo, requires the availability of the pd.hg.u95a annotation package, built with the pdInfoBuilder package.

After the annotation package is installed, the next step is to load oligo and identify the files to be used in the analysis. The `list.celfiles` function can be used to appropriately list Affymetrix CEL files. If CEL files were available in a directory called `expressionData`, then, in the snippet below, the `celFiles` would contain the CEL filenames, including the full path.

```
R> library(oligo)
R> celFiles <- list.celfiles("expressionData", full.names=TRUE)
```

Importing the CEL files is achieved with the `read.celfiles` function. The function will, in general, correctly identify the annotation package to be used with the experimental data being imported, but the user can specify the *pkgname* argument to force the use of a particular one, if for some reason this is required. Note that the snippet below corresponds to a hypothetical example, in which we would read CEL files saved in a directory called `expressionData`.

```
R> affyExpressionFS <- read.celfiles(celFiles, pkgname="pd.hg.u95a")
```

In reality, the `affyExpressionFS` object is already available in the oligoData package.

[1] http://www.affymetrix.com/support/technical/sample_data/datasets.affx

The example below demonstrates how it could be loaded.

```
R> library(oligoData)
R> data(affyExpressionFS)
```

The object `affyExpressionFS` belongs to the *ExpressionFeatureSet* class, as it corresponds to expression data. The object, like all *FeatureSet*-like objects, represents features in the rows and samples in the columns and can be easily subsetted, using the standard [operator. All the manipulation structure is inherited through te tight integration between oligo and Biobase, whose documentation we recommend to the interested reader.

```
R>   class(affyExpressionFS)
[1] "ExpressionFeatureSet"
attr(,"package")
[1] "oligoClasses"
```

Figure 1 demonstrates how the *image* method can be used to generate pseudo-images of the samples. In this particular plot, we use the first sample as an example and a grayscale palette for plotting.

```
R> image(affyExpressionFS, 1, col=gray((64:0)/64))
```

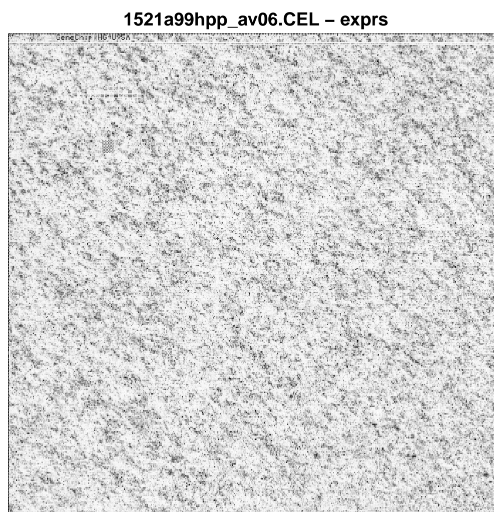


Figure 1: Pseudo-image, used for visual assessment of the array, for sample `1521a99hpp_av06.CEL`.

The user can evaluate the distribution of the observed data by using the *hist* method, which will produce smoothed histograms for each sample available in the dataset. Before

plotting, the method transforms the data using the function passed to the *transfo* argument, whose default is `log2`, explaining why the plot is shown on the $\log_2$ scale.

```
R> hist(affyExpressionFS)
```

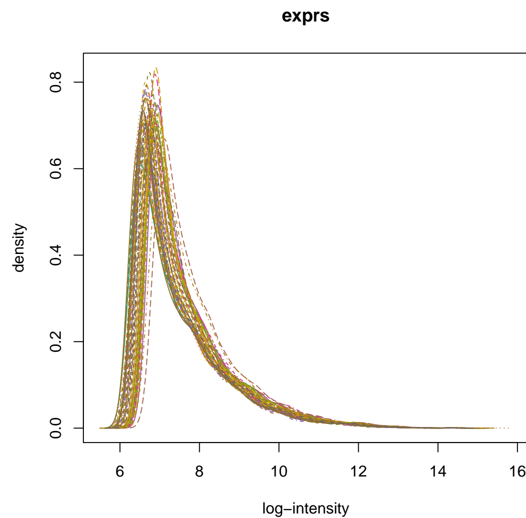


Figure 2: Smoothed histograms for samples in the dataset.

Another approach to assess the data distribution is to use the *boxplot* method. On *FeatureSet* objects, the method will automatically transform the data to the $\log_2$ scale, but this is easily modified through the *transfo* argument, which takes a function as a valid value.

```
R> boxplot(affyExpressionFS)
```

Plotting log-ratio *versus* average intensity can often reveal intensity effects on log-ratios, as shown by the MA plot on Figure 4. The argument *arrays* can be specified to determine which samples will be plotted and the *lowessPlot* is a logical flag to indicate that the user wants a lowess curve to be overlapped to the data points.

```
R> MAplot(affyExpressionFS, which=1, ylim=c(-1, 1))
```

The annotation packages used by oligo store feature sequences. This is done through instances of *DNAStringSet* objects implemented in the Biostrings package. The sequences for PM probes can be easily accessed via the `pmSequence` function, as shown below.

```
R> pmSeq <- pmSequence(affyExpressionFS)
R> pmSeq[1:5]
  A DNAStringSet instance of length 5
    width seq
```

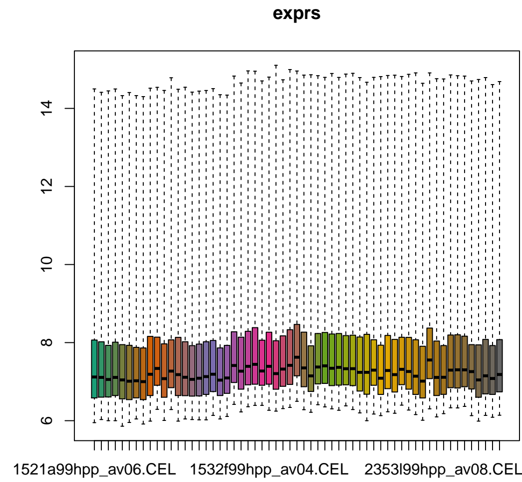


Figure 3: Boxplot showing the distribution of the observed $\log_2$-intensities on the sample dataset. The `boxplot` method implemented in oligo follows the standards of the original method used by R.

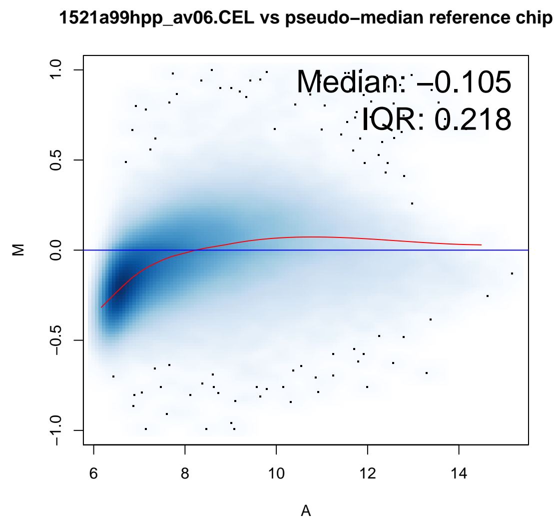


Figure 4: The MA plot can be used to assess the dependence of log-ratios on average log-intensities.

```
[1]     25 GCTGCCCACAGTGACCGACCAGGAG
[2]     25 GCAGCCACCAGTGGACCTAGCCTGG
[3]     25 CAGCCACCAGTGGACCTAGCCTGGA
[4]     25 CGCATCCACGTGAACTTGAGCACTG
[5]     25 GGCTTCACAGTCACTCGGCTCAGTG
```

When importing the data, oligo does not impose any transformation, so one needs to manually apply, for example, the $\log_2$ transform to the intensities of PM probes, which can be accessed with the `pm` function, as needed. Below, we present how to centralize the $\log_2$-PM intensities for each sample in `affyExpressionFS`.

```
R> pmsLog2 <- log2(pm(affyExpressionFS))
```

The dependence of intensity on probe sequence is a well established fact on the microarray literature. The use of the oligo package simplifies significantly the observation of this event, as it provides simple access to both observed intensities and annotation. Below, we estimate the affinity splines coefficients (**?**).

```
R> coefs <- getAffinitySplineCoefficients(pmsLog2, pmSeq)
```

On Figure 5, we show how the results above can be used to estimate the base-position effects on the $\log_2$-intensities observed for the first sample in the dataset. The `getBaseProfile` function provides a simple way of using the affinity coefficients to estimate the effects of interest. It accepts a *plot* argument, which takes logical values, to make the plot and returns, invisibly, the estimated effects. All the arguments that can be passed to the `matplot` function can also be passed to `getBaseProfile`.

```
R> colors <- darkColors(4)
R> xL <- "Base Position"
R> yL <- "Effect"
R> pchs <- c("A", "C", "G", "T")
R> getBaseProfile(coefs[,1], plot=TRUE, pch=pchs, type="b", xlab=xL, ylab=yL,
                 lwd=3, col=colors, ylim=c(-.4, .4))
```

Tools implemented in other packages can be used in conjunction with oligo to investigate different hypothesis. The example below shows how the `alphabetFrequency` function, defined by the Biostrings can be used to determine the GC content of the probe sequences accessed by oligo.

```
R> counts <- Biostrings::alphabetFrequency(pmSeq, baseOnly=TRUE)
R> GCcontent <- ordered(counts[, "G"]+counts[, "C"])
```

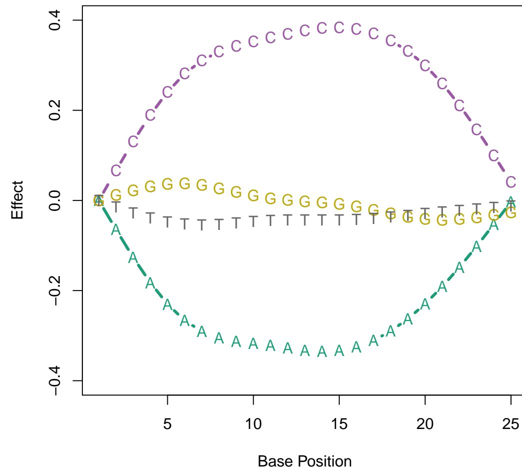


Figure 5: Sequence effect for the first sample in the dataset. These results have been reported in detail elsewhere, but can be easily reproduced with the use of the oligo package.

In addition to Figure 5, we can also plot the $\log_2$-intensities as a function of the GC content computed above. Figure 6 presents the strong dependency of $\log_2$-intensities on GC contents for sample 1, which is also present in all other samples.

```
R> colors <- seqColors(nlevels(GCcontent))
R> xL <- "GC Frequency in 25-mers"
R> yL <- expression(log[2]~intensity)
R> boxplot(pmsLog2[,1]~GCcontent, xlab=xL, ylab=yL, range=0, col=colors)
```

To preprocess expression data, oligo implements the RMA algorithm (**??**). The *rma* method, as shown below, proceeds with background subtraction, normalization and summarization using median-polish.

```
R> ppData <- rma(affyExpressionFS)
```

The results are returned in an *ExpressionSet* instance and used in downstream analyses, as currently done by several strategies for microarray data analysis and described elsewhere.

```
R> class(ppData)
[1] "ExpressionSet"
attr(,"package")
[1] "Biobase"
```

At this point, the user can proceed with, for example, differential expression analyses. The methodologies involved in this step make use of several other packages, like limma

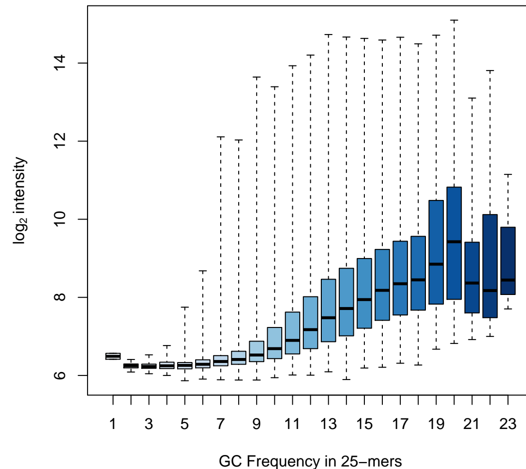


Figure 6: On this boxplot stratified by GC content, we can observe the strong dependency of $\log_2$-intensities on the number of G or C bases observed in the probe sequency.

and genefilter. When preprocessing the data, oligo stores the summaries in a matrix called `exprs`, present in the `assayData` data slot of the *ExpressionSet* object. Therefore, the only restriction the additional strategies used with the preprocessed data have is to be aware that the processed data can be easily accessed with the *exprs* method.

2.2 Nimblegen Expression

This section presents a non-trivial use of the oligo Package for the analysis of NimbleGen Expression data. This vignette follows the structure of the chapter **From CEL files to a list of interesting genes** by R. A. Irizarry *in Bioinformatics and Computational Biology Solutions Using R and Bioconductor*, which shows a case study for Affymetrix Expression arrays.

In order to analyze microarray data using oligo, the user is expected to have installed on the system a package with the annotation for the particular array design on which the experiment was performed. For the example in question here, the design is hg18_60mer_expr and the annotation package associated to it is pd.hg18.60mer.expr, which is built by using

the pdInfoBuilder package.

2.2.1 Initialization of the environment

On this particular example, we will read XYS files instead of loading the *FeatureSet* object already available through the oligoData package (the `maqc` object that we will create below is exactly the `nimbleExpressionFS` data object provided by the oligoData package). We start by loading the packages that are going to be used in this session. The maqcExpression4plex package provides a set of six samples on the MAQC Study; the set is comprised of samples on two groups: universal reference and brain. The remaining packages offer additional functionality, like tools for filtering, plotting and visualization.

```
R>   library(oligo)
R>   library(maqcExpression4plex)
R>   library(genefilter)
R>   library(limma)
```

Once the package is loaded, we can easily get the location of the XYS files that contain the intensities by calling `list.xysfiles`, which takes the same arguments as `list.files`. To minimize the chance of problems, we strongly recommend the use of `full.names=TRUE`.

```
R> extdata <- system.file("extdata",
       package="maqcExpression4plex")
R> xys.files <- list.xysfiles(extdata,
       full.names=TRUE)
R> basename(xys.files)
[1] "9868701_532.xys" "9868901_532.xys" "9869001_532.xys"
[4] "9870301_532.xys" "9870401_532.xys" "9870601_532.xys"
```

To read the XYS files, we provide the `read.xysfiles` function, which also takes `phenoData`, `experimentData` and `featureData` objects and returns an appropriate subclass of *FeatureSet*.

```
R> theData <- data.frame(Key=rep(c("brain", "universal reference"), each=3))
R> rownames(theData) <- basename(xys.files)
R> lvls <- c("channel1", "channel2", "_ALL_")
R> vMtData <- data.frame(channel=factor("_ALL_", levels=lvls),
                         labelDescription="Sample type")
R> pd <- new("AnnotatedDataFrame", data=theData, varMetadata=vMtData)
R> maqc <- read.xysfiles(xys.files, phenoData=pd)
```

```
Checking designs for each XYS file... Done.
Allocating memory... Done.
Reading /Users/carval03/Rlibs/2.15/maqcExpression4plex/extdata/9868701_532.xys.
Reading /Users/carval03/Rlibs/2.15/maqcExpression4plex/extdata/9868901_532.xys.
Reading /Users/carval03/Rlibs/2.15/maqcExpression4plex/extdata/9869001_532.xys.
Reading /Users/carval03/Rlibs/2.15/maqcExpression4plex/extdata/9870301_532.xys.
Reading /Users/carval03/Rlibs/2.15/maqcExpression4plex/extdata/9870401_532.xys.
Reading /Users/carval03/Rlibs/2.15/maqcExpression4plex/extdata/9870601_532.xys.
```

```
R> class(maqc)
```

```
[1] "ExpressionFeatureSet"
attr(,"package")
[1] "oligoClasses"
```

2.2.2 Exploring the feature-level data

The `read.xysfiles` function returns, in this case, an instance of *ExpressionFeatureSet* and the intensities of these files are stored in its `exprs` slot, which can be accessed with a method with the same name.

```
R> exprs(maqc)[10001:10010, 1:2]
```

```
      9868701_532.xys 9868901_532.xys
10001          734.67          742.22
10002         4786.11         4434.67
10003        25600.33        26154.89
10004         1078.56         1092.78
10005         3056.44         3128.33
10006          310.22          385.00
10007              NA              NA
10008              NA              NA
10009          599.44          713.00
10010        28711.67        29794.67
```

The *boxplot* method can be used to produce boxplots for the feature-level data.

```
R> boxplot(maqc, main="MAQC Sample Data")
```

Similarly, a smoothed histogram for the feature-level data can be obtained with the *hist* method.

```
R> hist(maqc, main="MAQC Sample Data")
```

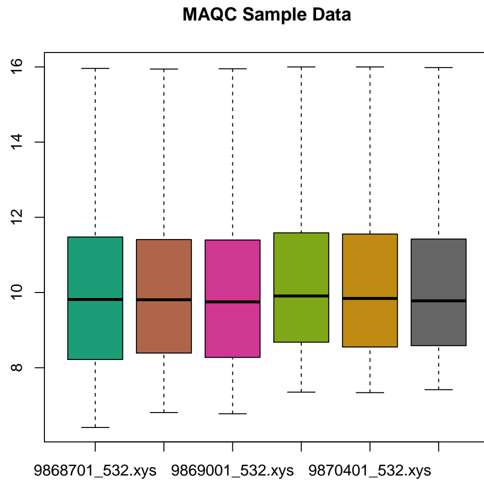


Figure 7: Distribution of $\log_2$-intensities of samples on the MAQC dataset.

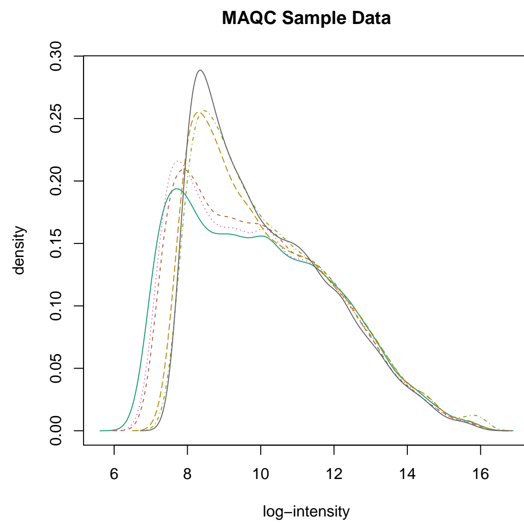


Figure 8: Smoothed histogram of $\log_2$-intensities of samples on the MAQC dataset.

2.2.3 RMA algorithm

The RMA algorithm can be applied to the raw data of expression arrays. It is available via the *rma* method. The algorithm will perform background subtraction, quantile normalization and summarization via median polish. The result of *rma* is an instance of *ExpressionSet* class, which also contains an *exprs* slot and method.

```
R>   eset <- rma(maqc)
Background correcting
Normalizing
Calculating Expression
R>   class(eset)
[1] "ExpressionSet"
attr(,"package")
[1] "Biobase"
R>   show(eset)
ExpressionSet (storageMode: lockedEnvironment)
assayData: 24000 features, 6 samples
  element names: exprs
protocolData
  rowNames: 9868701_532.xys 9868901_532.xys ...
    9870601_532.xys (6 total)
  varLabels: exprs dates
  varMetadata: labelDescription channel
phenoData
  rowNames: 9868701_532.xys 9868901_532.xys ...
    9870601_532.xys (6 total)
  varLabels: Key
  varMetadata: channel labelDescription
featureData: none
experimentData: use 'experimentData(object)'
Annotation: pd.hg18.60mer.expr
R>   exprs(eset)[1:10, 1:2]
          9868701_532.xys 9868901_532.xys
NM_000014       12.286393       12.272719
NM_000015        4.455020        4.625539
NM_000016       12.386405       12.203391
NM_000017        8.516991        8.541788
NM_000018       12.578168       12.414070
NM_000019       11.698035       11.636985
NM_000020        8.910401        9.209599
```

```
NM_000021        11.763186        11.810772
NM_000022         8.918243         8.445262
NM_000023         8.937284         9.075812
```

The *boxplot* and *hist* methods are also implemented for *ExpressionSet* objects. Note that *rma*'s output is in the $\log_2$ scale, so we call such methods using the argument `transfo=identity`, so the data are not transformed in any way.

```
R>   boxplot(eset, transfo=identity, main="After RMA")
R>   hist(eset, transfo=identity, main="After RMA")
```

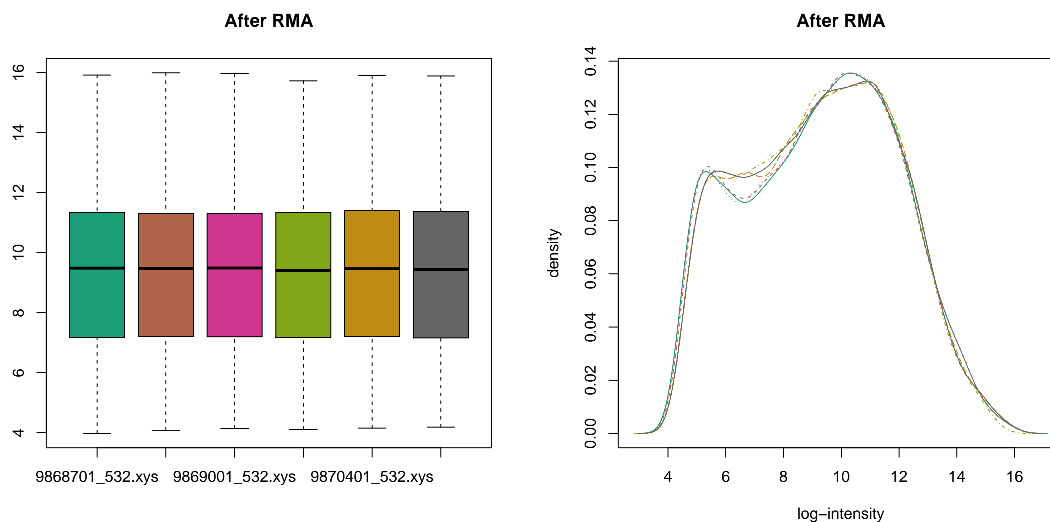


Figure 9: Boxplot and smoothed histogram for MAQC data after preprocessing.

2.2.4 Assessing differential expression

One simple approach to assess differential expression is to flag units with log-ratios greater (in absolute value) than 1, i.e. a change greater than 2-fold when comparing brain vs. universal reference.

```
R>   e <- exprs(eset)
R>   index <- which(eset[["Key"]] == "brain")
R>   d <- rowMeans(e[, index])-rowMeans(e[, -index])
R>   a <- rowMeans(e)
R>   sum(abs(d)>1)
[1] 10043
```

Another approach is to use t-tests to infer whether or not there is differential expression.

```
R>    tt <- rowttests(e, factor(eset[["Key"]]))
R>    lod <- -log10(tt[["p.value"]])
```

The MA plot can be used to visualize the behavior of the log-ratio as a function of average log-intensity. Features with log-ratios greater (in absolute value) than 1 are candidates for being classified as differentially expressed.

```
R>    smoothScatter(a, d, xlab="Average Intensity", ylab="Log-ratio", main="MAQC Sample
R>    abline(h=c(-1, 1), col=2)
```

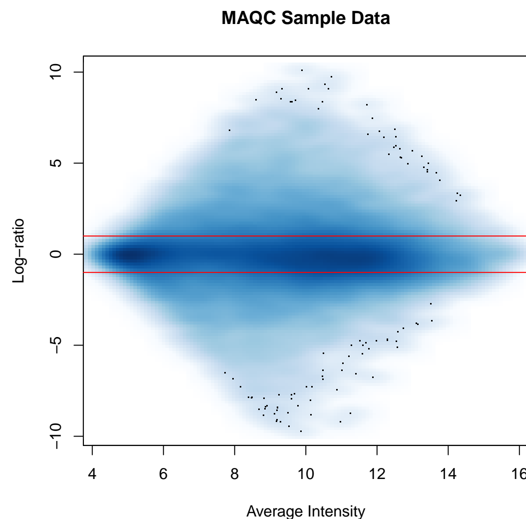


Figure 10: MA plot for Brain vs. Universal Reference. The red lines show the threshold for fold-change of 2, up or down, which correspond to log- fold-change of 1 and -1, respectively.

The use of t-tests allows us to use the volcano plot to visualize candidates for differential expression. Below, we highlight, in blue, the top 25 in log-ratio and, in red, the top 25 in effect size.

The limma Package can also be used to assess difference in expression between the two groups.

```
R>    design <- model.matrix(~factor(eset[["Key"]]))
R>    fit <- lmFit(eset, design)
R>    ebayes <- eBayes(fit)
R>    lod <- -log10(ebayes[["p.value"]][,2])
R>    mtstat<- ebayes[["t"]][,2]
```

The Empirical Bayes approach implemented in limma provides moderated t-statistic, shown to have a better performance when compared to the standard t-statistic. Below, we

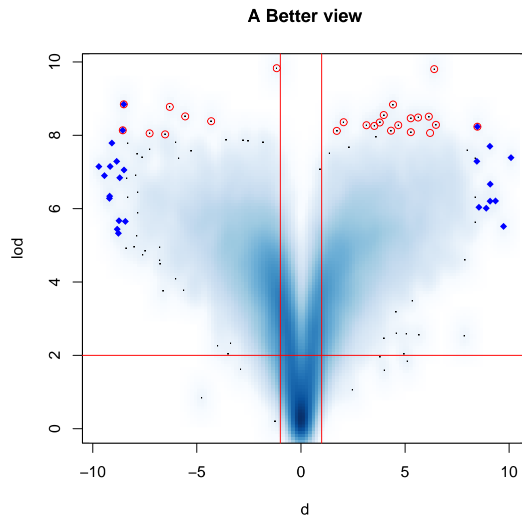


Figure 11: Volcano plot for Brain vs. Universal Reference. The vertical red lines show the threshold for fold-change of 2 (up or down), while the horizontal red line shows the threshold for p-values at the 10^{-2} level. The probesets shown in solid blue diamonds are the top-25 probesets for log-ratio. The probesets highlighted in red are the top-25 in p-value.

reconstruct the volcano plot, but using the moderated t-statisic.

```
R>   o1 <- order(abs(d), decreasing=TRUE)[1:25]
R>   o2 <- order(abs(mtstat), decreasing=TRUE)[1:25]
R>   o <- union(o1, o2)
R>   smoothScatter(d, lod, main="Moderated t", xlab="Log-ratio", ylab="LOD")
R>   points(d[o1], lod[o1], pch=18,col="blue")
R>   points(d[o2], lod[o2], pch=1,col="red")
R>   abline(h=2, v=c(-1, 1))
```

The *topTable* command provides us a way of ranking genes for further evaluation. In the case below, we adjust for multiple testing by FDR and look at the Top-10 genes.

```
R>   tab <- topTable(ebayes, coef=2, adjust="fdr", n=10)
R>   tab
             ID      logFC   AveExpr          t      P.Value
13761 NM_021871   8.513289  8.690249  118.41418 6.065725e-13
746   NM_000806  -8.476382  8.601508 -111.28880 9.413509e-13
169   NM_000184   8.563324  9.195145  110.72598 9.757636e-13
13760 NM_021870   9.084375  9.194213  108.86015 1.100550e-12
10465 NM_014841  -9.077481 10.074374 -106.82013 1.258312e-12
7467  NM_005277 -10.090787  9.892753 -104.77864 1.442549e-12
3286  NM_001034   8.318295  8.903851  102.58414 1.675802e-12
4919  NM_002421   7.271857  8.368350   96.45235 2.592701e-12
9238  NM_007325  -7.997233  9.064384  -96.40369 2.601981e-12
```

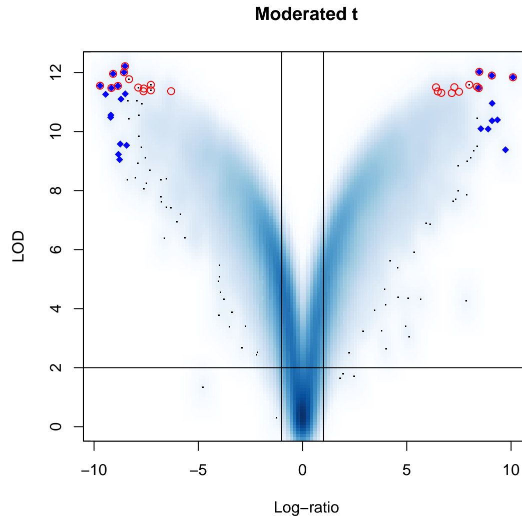


Figure 12: Volcano plot for Brain vs. Universal Reference using moderated t-tests. The vertical red lines show the threshold for fold-change of 2 (up or down), while the horizontal red line shows the threshold for p-values at the 10^{-2} level. The probesets shown in solid blue diamonds are the top-25 probesets for log-ratio. The probesets highlighted in red are the top-25 in p-value (for the moderated t-test). Note that there is more overlap between the top-25 for both log-ratio and p-value.

```
4201  NM_001622   9.710443  9.857097   95.50024 2.781356e-12
         adj.P.Val        B
13761 3.827849e-09 19.04051
746   3.827849e-09 18.81631
169   3.827849e-09 18.79731
13760 3.827849e-09 18.73289
10465 3.827849e-09 18.65980
7467  3.827849e-09 18.58377
3286  3.827849e-09 18.49864
4919  3.827849e-09 18.24048
9238  3.827849e-09 18.23831
4201  3.827849e-09 18.19750
```

3 Obtaining Genotype Calls from SNP Arrays

The oligo package can genotype, using the CRLMM algorithm, several Affymetrix SNP arrays. To do so, the user will need, in addition to the oligo package, an annotation data package specific to the designed used in the experiment. Although these annotation packages

are created using the pdInfoBuilder package, the CRLMM algorithm requires additional hand-curated data, which are included in the packages made available through the BioConductor website. Table 2 describes the supported designs and the respective annotation packages.

Design	Annotation Package
Mapping 50K XBA	pd.mapping50k.xba240
Mapping 50K HIND	pd.mapping50k.hind240
Mapping 250K NSP	pd.mapping250k.nsp
Mapping 250K STY	pd.mapping250k.sty
Genomewide SNP 5.0	pd.genomewidesnp.5
Genomewide SNP 6.0	pd.genomewidesnp.6

Table 2: SNP array designs currently supported by the oligo package and their respective annotation packages. These annotation packages are made available through the BioConductor website and contain hand-curated data, required by the CRLMM algorithm.

As an example, we will use the 269 CEL files, on the XBA array, available on the HapMap website[2], which were downloaded and saved, uncompressed, to a subdirectory called `snpData`. Therefore, we need to instruct the software to look for the files at the correct location. An output directory should also be defined and that is the place where the summary files, including genotype calls and confidences are stored. This output directory, which we chose to call `crlmmResults`, must not exist prior to the CRLMM call, the software will take care of this task.

```
R> library("oligo")
R> fullFilenames <- list.celfiles("snpData", full.names=TRUE)
R> outputDir <- file.path(getwd(), "crlmmResults")
```

Given the always increasing density of the SNP arrays, we developed efficient methods to process these chips, reducing the required amount of memory even for large studies. Using this approach, we process batches of SNPs at a time, saving partial results to disk. We refer the interested reader to **?** for detailed information on the CRLMM algorithm. The genotyping strategy, in summary, has three steps: A) quantile normalizes against a known reference distribution; B) summarizes the data to the SNP-allele level using median polish; C) uses estimated parameters to classify the samples in genotype groups using Mahalanobis

[2] http://www.hapmap.org

distance.

The summaries are average intensities and log-ratios, defined as across allele and within strand, ie:

$$A_s = \frac{\theta_{A,s} + \theta_{B,s}}{2} \quad (1)$$

$$M_s = \theta_{A,s} - \theta_{B,s}, \quad (2)$$

where s defines the strand (antisense or sense). On the genomewide designs, SNP 5.0 and 6.0, the strand information is dropped. These summaries can be obtained via *getA* and *getM* methods, which return arrays with dimensions corresponding to SNPs, samples and strands (if applicable), respectively. These measures are later used for genotyping.

CRLMM involves running an EM algorithm to adjust for average intensity and fragment length in the log-ratio scale. These adjustments may take long time to run, depending on the combination of number of samples and computer resources available. Below, we show the simplest way to call CRLMM, which requires only the file names and output directory.

```
R> if (!file.exists(outputDir))
   crlmm(fullFilenames, outputDir)
```

The `crlmm` method does not return an object to the R session. Instead, it saves the objects to disk, as not all systems are guaranteed to meet the memory requirements that *SnpSuperSet* objects might need. For the user's convenience, the `getCrlmmSummaries` will read the information from disk and make a *SnpCallSetPlus* or *SnpCnvCallSetPlus* object available to the user.

```
R> crlmmOut <- getCrlmmSummaries(outputDir)
R> calls(crlmmOut[1:5,1:2])

             CEU_NA06985_XBA.CEL CEU_NA06991_XBA.CEL
SNP_A-1507972                   3                   3
SNP_A-1510136                   3                   3
SNP_A-1511055                   3                   3
SNP_A-1518245                   2                   3
SNP_A-1641749                   3                   3

R> confs(crlmmOut[1:5,1:2])
```

```
             CEU_NA06985_XBA.CEL CEU_NA06991_XBA.CEL
SNP_A-1507972        0.0009994257        0.0009994068
SNP_A-1510136        0.0009993051        0.0009993733
SNP_A-1511055        0.0009994257        0.0009994257
SNP_A-1518245        0.0009990180        0.0009994257
SNP_A-1641749        0.0009984225        0.0009970547
```

The genotype calls are represented by 1 (AA), 2 (AB) and 3 (BB). The confidence is the predicted probability that the algorithm made the right call.

Summaries generated by the algorithm can also be accessed from the R session. The options for summaries are *"alleleA"*, *"alleleB"*, *"alleleA-sense"*, *"alleleA-antisense"*, *"alleleB-sense"*, *"alleleB-antisense"*. The options *"alleleA"* and *"alleleB"* are only available for SNP 5.0 and SNP 6.0 platforms. The other options are to be used with 50K and 250K arrays.

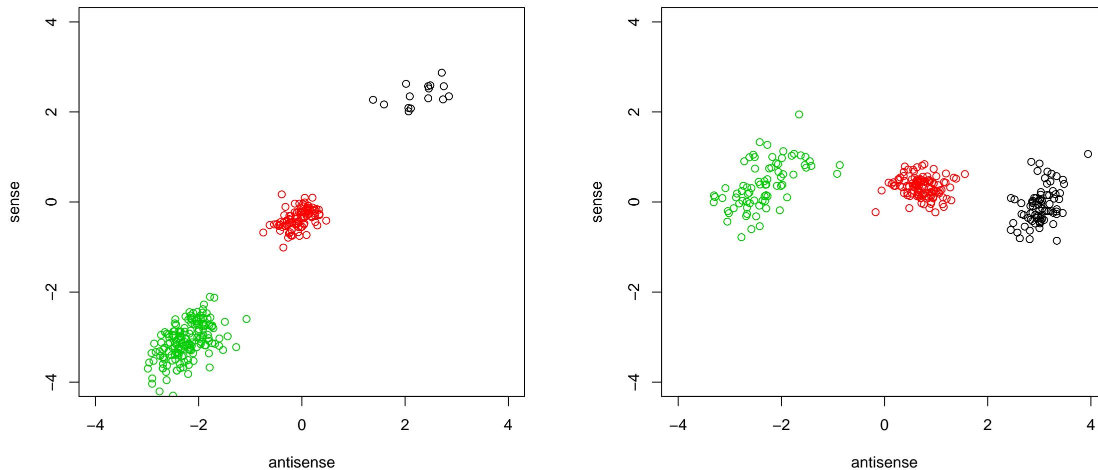
Below, we choose two SNPs to show the different configurations of the genotype groups.

```
R> snps <- paste("SNP_A-", c(1703121, 1725330), sep="")
R> LIM <- c(-4, 4)
```

Figure 13(a) represents a SNP for which genotyping is simplified by the good discrimination of both strands. Figure 13(b) shows a SNP for which features on the antisense strand have very good discrimination power, while no information (for classification) can be extracted from the sense strand.

```
R> gtypes <- as.integer(calls(crlmmOut[snps[1],]))
R> plotM(crlmmOut, snps[1], ylim=LIM, xlim=LIM, col=gtypes)
R> gtypes <- as.integer(calls(crlmmOut[snps[2],]))
R> plotM(crlmmOut, snps[2], ylim=LIM, xlim=LIM, col=gtypes)
```

CRLMM was shown to outperform competing genotyping tools. We refer the reader to **?** for further details on this subject. The genotypes provided by CRLMM, and in this example stored in `crlmmOut`, can be easily used with other BioConductor tools, like the snpStats package, for downstream analyses.



(a) SNP_A-1703121 has very good discrimination on both strands and, as competing algorithms, CRLMM has excelent performance on scenarios like this. On this plot, genotype calls provided by oligo are represented in different colors (black: AA; red: AB; green: BB)

(b) SNP_A-1725330 presents poor discrimination on the sense strand. Because CRLMM does not average across strands, it can perfectly predict the genotype cluster each sample belongs to. On similar scenarios, competing algorithms are known to fail. Color scheme follows Figure 13(a).

4 Preprocessing Exon Arrays

On this section, we use colon cancer sample data for exon arrays, available on the Affymetrix website[3], to demonstrate the use of the oligo package to preprocess these data. The interested reader can download the CEL files and use `read.celfiles` to import the data. Here, however, we will use the oligoData package to load this dataset, as shown below.

```
R> library(oligoData)
R> data(affyExonFS)
```

As already noted, oligo implements different classes depending on the nature of the data. Therefore, a quick inspection, as in the snippet below, shows that `affyExonFS` is an *ExonFeatureSet* object. This is a especially interesting feature, as it allows methods to behave differently depending on the object class.

```
R> affyExonFS
```

[3]`http://www.affymetrix.com/support/technical/sample_data/exon_array_data.affx`

Generally, RMA will background correct, quantile normalize and summarize to the probeset level, as defined in the annotation packages. When working with an *ExonFeatureSet* object, processing to the probeset level provides expression summaries at the exon level and can be obtained by setting the argument *target* to `"probeset"`, as presented below.

```
R> probesetSummaries <- rma(affyExonFS, target="probeset")
```

For Exon arrays, Affymetrix provides additional annotation files that define meta-probesets (MPSs), used to summarize the data to the gene level. These MPSs are classified in three groups – core, extended and full – depending on the level of confidence of the sources used to generate such annotations. Additional values allowed for the *target* argument are `"core"`, `"extended"` and `"full"`. The example below shows how gene level summaries can be obtained through oligo.

```
R> geneSummaries <- rma(affyExonFS, target="core")
```

The results obtained from analyses performed with oligo can be easily combined with features offered by other packages. As an example, we use the biomaRt package to obtain IDs of probesets on the Human Exon array that map to Entrez Gene ID 10948 (ENSG00000131748).

```
R> library(biomaRt)
R> ensembl <- useMart("ensembl", dataset="hsapiens_gene_ensembl")
R> theIDs <- getBM(attributes="affy_huex_1_0_st_v2", filters="entrezgene",
                   values=10948, mart=ensembl)
R> names(theIDs) <- 'psets'
```

Combining this information with the annotation package associated to the data in `affyExonFS`, we can get detailed facts on the probesets found to map to Entrez Gene ID 10948. Below, we obtain, respectively, the MPS IDs, probeset IDs, probe IDs and start/stop positions for the probesets identified above.

```
R> library(AnnotationDbi)
R> conn <- db(affyExonFS)
R> fields <- 'meta_fsetid, pmfeature.fsetid, fid, start, stop'
R> tables <- 'featureSet, pmfeature, core_mps'
R> sql <- paste("SELECT", fields,
                "FROM", tables,
                "WHERE pmfeature.fsetid=featureSet.fsetid",
                "AND featureSet.fsetid=core_mps.fsetid",
                "AND pmfeature.fsetid=:psets")
R> probesetInfo <- dbGetPreparedQuery(conn, sql, theIDs)
```

The availability of start and stop positions of the probesets improves the visualization of the summaries at the exon level. If genomic coordinates were available for probes themselves, visualization could be improved even more. To achieve this, we first obtain the sequences for the probes identified above. We saw that the *pmSequence* method provides the sequences for all PM probes identified on the chip but, instead, we directly load the Biostrings object used to store the sequence information for these probes. This gives us access not only to the sequences, but also to the probe IDs linked to them.

```
R> library(Biostrings)
R> data(pmSequence, package=annotation(affyExonFS))
```

Because probe IDs are available in the `pmSequence` object, we can easily restrict our search to the probes listed in the `probesetInfo` object.

```
R> idx <- match(probesetInfo[["fid"]], pmSequence[["fid"]])
R> pmSequence <- pmSequence[idx,]
```

The `pmSequence` object behaves like a *data.frame*, but it is comprised of complex data structures defined in Biostrings. Below, we modify its representation to make it a regular *data.frame* object.

```
R> pmSequence <- data.frame(fid=pmSequence[["fid"]],
                           sequence = as.character(pmSequence[["sequence"]]),
                           stringsAsFactors=FALSE)
```

By joining the `probesetInfo` and `pmSequence` objects, we centralize the available probe annotation.

```
R> probeInfo <- merge(probesetInfo, pmSequence)
```

The genomic coordinates in `probeInfo` refer to the probesets. To better visualize the observed probe intensities, we would be better off if the coordinates were relative to the probes. Below, we use the BSgenome.Hsapiens.UCSC.hg18 to obtain up-to-date genomic coordinates. The coordinates are found by aligning the probe sequences to the reference genome made available through the package. Because Entrez Gene ID 10948 is located on chromosome 17, the search is limited to this region.

```
R> library("BSgenome.Hsapiens.UCSC.hg19")
R> chr17 <- Hsapiens[["chr17"]]
R> seqs <- complement(DNAStringSet(probeInfo[["sequence"]]))
R> seqs <- PDict(seqs)
```

```
R> matches <- matchPDict(seqs, chr17)
```

After matching the sequences, we update the genomic coordinates.

```
R> probeInfo[["start"]] <- unlist(startIndex(matches))
R> probeInfo[["stop"]] <- unlist(endIndex(matches))
```

With the updated coordinates, we reorder the probe information object, `probeInfo`, and extract the probe intensities in the same order. The probe ID field, `fid` in `probeInfo`, provides direct access to the probes of interest. The *exprs* method is used to access the intensity matrix of the `affyExonFS` object and immediately subsetted to the probes of interest. After subsetting the observed intensities, we $\log_2$-transform the data.

```
R> probeInfo <- probeInfo[order(probeInfo[["start"]]),]
R> probeData <- exprs(affyExonFS)[probeInfo[["fid"]],]
R> probeData <- log2(probeData)
```

We use the updated genomic to estimate the probeset coverage. This information will be used when plotting the data and will provide approximate delimiters of the probesets.

```
R> attach(probeInfo)
R> probesetStart <- aggregate(as.data.frame(start), list(fsetid=fsetid), min)
R> names(probesetStart) <- c("fsetid", "start")
R> probesetStop <- aggregate(as.data.frame(stop), list(fsetid=fsetid), max)
R> names(probesetStop) <- c("fsetid", "stop")
R> detach(probeInfo)
```

The `psInfo` object will store the probeset information (probeset ID, start and stop positions), as shown below. After ordering appropriately the data, the `psInfo` probeset is attached, to simplify its usage during the R session.

```
R> psInfo <- merge(probesetStart, probesetStop)
R> psInfo <- psInfo[order(psInfo[["start"]]),]
R> psInfo[["fsetid"]] <- as.character(psInfo[["fsetid"]])
R> attach(psInfo)
R> probesetData <- exprs(probesetSummaries[fsetid,])
R> detach(psInfo)
```

To visualize the data processed by oligo, we will use the GenomeGraphs package. To match the genome build used to update the probe coordinates, an archived version of the database will be queried.

```
R> library(GenomeGraphs)
R> probeids <- as.character(probeInfo[["fsetid"]])
R> ensembl = useMart("ensembl", dataset="hsapiens_gene_ensembl")
```

```
R> geneid <- "ENSG00000131748"
R> title <- makeTitle(text=geneid, color="darkred")
```

The raw data, in the $\log_2$ scale, will be represented by the `raw` object below, created with the *makeExonArray* constructor.

```
R> attach(probeInfo)
R> raw <- makeExonArray(intensity=probeData,
                        probeStart=start,
                        probeEnd=stop,
                        probeId=probeids,
                        nProbes=rep(1, nrow(probeInfo)),
                        dp=DisplayPars(color="blue", mapColor="dodgerblue2"),
                        displayProbesets=FALSE)
R> detach(probeInfo)
```

The summarized data is also represented through an object created by *makeExonArray*. The structure is identical to the one used above.

```
R> attach(psInfo)
R> exon <- makeExonArray(intensity=probesetData,
                         probeStart=start,
                         probeEnd=stop,
                         probeId=fsetid,
                         nProbes=rep(1, nrow(psInfo)),
                         dp=DisplayPars(color="seagreen",
                           mapColor="seagreen"),
                         displayProbesets=FALSE)
```

To represent the probesets designed by Affymetrix, we use an *AnnotationTrack* object.

```
R> affyModel <- makeAnnotationTrack(start = start,
                                    end = stop,
                                    feature = "gene_model",
                                    group = geneid,
                                    dp = DisplayPars(gene_model="darkgreen"))
R> detach(psInfo)
```

The gene and transcripts representations are build as follows. Affymetrix probes will be represented in green, while the gene will be in orange; transcripts are represented in blue.

```
R> gene <- makeGene(id=geneid, biomart=ensembl)
R> transcript <- makeTranscript(id=geneid, biomart=ensembl)
R> legend <- makeLegend(c("Affymetrix", "Gene"), fill=c("darkgreen", "orange"))
```

Figure 13, generated with the `gdPlot` function, shows the representation of the $\log_2$-intensities and summaries at the exon level. It also shows probesets, gene and transcripts on

the region of interest.

```
R> gdPlot(list(title, raw, exon, affyModel, gene, transcript, legend))
```

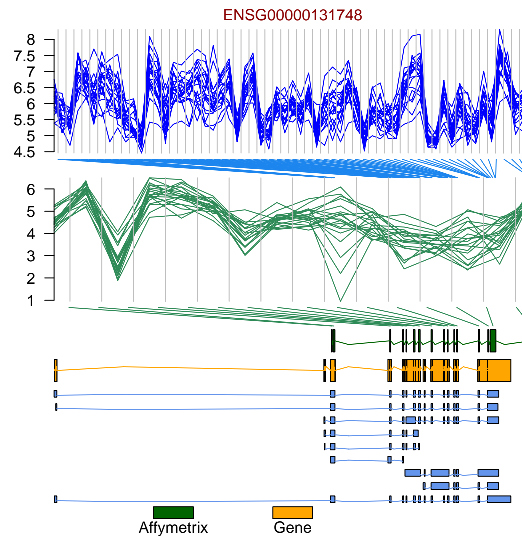


Figure 13: Visual representation of observed $\log_2$-intensities and summarized data at the exon level for gene ENSG00000131748. The probes, gene and transcript are also represented, respectively, in green, orange and blue.

Below, we identify the meta-probeset ID associated to the probes used above. Once that is known, we can extract the proper gene-level summaries stored in geneSummaries.

```
R> mps <- unique(probeInfo[["meta_fsetid"]])
R> mps <- as.character(mps)
R> mps
[1] "3720343" "3720383"
```

Therefore, the standard accessors can be used to obtain the gene summaries for the unit above. Figure 14 shows the expressions for gene ENSG00000131748 across the 33 samples available on this dataset.

```
R> gSummaries <- exprs(geneSummaries[mps,])
R> x <- 1:length(gSummaries)
R> plot(x, gSummaries, xlab="Sample", ylab="Expression", main=geneid)
```

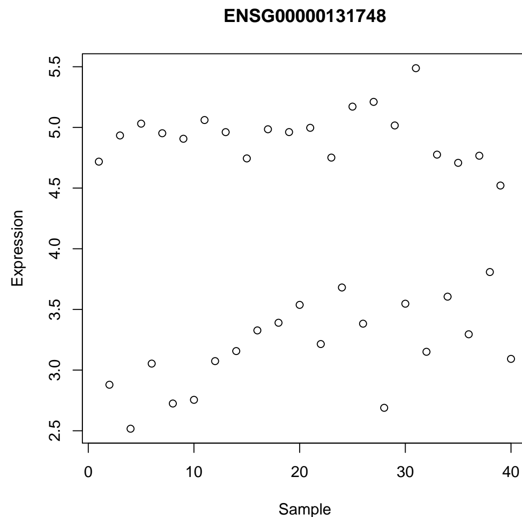


Figure 14: Expression levels estimated through RMA at the gene level.

5 Interfacing with ACME to Find Enriched Regions Using Tiling Arrays

On this Section, we demonstrate how oligo can be easily combined with tools that rely on the structure implemented in the Biobase package. Using a sample ChIP-chip dataset kindly provided by NimbleGen, we could use the `getNgsColorsInfo` function to obtain the information regarding channels and sample names for the XYS files saved on disk. The `getNgsColorsInfo` parses the file names and, using the _532 and _635 strings in the names, suggests channels and sample names for each XYS file available.

```
R> library(oligo)
R> info <- getNgsColorsInfo("tilingData", full=TRUE)
```

Combining the results in `info` with `read.xyfiles2`, we read the XYS files using a data structure that simplifies the data management across different channels.

```
R> nimbleTilingFS <- read.xysfiles2(info[,2], info[,1], sampleNames=info[,3])
```

However, on this example, we will load the aforementioned dataset from the oligoData package, as described below:

```
R> library(oligoData)
R> data(nimbleTilingFS)
R> nimbleTilingFS
```

The user can access the channel specific data by calling the *channel* method. The resulting object is an *ExpressionSet* object that the user can use as required.

```
R> c1 <- channel(nimbleTilingFS, "channel1")
R> c2 <- channel(nimbleTilingFS, "channel2")
```

Detailed information on the PM probes available on the array can be obtained by directly querying the annotation package. The call below will extract the `fid`, `fsetid`, `chromosome` and `start` position of each probe from the annotation package and order the results by chromosome and start position.

```
R> fields <- 'fid, fsetid, chrom as chromosome, position as start'
R> sql <- paste("SELECT", fields,
              "FROM pmfeature INNER JOIN featureSet USING(fsetid)",
              "ORDER BY chrom, position")
R> annotPM <- dbGetQuery(db(nimbleTilingFS), sql)
```

Using the probe sequence, the end position of the probe can be easily obtained. We load the sequences directly, so the `fid` field can be used to order the sequences appropriately.

```
R> data(pmSequence, package=annotation(nimbleTilingFS))
R> idx <- match(annotPM[["fid"]], pmSequence[["fid"]])
R> pmSequence <- pmSequence[idx,]
```

To obtain the end position, we use *width*, defined in the Biostrings package.

```
R> attach(annotPM)
R> library(Biostrings)
R> annotPM[["end"]] <- start+width(pmSequence[["sequence"]])-1
R> head(annotPM)
     fid fsetid chromosome start   end
1 392369   1655       chr1 56753 56808
2 286872   1655       chr1 56853 56909
3 229027   1655       chr1 56953 57007
4 386658   1655       chr1 57053 57114
5  85534   1655       chr1 57153 57202
6 170025   1655       chr1 57253 57307
```

The `fid` field corresponds to the row number in the `nimbleTilingFS` object. When applied to the raw data object, the *getM* function returns a matrix with the $\log_2$-ratio of the intensities. Below, we get the $\log_2$-ratios corresponding to the PM probes described in the `annotPM` object.

```
R> ratioPM <- getM(nimbleTilingFS)[fid,]
R> dimnames(ratioPM) <- NULL
```

```
R> detach(annotPM)
R> class(ratioPM)
[1] "matrix"
```

By converting `annotPM` to an *AnnotatedDataFrame*, it can be used in the *featureData* slot of *eSet*-like objects.

```
R> annotPM <- as(annotPM, "AnnotatedDataFrame")
```

We will use the ACME package to calculate enrichment, using algorithms that are insensitive to normalization strategies and array noise. To work with this package, we need to create, first, an *ACMESet* object, which contains `chromosome`, `start` and `end` positions in the `featureData` slot.

```
R> library(ACME)
R> acme <- new("ACMESet", exprs=ratioPM, featureData=annotPM)
```

The `do.aGFF.calc` function processes the *ACMESet* object, using a window size and threshold to identify the positive probes in the object.

```
R> calc <- do.aGFF.calc(acme, window=1000, thresh=0.95)
```

The `calc` object is then used to find enriched regions with the `findRegions` function, as shown below.

```
R> regs <- findRegions(calc)
R> head(regs)
         Length    TF StartInd EndInd Sample Chromosome  Start    End
1.chr1.1     37 FALSE        1     37      1       chr1  56753 356721
1.chr1.2      7  TRUE       38     44      1       chr1 356821 357621
1.chr1.3      8 FALSE       45     52      1       chr1 357721 611797
1.chr1.4      2  TRUE       53     54      1       chr1 611897 611997
1.chr1.5     11 FALSE       55     65      1       chr1 612097 613097
1.chr1.6      7  TRUE       66     72      1       chr1 613197 613797
               Median         Mean
1.chr1.1 5.164068e-01 3.799430e-01
1.chr1.2 4.321883e-06 3.704507e-06
1.chr1.3 1.451644e-03 4.556099e-03
1.chr1.4 9.630698e-05 9.630698e-05
1.chr1.5 1.776574e-02 1.303595e-01
1.chr1.6 9.630698e-05 5.924166e-05
```

6 High Performance Computing Features

Starting on series 1.12.x, the oligo package offers high performance computing features:

- **Support to larger datasets; and**
- **Support to parallel computing.**

These features are initially available for RMA methods on Expression/Gene/Exon arrays and will be implemented in other methods as necessity arrives.

The use of such features is as simple as loading the required packages (and registering a parallel backend, if parallel computing is desired). The methods themselves are able to detect if these experimental features are enabled and use them if possible, without any modification of the method call.

6.1 Support to large datasets

The oligopackage uses the features implemented by the ff package to provide a better support to large datasets. If the user prefers not to use the ff package, then regular R objects are used and the usual memory restrictions apply.

The support to large datasets is enabled by simply loading the ff package. Once that is done, oligo saves ff files to the directory pointed by `ldPath()`.

```
R> library(oligo)
R> library(ff)
R> ldPath()
```

Methods (*rma*) uses batches to process data. When possible (eg., background correction), it uses at most `ocSamples()` samples simultaneously at processing. For procedures that process probes (like summarization), a maximum of `ocProbesets()` are used simultaneously. Therefore, the user should tune these parameters for a better performance.

```
R> ocSamples()
R> ocSamples(50) ## changing default to 50
R> ocProbesets()
R> ocProbesets(100) ## changing default to 100
```

```
R>   library(oligo)
R>   library(ff)
R>   rawData <- read.celfiles(list.celfiles())
R>   rmaRes <- rma(rawData)
R>   exprs(rmaRes)[1:10,]
```

6.2 Parallel computing

The oligopackage can make use of a parallel environment (with `rma` in the meantime) set via foreach package, as long as the user:

- enables support to large datasets (load ff);
- loads the foreach package;
- register a parallel backend (for example, through one of the doMPI, doMC, doSNOW packages).

A simple example is shown below:

```
R> library(ff)
R> library(foreach)
R> library(doMC)
R> registerDoMC(2)
R> library(oligo)
R> rawData <- read.celfiles(list.celfiles())
R> rmaRes <- rma(rawData)
R> rmaRes
```

6.3 Parallel Computing on Multicore Machines

On multicore machines, one alternative for parallel preprocessing is shown below. It assumes that the machine has enough RAM to deal with the dataset and that the ff package is **NOT** loaded. The snippet compares the performance between a single-threaded run of *rma*, although *fitProbeLevelModel* would also benefit from it, and a run using 4 threads (which is enabled by setting the `R_THREADS` environment variable).

```
R> library(oligoData)
R> data(affyExonFS)
R> t0 <- system.time(res0 <- rma(affyExonFS))

Background correcting
Normalizing
Calculating Expression

R> Sys.setenv(R_THREADS=4)
R> t1 <- system.time(res1 <- rma(affyExonFS))

Background correcting
Normalizing
Calculating Expression

R> all.equal(res0, res1)

[1] TRUE

R> t0

   user  system elapsed
 23.145   1.105  24.251

R> t1

   user  system elapsed
 23.704   1.710  14.249
```

7 Session Info

- R version 2.15.0 beta (2012-03-20 r58793), `x86_64-apple-darwin9.8.0`
- Locale: `C`
- Base packages: base, datasets, grDevices, graphics, grid, methods, stats, utils
- Other packages: ACME 2.11.1, AnnotationDbi 1.17.27, BSgenome 1.23.4, BSgenome.Hsapiens.UCSC.hg19 1.3.17, Biobase 2.15.4, BiocGenerics 0.1.14, BiocInstaller 1.1.28, Biostrings 2.23.6, DBI 0.2-5, GenomeGraphs 1.15.1, GenomicRanges 1.7.42, IRanges 1.13.35, RColorBrewer 1.0-5, RSQLite 0.11.1, biomaRt 2.11.2, genefilter 1.37.1, limma 3.11.19, maqcExpression4plex 1.4.1, oligo 1.19.26, oligoClasses 1.17.39, oligoData 1.6.0,

pd.2006.07.18.hg18.refseq.promoter 1.6.0, pd.hg.u95av2 1.6.0, pd.hg18.60mer.expr 3.0.0, pd.huex.1.0.st.v2 3.6.0

- Loaded via a namespace (and not attached): KernSmooth 2.23-7, RCurl 1.91-1, XML 3.9-4, affxparser 1.27.5, affyio 1.23.2, annotate 1.33.8, bit 1.1-9, codetools 0.2-8, ff 2.2-5, foreach 1.3.5, iterators 1.0.5, preprocessCore 1.17.7, splines 2.15.0, stats4 2.15.0, survival 2.36-12, tools 2.15.0, xtable 1.7-0, zlibbioc 1.1.2