oligo User's Guide

Benilton S. Carvalho

April 11, 2015

Contents

Chapter 1

Introduction

Oligo is a *Bioconductor* package for preprocessing oligonucleotide microarrays. It currently supports chips produced by Affymetrix and NimbleGen and uses files provided by these manufacturers in their native format. The package provides a unified framework for preprocessing and uses the data representation established by the *Bioconductor* project, which simplifies the interface with other packages in the project.

The *oligo* package allows users to preprocess their microarray data using *R*. This is a convenient approach as analysts can combine the preprocessed data with a number of tools already implemented in *R*, like downstream analyses and visualization.

The software is designed to support large datasets and also provides parallel execution of common tasks like background subtraction, normalization and summarization.

This guide describes *oligo* and its features as available on *R* Version 3.2.0 with BioConductor Version 3.1.

Chapter 2

Preamble

2.1 Citing oligo

The *oligo* package is comprised of a collection of tools developed by different authors. Please cite their work appropriately.

If you use *oligo*, please cite:

Carvalho and Irizarry. A Framework for Oligonucleotide Microarray Preprocessing. Bioinformatics (2010) vol. 16 (19) pp. 2363-2367.

If you use the SNPRMA and/or CRLMM algorithm implemented in *oligo*, please also cite:

Carvalho et al. Exploration, normalization, and genotype calls of high-density oligonucleotide SNP array data. Biostatistics (2007) vol. 8 (2) pp. 485-99.

If you use the RMA algorithm, please cite:

Bolstad, B.M., Irizarry R. A., Astrand M., and Speed, T.P. (2003), A Comparison of Normalization Methods for High Density Oligonucleotide Array Data Based on Bias and Variance. Bioinformatics 19(2):185-193;

Rafael. A. Irizarry, Benjamin M. Bolstad, Francois Collin, Leslie M. Cope, Bridget Hobbs and Terence P. Speed (2003), Summaries of Affymetrix GeneChip probe level data Nucleic Acids Research 31(4):e15;

Irizarry, RA, Hobbs, B, Collin, F, Beazer-Barclay, YD, Antonellis, KJ, Scherf, U, Speed, TP (2003) Exploration, Normalization, and Summaries of High Density Oligonucleotide Array Probe Level Data. Biostatistics .Vol. 4, Number 2: 249-264.

If you use the PLM algorithm, please cite:

Bolstad, BM (2004). Low Level Analysis of High-density Oligonucleotide Array Data: Background, Normalization and Summarization. PhD Dissertation. University of California, Berkeley.

fixme: If you use the MAS5 Present/Absent calls...

fixme: If you use the DABG Present/Absent calls...

2.2 Installation

The *oligo* package is available for download through the BioConductor project for all platforms. We recommend the installation of the latest R in order to get the latest features available in the package, which can be installed using `biocLite` as shown below:

```
source('http://www.bioconductor.org/biocLite.R')
biocLite('oligo')
```

oligo is in constant development and the users can obtain a summary of the changes by using the `news` command:

```
## All documented changes
news(package='oligo')
```

2.3 Requirements

oligo depends on a few packages that will be automatically installed if the instructions on Section 2.2 are used. These dependencies are available for all platforms and do not require any intervention for their successful installation.

Once *oligo* is installed, the users will need to install the annotation packages associated to the data they want to import. The annotation packages are built using the *pdInfoBuilder* package, but several of them are available for download on the BioConductor website.

If a user tries to import a dataset for which an annotation package is not installed on the user's system, *oligo* will search for it on the BioConductor website. If the annotation package is found, then *oligo* will download and install it automatically. If the annotation package is not found, *oligo* will return an error and the user is expected to build the one using the *pdInfoBuilder* package. After the package is built, the user must install it before attempting to import the data.

Chapter 3

Getting Started

To get started with *oligo*, one must load the package, which can be achieved with the `library` command:

```
library(oligo)
```

oligo can appropriately handle data files for Affymetrix and NimbleGen designs. The supported formats are CEL (Affymetrix) and XYS (NimbleGen).

3.1 Importing Data

3.1.1 Affymetrix Data

Affymetrix distributes data using CEL files, to simplify the access to these files, *oligo* provides the `list.celfiles` tool, which is a wrapper around `list.files` (consult the documentation for `list.files` to get detailed information on advanced usage). The `list.celfiles` command should be used to obtain the list of CEL files at a given directory. We strongly recommend the use of fully qualified names (i.e., including the whole path) for CEL files, to minimize the chance of problems. The snippet below shows the syntax to list CEL files in the hypothetical directory `myCELs`:

```
celFiles <- list.celfiles('myCELs', full.names=TRUE)
```

The CEL files can be in either binary or text formats. Regardless the internal structure of the files, *oligo* can import them transparently via the command `read.celfiles` as shown below:

```
rawData <- read.celfiles(celFiles)
```

3.1.2 NimbleGen Data

The NimbleGen data supported by oligo is provided as XYS files. They are produced through the NimbleScan software from the TIFF image and NDF specification file. The `list.xysfiles` function

can be used to simplify the access to the XYS files. If a hypothetical directory `myXYSs` contains the XYS files for a given dataset, the suggested approach to point to these files is as follows:

```
xysFiles <- list.celfiles('myXYSs', full.names=TRUE)
```

The files listed in `xysFiles` can then be imported using the `read.xysfiles` command:

```
rawData <- read.xysfiles(xysFiles)
```

3.2 Containers for Raw Data

oligo uses different containers to store data at the feature-level, i.e. data imported from CEL and/or XYS files, as Table 3.1 shows. This approach improves the flexibility of the package as it allows any method to behave differently depending on the type of array from which the data were obtained. As a consequence, the user benefits from the simplicity of the software, as algorithms should be able to handle data appropriately independent of their origin.

Type	**Array**
ExonFeatureSet	Exon ST
ExpressionFeature	Expression
GeneFeatureSet	Gene ST
TilingFeatureSet	Tiling
SnpFeatureSet	SNP

Table 3.1: Types of containers for feature-level data used in oligo.

One simple example is the RMA algorithm. When it is applied to expression data, the software software uses the usual definition of sets of features (often referred to as *probesets*) to group features together for summarization. If the same method is applied to Affymetrix exon arrays, the software is able to identify that and use the definition of *meta-probesets* given by Affymetrix to provide summaries at the transcript level, if such behavior is requested.

Chapter 4

Visualization and QC Tools

On this chapter, we will demonstrate how *oligo* can be used for visualization of data at the feature-level.

To demonstrate the capabilities of the software, the `affyExpressionFS` dataset from the *oligoData* package will be used.

```
library(oligoData)
data(affyExpressionFS)
```

This dataset is comprised of 59 samples on expression arrays provided by Affymetrix. This dataset is the *Human Genome U95 Data Set*, used to validade preprocessing algorithms, as it contains genes that were spiked-in in known concentrations. Below we create a table containing sample information, using descriptors found on the Affymetrix website.

The user must pay attention to the fact that the objects handled by *oligo* always carry information about channels. This information must be reported on a metadata object, which is represented below by the `metadata` data.frame. Because Affymetrix expression arrays are one-color devices and the information we provide is valid for this channel, we fill the `channel` column with the value `ALL`.

```
affyExpressionFS
```

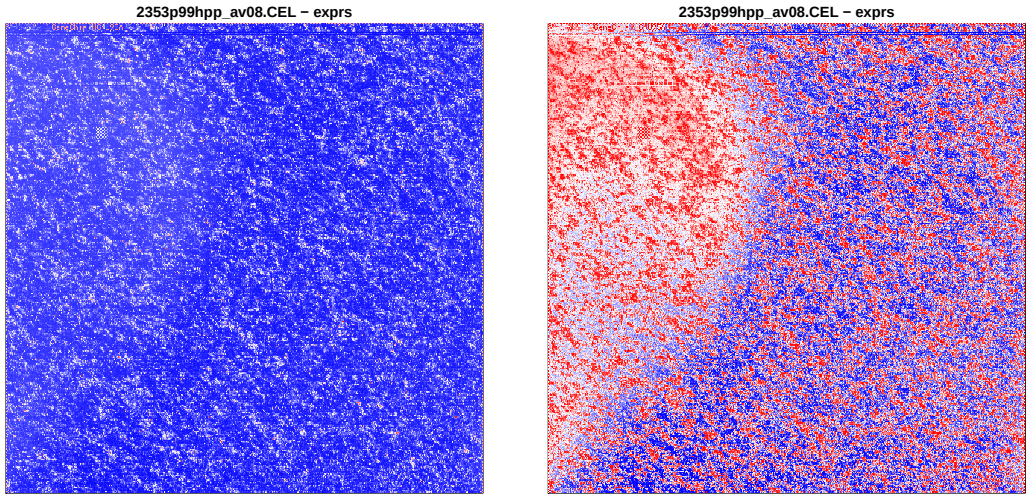
```
## ExpressionFeatureSet (storageMode: lockedEnvironment)
## assayData: 409600 features, 59 samples
##   element names: exprs
## protocolData: none
## phenoData
##   rowNames: 1521a99hpp_av06.CEL 1521b99hpp_av06.CEL ... 2353t99hpp_av08.CEL (59
##     total)
##   varLabels: exprs
##   varMetadata: labelDescription channel
## featureData: none
## experimentData: use 'experimentData(object)'
## Annotation: pd.hg.u95av2
```

```
sns <- sampleNames(affyExpressionFS)
```

```
## all 1521 were meant to be 1251
sns <- gsub('1521', '1251', sns)
## removing the 'r' (repeat) flag from the name
sns <- gsub('r\\.CEL$', '\\.CEL', sns)
wafer <- substr(sns, 1, 4)
experiment <- substr(sns, 5, 5)
tmp <- substr(sns, 6, 7)
complex <- rep('+', length(tmp))
complex[tmp == '00'] <- '-'
info <- data.frame(wafer=wafer, experiment=experiment, complex=complex)
rownames(info) <- sns
metadata <- data.frame(labelDescription=c('wafer', 'experiment', 'complex'), channel=factor
sampleNames(affyExpressionFS) <- sns
pd <- new('AnnotatedDataFrame', data=info, varMetadata=metadata)
phenoData(affyExpressionFS) <- pd
rm(tmp, wafer, experiment, complex, pd, metadata)
```

4.1 Pseudo-image Plots

Pseudo-image plots are used to assess the spatial distribution of the data on the chips. Due to the magnitude of the readings, pseudo-images using data on the original scale often mask spatial features that may be present on the arrays. This is why we recommend the use of the default $\log_2$-scale of the `image` method. One useful alternative for the $\log_2$-scale pseudo-image is the use of the ranks of the observations. This can be achieved by setting the `transfo` argument on the `image` method.



(a) $\log_2$-intensities (b) Rank

Figure 4.1: Pseudo-images

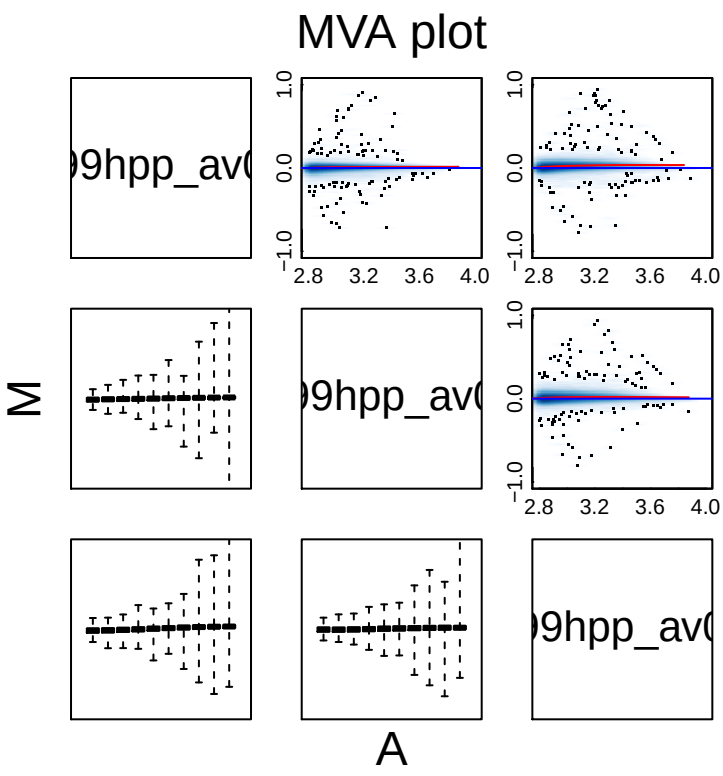


Figure 4.2: MA Plot using smoothScatter

4.2 MA Plots

Plotting log-ratios, M, *versus* average log-intensities, A, is a strategy to visualize the relationship between these two variables. Both M and A are computed as a function of a reference. To illustrate this, the definitions of log-ratios and average log-intensities of a generic sample, indexed by i, and a given reference, R, are given below:

$$M_{i,R} = \log I_i - \log I_R \tag{4.1}$$

$$A_{i,R} = \frac{1}{2}\left[\log\left(I_i\right) + \log\left(I_R\right)\right] \tag{4.2}$$

For one color arrays, one common approach is to create MA plots for every combination of two samples on the (sub-)dataset of interest. On the snippet below, we use the `pairs` argument to generate MA plots for pairs of samples (restricted to the first three samples, which belong to the same group).

```
xl <- c(2.8, 4)
yl <- c(-1, 1)
MAplot(affyExpressionFS[, 1:3], pairs=TRUE, ylim=yl, xlim=xl)
```

The standard approach to plot data used by `MAplot` is to use `smoothScatter`, which provides better visualization through the use of 2-D smoothed densities. This behavior can be changed by setting the

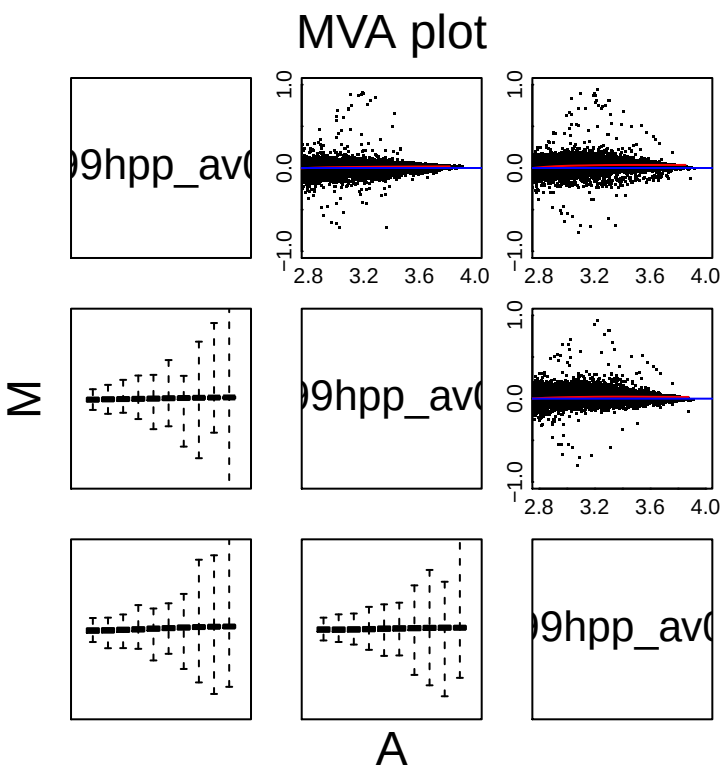


Figure 4.3: MA Plot using points

`plotFun` argument, as shown below. Valid values for this argument are functions that preferentially take the same arguments as `smoothScatter`, like the `plot` function.

```
MAplot(affyExpressionFS[, 1:3], pairs=TRUE, ylim=yl, xlim=xl, plotFun=plot)
```

The `MAplot` method also allows the combination of data into groups. With the code below, the investigator obtains the MA plot for the first levels of `wafer`, *'1251'*, comparing the results against the reference group, *2353*. Note that `wafer` is a factor and that the definition of a reference group was arbitrary and used here just to illustrate the software capabilities.

```
wafer <- affyExpressionFS$wafer
levels(wafer)
```

```
## [1] "1251" "1532" "2353"
```

```
MAplot(affyExpressionFS, groups=wafer, which=1, refSamples=3)
```

When the `groups` argument is not set and the `pairs` argument is set to `FALSE`, the `MAplot` method estimates a pseudo-reference sample from the whole dataset passed to the function. The pseudo-reference sample and the group summaries (if `groups` is defined) are estimated using the `summaryFun` argument, which must be a function that takes an $N \times C$ matrix and returns a vector of length N. The default value for `summaryFun` is `rowMedians`.

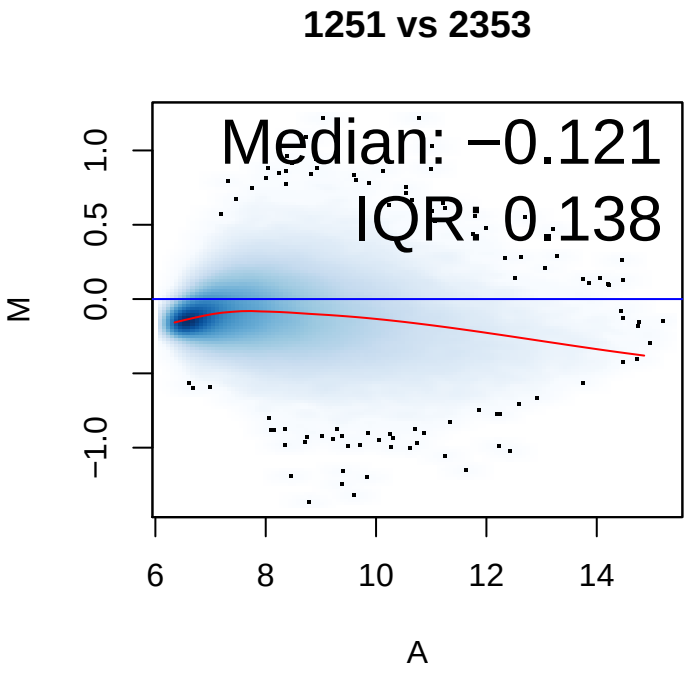


Figure 4.4: MA Plot comparing groups

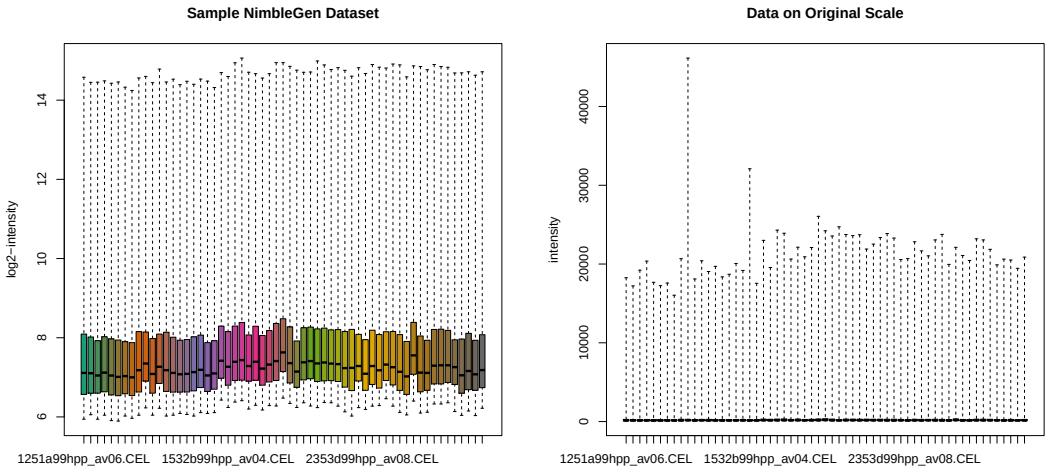
4.3 Boxplots

Boxplots are used to visualize key components on the distribution of the data and simplify the comparison of such statistics across samples.

The call above produces a boxplot for the PM features. If the array contains other features types, the `boxplot` method can be used to generate figures for specific probe types by using the `which` argument, which take values `'pm'`, `'mm'`, `'bg'`, `'both'` and `'all'`.

Data transformation can also be applied. The default is to log-transform (base 2) the data, but other functions can be used, as long as they are passed through the `transfo` argument. The example below presents the boxplot using the original scale.

The `boxplot` method for *FeatureSet* and *ExpressionSet* objects uses a sample of the data (of size `nsample`) to produce the plot. Therefore small differences between consecutive calls to the method are expected. Users interested in getting the exact same plot should specify a fixed seed through `set.seed` prior to calling `boxplot`.



(a) log-scale (b) original scale

Figure 4.5: Boxplots for intensity data: the visualization of the data is simplified if the logarithmic-scale is used

4.4 Density Plots

Smoothed histograms are also used to assess the distribution of the data under analysis. They allow the immediate visualization (possibly non-unique) modes, which can not be reliably detected through the investigation of boxplots and other graphical tools.

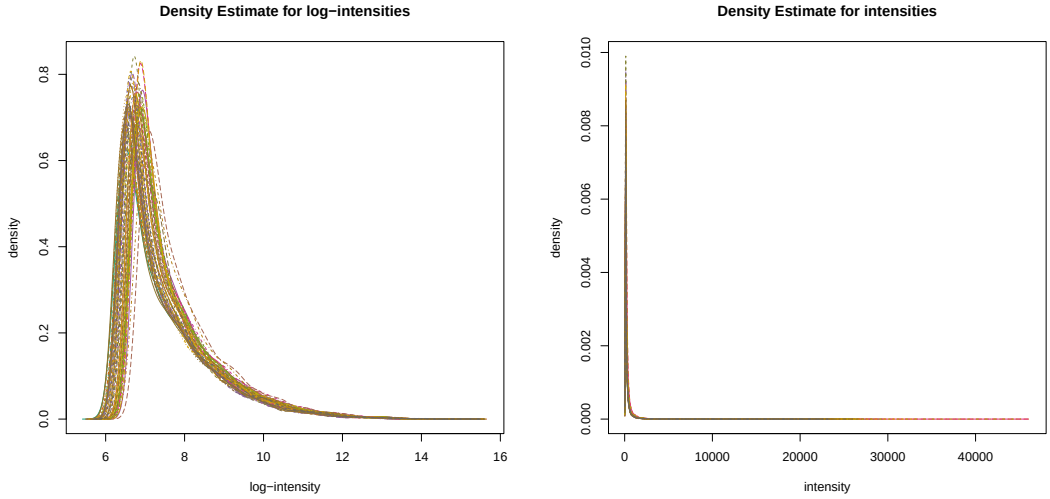
Similar to the `boxplot` method described on Section 4.3, `hist`:

- allows subsetting by feature type, if such probes are available on the chip, through the `which` argument;
- uses a random sample of the data to generate the plot, requiring the use of `set.seed` to create reproducible charts. The size of the sample is determined by the `nsample` argument;
- permits the use of functions other than $\log_2$ to transform the data prior to plotting. The argument `transfo` is the one that handles the transformation function, which should return an object with the same attributes as the input.

4.5 Accessing Probe Sequences

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.

```
pmSeq <- pmSequence(affyExpressionFS)
pmSeq[1:5]
```



(a) log-scale (b) original scale

Figure 4.6: Density plots for intensity data: the visualization of the data is simplified if the logarithmic-scale is used

```
##   A DNAStringSet instance of length 5
##     width seq
## [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`.

```
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 [**?**].

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

On Figure 4.7, 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`.

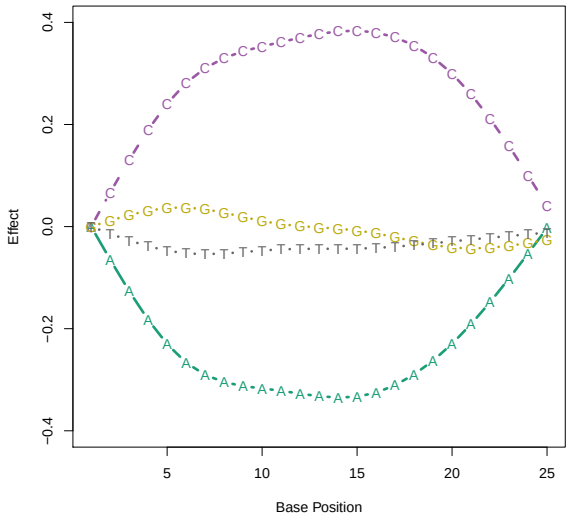


Figure 4.7: Sequence effect for the first sample in the dataset. These results have been reported in detail elsewhere, but can be easily reproduced with the use of the *oligo* package.

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*.

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

In addition to Figure 4.7, we can also plot the $\log_2$-intensities as a function of the GC content computed above. Figure 4.8 presents the strong dependency of $\log_2$-intensities on GC contents for sample 1, which is also present in all other samples.

4.6 Probe Level Models

Using the `fitProbeLevelModel` method, the user is able to fit Probe Level Models (PLMs) with probe-level and sample-level parameters. The resulting object is an *oligoPLM* object, which stores parameter estimates, residuals and weights.

4.6.1 Fitting PLMs

The simplest call to adjust a probe level model is as simple as

```
fit1 <- fitProbeLevelModel(affyExpressionFS)
```

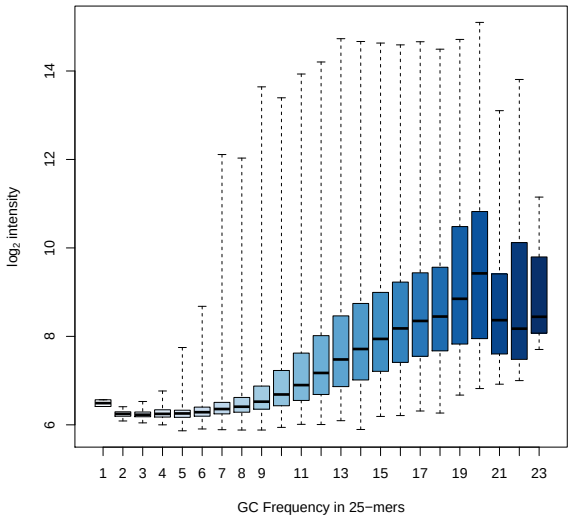


Figure 4.8: On this boxplot stratified by GC content, we can observe the strong dependency of $\log_2$-intensities on the number of G or C bases observed in the probe sequency.

```
## Warning:  'isIdCurrent' is deprecated.
## Use 'dbIsValid' instead.
## See help("Deprecated")
```

```
## Background correcting...  OK
## Normalizing...  OK
## Summarizing...  OK
## Extracting...
##  Estimates...  OK
##  StdErrors...  OK
##  Weights.....  OK
##  Residuals...  OK
##  Scale.......  OK
```

and will fit a model that accounts for probe (feature) and sample effects, whose estimates and standard errors can be recovered, respectively, with the `coef` and `se` methods, as shown below.

```
coef(fit1)[1:4, 1:2]
```

```
##           1251a99hpp_av06.CEL 1251b99hpp_av06.CEL
## 100_g_at             7.500033            7.386941
## 1000_at              6.818087            6.569597
## 1001_at              4.952848            4.899599
## 1002_f_at            5.341271            5.264827
```

```
se(fit1)[1:4, 1:2]
```

```
##           1251a99hpp_av06.CEL 1251b99hpp_av06.CEL
## 100_g_at           0.05769347          0.05841853
## 1000_at            0.07123716          0.07192502
## 1001_at            0.09138853          0.09233888
## 1002_f_at          0.09297926          0.09170566
```

4.6.2 Visualizing `fitProbeLevelModel` Results

One of the most used QC metrics is the Relative Log Expression (RLE), which is computed (for each sample on every probeset) by comparing the expression level of one probeset against the median expression of that probeset across samples.

The estimates obtained via RLE can be accessed by setting the argument `type` to `values`. By setting this argument to `stats`, the user will be able to access the statistics (quantiles) for each sample.

```
RLE(fit1, type='stats')[, 1:2]
```

```
##      1251a99hpp_av06.CEL 1251b99hpp_av06.CEL
## 0%          -6.979272632        -6.507382388
## 25%         -0.070293591        -0.100300464
## 50%          0.006905502         0.001414062
## 75%          0.082904157         0.105041242
## 100%         1.390826287         2.179719560
```

```
RLE(fit1, type='values')[1:4, 1:2]
```

```
##           1251a99hpp_av06.CEL 1251b99hpp_av06.CEL
## 100_g_at           0.12183567         0.008743574
## 1000_at            0.08703757        -0.161453002
## 1001_at           -0.18044837        -0.233698152
## 1002_f_at          0.08315700         0.006713455
```

Generating a boxplot of the RLE values is the default behavior of the method.

Another useful tool for QC is the Normalized Unscaled Standard Errors (NUSE). To determine NUSE, the standard error estimates are standardized across arrays so that the median standard error for that probeset is 1 across all arrays. Therefore, arrays whose NUSE values are significantly higher than other samples are often lower quality chips. Similarly to RLE, the statistics, values and boxplot of NUSE can be obtained by appropriately setting the `type` argument of the `NUSE` method.

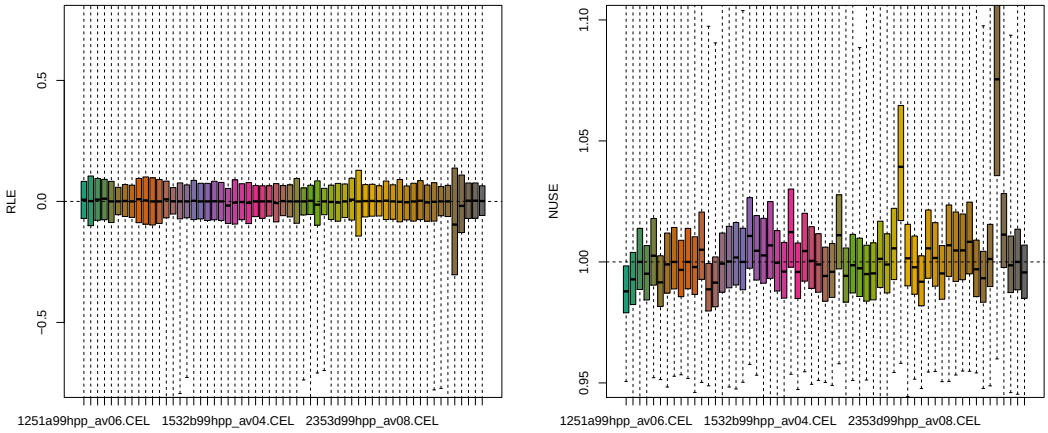
```
NUSE(fit1, type='stats')[, 1:2]
```

```
##      1251a99hpp_av06.CEL 1251b99hpp_av06.CEL
## 0%             0.9506179           0.9301493
## 25%            0.9789673           0.9823609
## 50%            0.9878402           0.9927843
## 75%            0.9983611           1.0039378
```

```
## 100%           1.2025932           1.1917781
```

```
NUSE(fit1, type='values')[1:4, 1:2]
```

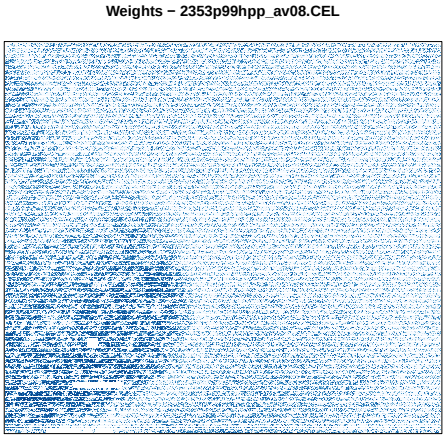
```
##           1251a99hpp_av06.CEL 1251b99hpp_av06.CEL
## 100_g_at            0.9700310           0.9822220
## 1000_at             0.9835805           0.9930779
## 1001_at             0.9780951           0.9882664
## 1002_f_at           1.0115476           0.9976917
```



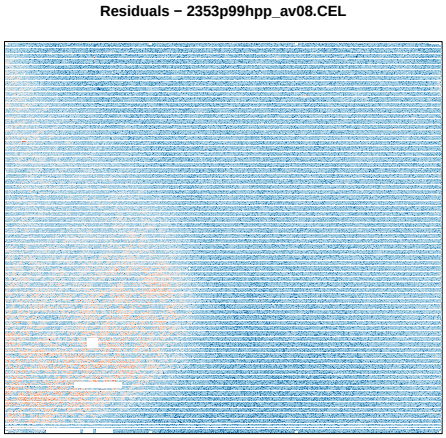
(a) RLE (b) NUSE

Figure 4.9: Visualization of PLM results

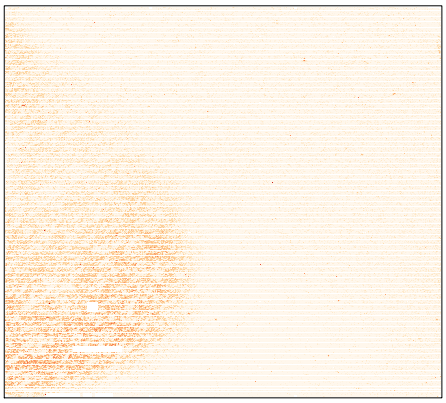
The use of PLMs also permits the inspection of the spatial distribution of the data on the chip. The current implementation allows the visualization of the estimated weights and residuals. Residuals can be further decomposed in 4 types: residuals, positive residuals, negative residuals and residual signs. Figures 4.10(a)-4.10(e) show these plots for the dataset used as example here.



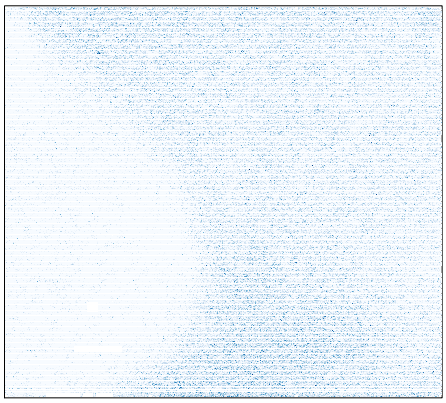
(a) Weights



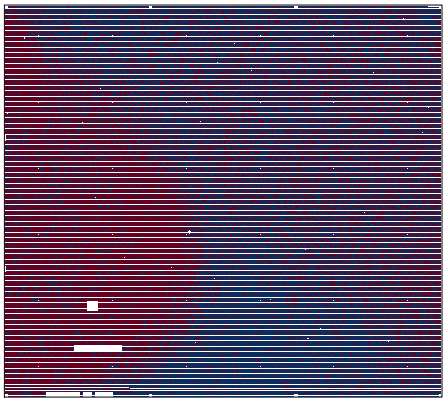
(b) Residuals



(c) Positive Residuals



(d) Negative Residuals



(e) Residual Signs

Figure 4.10: Pseudo-images for PLM objects

Chapter 5

Preprocessing

Preprocessing refers to a series of complex statistical procedures applied to microarray data prior to dowstream analyses. These steps are required mainly for two reasons: A) technical artifacts are known to affect results, so background subtraction and normalization are used to minimize these issues; and B) there are multiple probes per probeset, therefore summarization to the probeset level is needed, so downstream analyses can be carried on.

5.1 Background Subtraction

The *oligo* package implements background subtraction through the `backgroundCorrect` command. The method currently available is the one used in RMA, which treats the PM intensities as a convolution of noise and true signal. Additional methods will be available on future releases and choices will be made with the `method` argument (currently, the default is `'rma'`).

```
backgroundCorrectionMethods()
```

```
## [1] "rma"  "mas"  "LESN"
```

```
bgData1 <- backgroundCorrect(affyExpressionFS)
```

```
## Background correcting...  OK
```

```
bgData2 <- backgroundCorrect(affyExpressionFS, method='mas')
#bgData3 <- backgroundCorrect(affyExpressionFS, method='LESN')
```

Because the input was an *ExpressionFeatureSet* object, the output `bgData1` is also an *ExpressionFeatureSet*. Below, we show a boxplot of the corrected data, which can be compared to Figure **??**.

```
boxplot(bgData1)
```

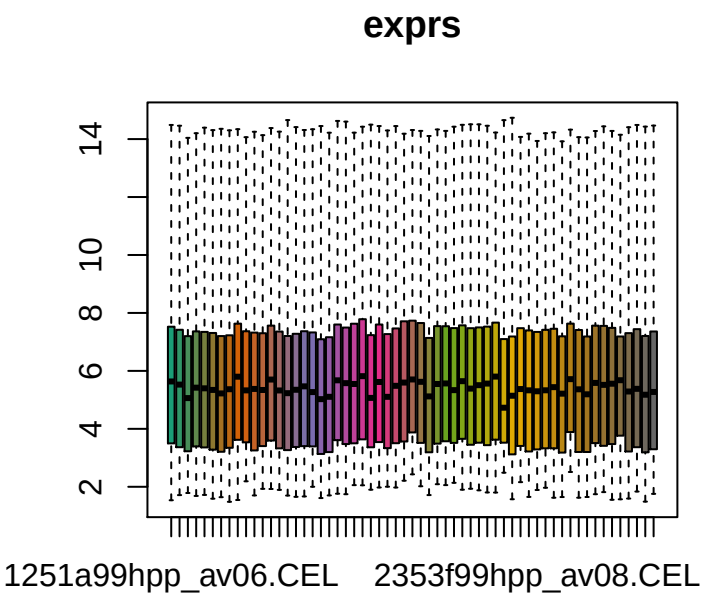


Figure 5.1: Boxplot of background corrected data

5.2 Normalization

The Rmethodnormalize method provided by *oligo* allows the user to normalize the input data. Different normalization methods are available. The available options are given by `normalizationMethods` and the argument `method` in `normalize` is used to select the normalization approach to be used.

```
normData <- normalize(bgData1)

## Normalizing...   OK
```

5.3 Summarization

fixme: to finish

Chapter 6

Workflows

6.1 Preprocessing Affymetrix Expression Arrays

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.

```
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.

```
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* 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.

6.2 Preprocessing NimbleGen Expression Arrays

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.

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.

```
  library(oligo)
  library(maqcExpression4plex)
```

```
## /-----------------------------------------\
## | SAMPLE EXPRESSION DATA - MAQC/ HG18 - NGS |
## |-----------------------------------------|
## | Data provided by NimbleGen Systems (NGS). |
## | This package is meant to be used only for |
## | demonstration of BioConductor packages.  |
## |-----------------------------------------|
## | The contents of this package are provided |
## | in good faith and the maintainer does not |
## | warrant their accuracy.                  |
## \-----------------------------------------/
```

```
  library(genefilter)
```

```
##
## Attaching package:  'genefilter'
##
## The following object is masked from 'package:base':
##
##    anyNA
```

```
  library(limma)
```

```
##
## Attaching package:  'limma'
##
## The following object is masked from 'package:oligo':
##
##    backgroundCorrect
```

```
##
## The following object is masked from 'package:BiocGenerics':
##
##     plotMA
```

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`.

```
extdata <- system.file("extdata", package="maqcExpression4plex")
xys.files <- list.xysfiles(extdata, full.names=TRUE)
basename(xys.files)
```

```
## [1] "9868701_532.xys" "9868901_532.xys" "9869001_532.xys" "9870301_532.xys"
## [5] "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*.

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

```
## Loading required package:  pd.hg18.60mer.expr
## Platform design info loaded.
```

```
## Checking designs for each XYS file... Done.
## Allocating memory... Done.
## Reading /Library/Frameworks/R.framework/Versions/3.2/Resources/library/maqcExpression4pl
## Reading /Library/Frameworks/R.framework/Versions/3.2/Resources/library/maqcExpression4pl
## Reading /Library/Frameworks/R.framework/Versions/3.2/Resources/library/maqcExpression4pl
## Reading /Library/Frameworks/R.framework/Versions/3.2/Resources/library/maqcExpression4pl
## Reading /Library/Frameworks/R.framework/Versions/3.2/Resources/library/maqcExpression4pl
## Reading /Library/Frameworks/R.framework/Versions/3.2/Resources/library/maqcExpression4pl
```

```
class(maqc)
```

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

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.

```
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.

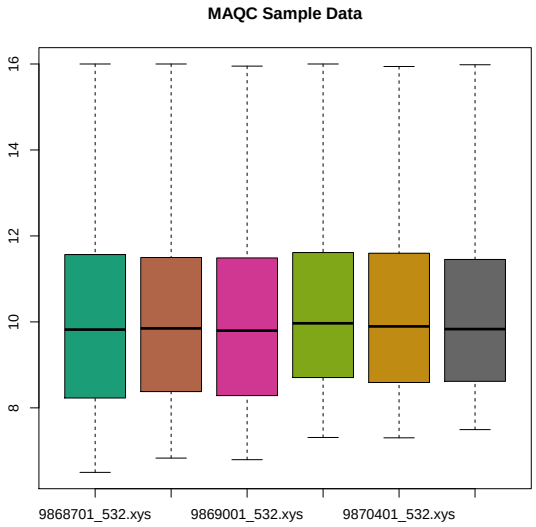


Figure 6.1: Distribution of $\log_2$-intensities of samples on the MAQC dataset.

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

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

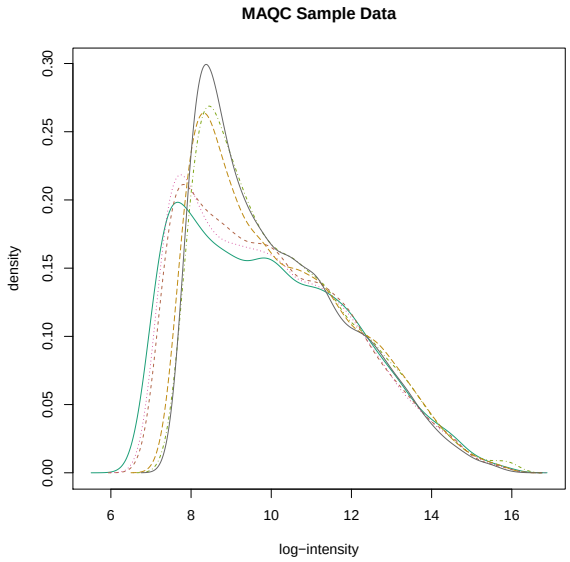


Figure 6.2: Smoothed histogram of $\log_2$-intensities of samples on the MAQC dataset.

via median polish. The result of `rma` is an instance of *ExpressionSet* class, which also contains an `exprs` slot and method.

```
  eset <- rma(maqc)
```

```
## Background correcting
## Normalizing
## Calculating Expression
```

```
  class(eset)
```

```
## [1] "ExpressionSet"
## attr(,"package")
## [1] "Biobase"
```

```
  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
```

```
  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.

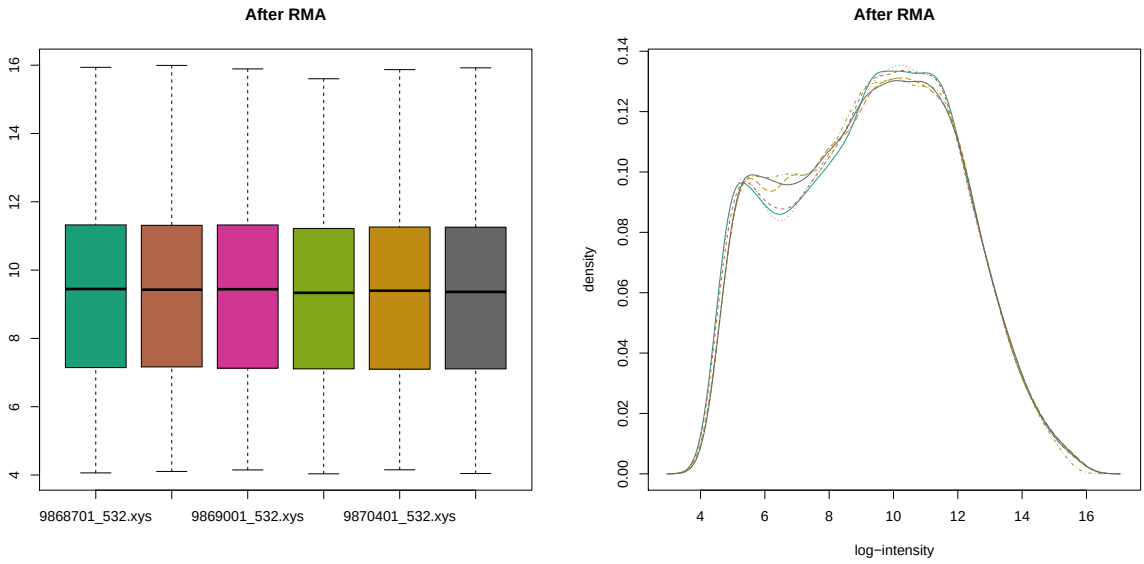


Figure 6.3: Boxplot and smoothed histogram for MAQC data after preprocessing.

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.

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

```
## [1] 10043
```

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

```
  tt <- rowttests(e, factor(eset[["Key"]]))
  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.

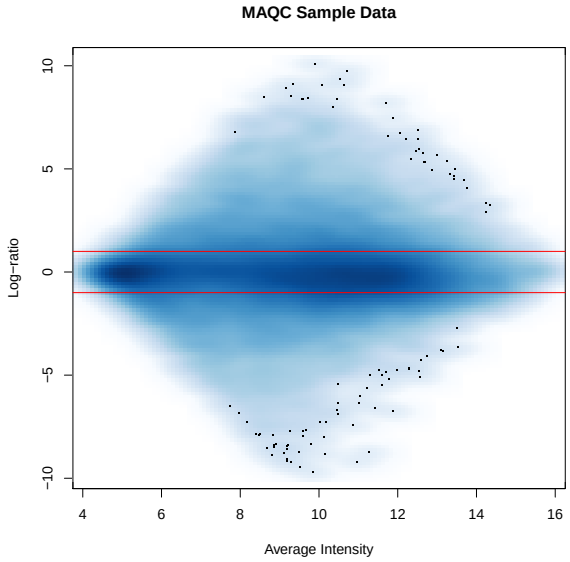


Figure 6.4: MA plot for Brain vs. Universal Reference. The red lines show the threshold for fold-change of 2, up or down, which correspond to log- fold-change of 1 and -1, respectively.

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.

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

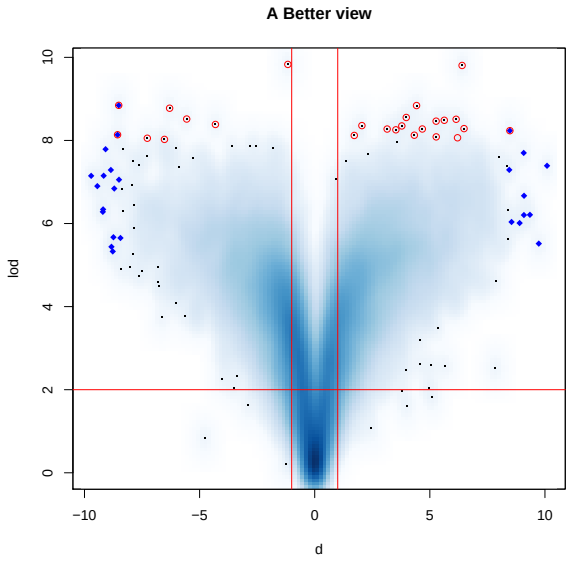


Figure 6.5: Volcano plot for Brain vs. Universal Reference. The vertical red lines show the threshold for fold-change of 2 (up or down), while the horizontal red line shows the threshold for p-values at the 10^{-2} level. The probesets shown in solid blue diamonds are the top-25 probesets for log-ratio. The probesets highlighted in red are the top-25 in p-value.

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

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.

```
  tab <- topTable(ebayes, coef=2, adjust="fdr", n=10)
  tab
```

```
##                logFC   AveExpr          t      P.Value    adj.P.Val        B
## NM_021871   8.513289  8.690249  118.41418 6.065725e-13 3.827849e-09 19.04051
## NM_000806  -8.476382  8.601508 -111.28880 9.413509e-13 3.827849e-09 18.81631
## NM_000184   8.563324  9.195145  110.72598 9.757636e-13 3.827849e-09 18.79731
## NM_021870   9.084375  9.194213  108.86015 1.100550e-12 3.827849e-09 18.73289
## NM_014841  -9.077481 10.074374 -106.82013 1.258312e-12 3.827849e-09 18.65980
## NM_005277 -10.090787  9.892753 -104.77864 1.442549e-12 3.827849e-09 18.58377
## NM_001034   8.318295  8.903851  102.58414 1.675802e-12 3.827849e-09 18.49864
## NM_002421   7.271857  8.368350   96.45235 2.592701e-12 3.827849e-09 18.24048
## NM_007325  -7.997233  9.064384  -96.40369 2.601981e-12 3.827849e-09 18.23831
## NM_001622   9.710443  9.857097   95.50024 2.781356e-12 3.827849e-09 18.19750
```

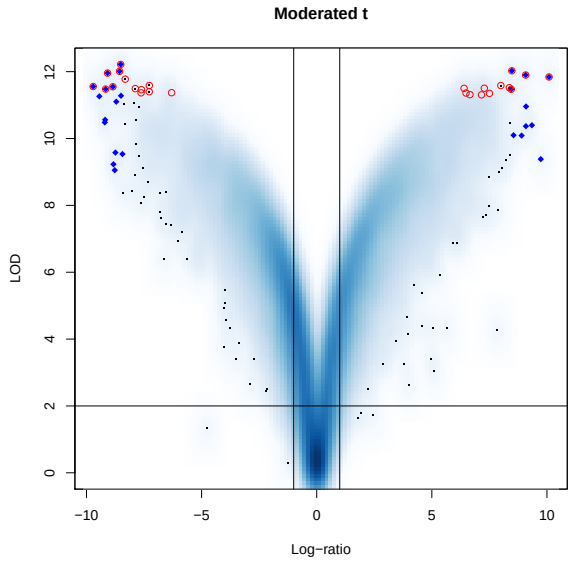


Figure 6.6: Volcano plot for Brain vs. Universal Reference using moderated t-tests. The vertical red lines show the threshold for fold-change of 2 (up or down), while the horizontal red line shows the threshold for p-values at the 10^{-2} level. The probesets shown in solid blue diamonds are the top-25 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.

6.3 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 6.1 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 6.1: 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[1], 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.

comment: Given the restrictions for building this vignette, we replaced the big example containing 269 samples by a smaller example with 3 samples only.

```
library("oligo")
#fullFilenames <- list.celfiles("snpData", full.names=TRUE)
path <- system.file('celFiles', package='hapmap100kxba')
fullFilenames <- list.celfiles(path, full.names=TRUE)
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 distance.

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

$$A_s = \frac{\theta_{A,s} + \theta_{B,s}}{2} \tag{6.1}$$

$$M_s = \theta_{A,s} - \theta_{B,s}, \tag{6.2}$$

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.

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

```
## Loading required package:  pd.mapping50k.xba240
```

```
## Warning:  'isIdCurrent' is deprecated.
## Use 'dbIsValid' instead.
## See help("Deprecated")
```

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

```
## Preparing environment for normalization.
## Normalization.
## Genotyping.

## Warning:  'isIdCurrent' is deprecated.
## Use 'dbIsValid' instead.
## See help("Deprecated")

## Warning:  'isIdCurrent' is deprecated.
## Use 'dbIsValid' instead.
## See help("Deprecated")
```

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.

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

##               NA06985.CEL NA06991.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

confs(crlmmOut[1:5,1:2])

##                NA06985.CEL  NA06991.CEL
## SNP_A-1507972 0.0009994257 0.0009993900
## SNP_A-1510136 0.0009994257 0.0009994257
## SNP_A-1511055 0.0009994257 0.0009994257
## SNP_A-1518245 0.0009992808 0.0009994257
## SNP_A-1641749 0.0009992852 0.0009992566
```

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.

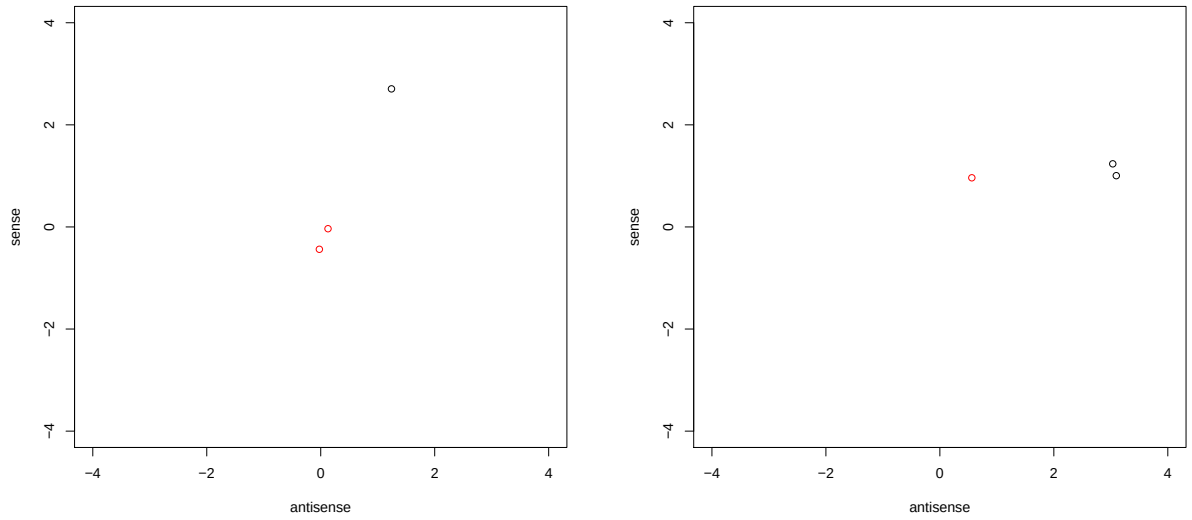
Summaries generated by the algorithm can also be accessed from the *R* session. The options for summaries are `"alleleA"`, `"alleleB"`, `"alleleA-sense"`, `"alleleA-antisense"`, `"alleleB-sense"`, `"alleleB-antisense"`. The options `"alleleA"` and `"alleleB"` are only available for SNP 5.0 and SNP 6.0 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.

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

Figure 6.7(a) represents a SNP for which genotyping is simplified by the good discrimination of both

strands. Figure 6.7(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.



(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 6.7(a).

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.

6.4 Preprocessing Exon Arrays

On this section, we use colon cancer sample data for exon arrays, available on the Affymetrix website[2], 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.

```
library(oligoData)
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

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

is a especially interesting feature, as it allows methods to behave differently depending on the object class.

```
affyExonFS
```

```
## Loading required package:  pd.huex.1.0.st.v2
```

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.

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

```
## Warning:  'isIdCurrent' is deprecated.
## Use 'dbIsValid' instead.
## See help("Deprecated")
```

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*.

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

```
## Warning:  'isIdCurrent' is deprecated.
## Use 'dbIsValid' instead.
## See help("Deprecated")
```

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).

```
library(RCurl)
```

```
## Loading required package:  bitops
```

```
options(RCurlOptions=list(http.version=HTTP_VERSION_1_0))
library(biomaRt)
ensembl <- useMart("ENSEMBL_MART_ENSEMBL", dataset="hsapiens_gene_ensembl",
                   host='may2009.archive.ensembl.org')
theIDs <- getBM(attributes="affy_huex_1_0_st_v2", filters="entrezgene",
                values=10948, mart=ensembl)
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.

```
library(AnnotationDbi)
```

```
## Loading required package:  GenomeInfoDb
```

```
conn <- db(affyExonFS)
```

```
## Warning:  'isIdCurrent' is deprecated.
## Use 'dbIsValid' instead.
## See help("Deprecated")
```

```
fields <- 'meta_fsetid, pmfeature.fsetid, fid, start, stop'
tables <- 'featureSet, pmfeature, core_mps'
sql <- paste("SELECT", fields,
             "FROM", tables,
             "WHERE pmfeature.fsetid=featureSet.fsetid",
             "AND featureSet.fsetid=core_mps.fsetid",
             "AND pmfeature.fsetid=:psets")
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.

```
library(Biostrings)
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.

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

```
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.

```
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.

```
library("BSgenome.Hsapiens.UCSC.hg18")
```

```
## Loading required package:  BSgenome
## Loading required package:  GenomicRanges
## Loading required package:  rtracklayer
```

```
chr17 <- Hsapiens[["chr17"]]
seqs <- complement(DNAStringSet(probeInfo[["sequence"]]))
seqs <- PDict(seqs)
matches <- matchPDict(seqs, chr17)
```

After matching the sequences, we update the genomic coordinates.

```
probeInfo[["start"]] <- unlist(startIndex(matches))
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.

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

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

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

```
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.

```
library(GenomeGraphs)
```

```
## Loading required package:  grid
```

```
probeids <- as.character(probeInfo[["fsetid"]])
ensembl <- useMart("ENSEMBL_MART_ENSEMBL", dataset="hsapiens_gene_ensembl", host='may2009.a
## ensembl = useMart("ensembl", dataset="hsapiens_gene_ensembl")
geneid <- "ENSG00000131748"
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.

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

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

```
attach(psInfo)
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.

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

```
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.

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

Figure 6.7, 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.

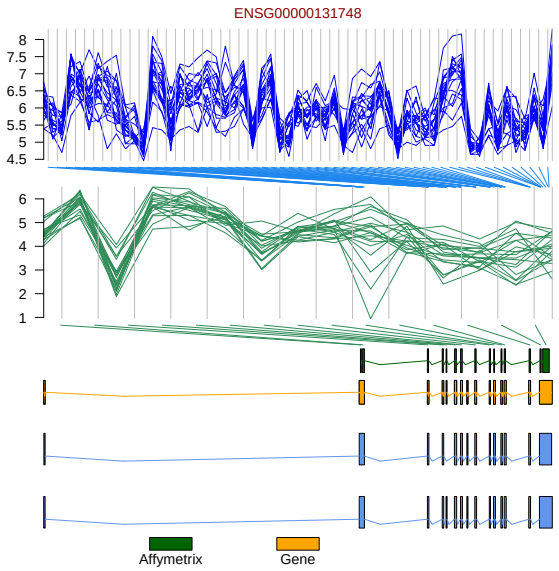


Figure 6.7: Visual representation of observed $\log_2$-intensities and summarized data at the exon level for gene ENSG00000131748. The probes, gene and transcript are also represented, respectively, in green, orange and blue.

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`.

```
mps <- unique(probeInfo[["meta_fsetid"]])
mps <- as.character(mps)
mps
```

```
## [1] "3720343"
```

Therefore, the standard accessors can be used to obtain the gene summaries for the unit above. Figure 6.8 shows the expressions for gene ENSG00000131748 across the 33 samples available on this dataset.

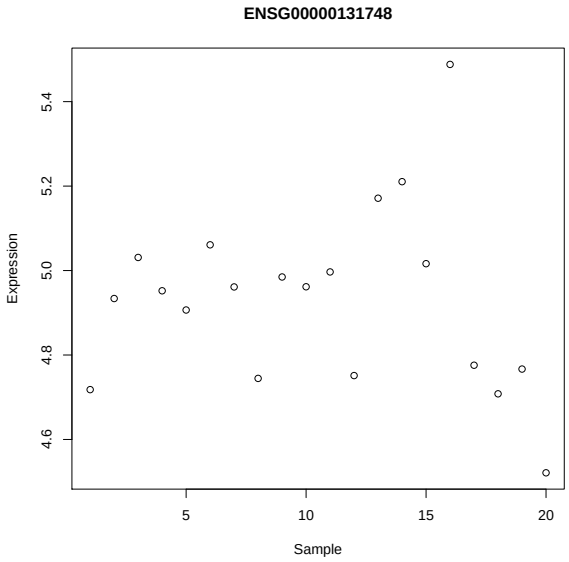


Figure 6.8: Expression levels estimated through RMA at the gene level.

6.5 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.

```
library(oligo)
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.

```
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:

```
library(oligoData)
data(nimbleTilingFS)
nimbleTilingFS
```

```
## Loading required package:  pd.2006.07.18.hg18.refseq.promoter
```

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.

```
c1 <- channel(nimbleTilingFS, "channel1")
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.

```
fields <- 'fid, fsetid, chrom as chromosome, position as start'
sql <- paste("SELECT", fields,
             "FROM pmfeature INNER JOIN featureSet USING(fsetid)",
             "ORDER BY chrom, position")
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.

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

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

```
attach(annotPM)
library(Biostrings)
annotPM[["end"]] <- start+width(pmSequence[["sequence"]])-1
head(annotPM)
```

```
##      fid fsetid chromosome start   end
## 1 392369   5622       chr1 56753 56808
## 2 286872   5622       chr1 56853 56909
## 3 229027   5622       chr1 56953 57007
## 4 386658   5622       chr1 57053 57114
## 5  85534   5622       chr1 57153 57202
## 6 170025   5622       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.

```
ratioPM <- getM(nimbleTilingFS)[fid,]
dimnames(ratioPM) <- NULL
detach(annotPM)
class(ratioPM)
```

```
## [1] "matrix"
```

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

```
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.

```
library(ACME)
```

```
##
## Attaching package:  'ACME'
##
## The following object is masked from 'package:oligoClasses':
##
##     chromosome
```

```
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.

```
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.

```
regs <- findRegions(calc)
head(regs)
```

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

6.6 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.

6.6.1 Support to large datasets

The *oligo* package 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()`.

```
library(oligo)
library(ff)
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.

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

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

6.6.2 Parallel computing

The *oligo* package 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:

```
library(ff)
library(foreach)
library(doMC)
registerDoMC(2)
library(oligo)
```

```
rawData <- read.celfiles(list.celfiles())
rmaRes <- rma(rawData)
rmaRes
```

6.6.3 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).

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

```
## Warning:  'isIdCurrent' is deprecated.
## Use 'dbIsValid' instead.
## See help("Deprecated")
```

```
## Background correcting
## Normalizing
## Calculating Expression
```

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

```
## Warning:  'isIdCurrent' is deprecated.
## Use 'dbIsValid' instead.
## See help("Deprecated")
```

```
## Background correcting
## Normalizing
## Calculating Expression
```

```
all.equal(res0, res1)
```

```
## [1] TRUE
```

```
t0
```

```
##    user  system elapsed
##  11.137   0.571  12.680
```

```
t1
```

```
##    user  system elapsed
##  16.746   0.898   6.464
```

6.7 Session Info

```
## \begin{itemize}\raggedright
##   \item R version 3.2.0 beta (2015-04-05 r68152), \verb|x86_64-apple-darwin10.8.0|
##   \item Locale: \verb|en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8|
##   \item Base packages: base, datasets, graphics, grDevices, grid, methods,
##     parallel, stats, stats4, utils
##   \item Other packages: ACME~2.23.0, AnnotationDbi~1.29.23, Biobase~2.27.3,
##     BiocGenerics~0.13.11, biomaRt~2.23.5, Biostrings~2.35.12, bitops~1.0-6,
##     BSgenome~1.35.20, BSgenome.Hsapiens.UCSC.hg18~1.3.1000, DBI~0.3.1,
##     genefilter~1.49.2, GenomeGraphs~1.27.0, GenomeInfoDb~1.3.18,
##     GenomicRanges~1.19.52, IRanges~2.1.44, knitr~1.9, limma~3.23.13,
##     maqcExpression4plex~1.11.1, oligo~1.31.7, oligoClasses~1.29.6,
##     oligoData~1.8.0, pd.2006.07.18.hg18.refseq.promoter~1.8.1,
##     pd.hg.u95av2~3.10.0, pd.hg18.60mer.expr~3.4.1, pd.huex.1.0.st.v2~3.10.0,
##     pd.mapping50k.xba240~3.10.0, RCurl~1.95-4.5, RSQLite~1.0.0,
##     rtracklayer~1.27.12, S4Vectors~0.5.23, XVector~0.7.4
##   \item Loaded via a namespace (and not attached): affxparser~1.39.4,
##     affyio~1.35.0, annotate~1.45.4, BiocInstaller~1.17.7, BiocParallel~1.1.27,
##     BiocStyle~1.5.4, bit~1.1-12, codetools~0.2-11, digest~0.6.8, evaluate~0.5.5,
##     ff~2.2-13, foreach~1.4.2, formatR~1.1, futile.logger~1.4,
##     futile.options~1.0.0, GenomicAlignments~1.3.34, highr~0.4.1,
##     iterators~1.0.7, KernSmooth~2.23-14, lambda.r~1.1.7, preprocessCore~1.29.0,
##     Rsamtools~1.19.52, splines~3.2.0, stringr~0.6.2, survival~2.38-1,
##     tools~3.2.0, XML~3.98-1.1, xtable~1.7-4, zlibbioc~1.13.3
## \end{itemize}
```