Introduction to the xps Package: Overview

Christian Stratowa

September, 2010

Contents

1 Introduction

Affymetrix GeneChip oligonucleotide arrays use several probes to assay each targeted transcript. Specialized algorithms have been developed to summarize low-level probe set intensities to get one expression measure for each transcript. Some of these methods, such as MAS 4.0's AvDiff (Affymetrix, 1999), MAS5's signal (Affymetrix, 2001) or RMA (Irizarry et al., 2003), are implemented in package affy (Gautier et al., 2004). Further methods, such as FARMS (Hochreiter et al., 2006) or DFW (Chen et al., 2007) are custom methods that can be registered for use with package affy.

Advantages in technology allow Affymetrix to supply whole-genome expression arrays such as the new GeneChip Exon array systems (Exon 1.0 ST) and Gene array systems (Gene 1.0 ST). The amount of data created with the new exon arrays poses a great challenge, since R stores all objects in memory.

Package xps - *eXpression Profiling System* - is designed to analyze Affymetrix GeneChip expression and exon arrays on computers with limited amounts of memory (1 GB RAM). To achieve this goal,

xps takes advantage of `ROOT`, a framework especially developed to handle and analyse large amounts of data in a memory efficient way.

Important installation note: Package xps is based on two powerful frameworks, namely `R` and `ROOT`. It is thus absolutely essential to install the `ROOT` framework before xps can be built and installed. For instructions how to install `ROOT` see the README file provided with package xps.

2 Why ROOT?

ROOT (`http://root.cern.ch`) is an object-oriented framework that has been developed at CERN for distributed data warehousing and data mining of particle data in the petabyte range, such as the data created with the new LHC collider. Data are stored as sets of objects in machine-independent files, and specialized storage methods are used to get direct access to separate attributes of selected data objects. For more information see the ROOT User Guide (The ROOT team (2009)).

Taking advantage of these features, package xps stores all data in portable `ROOT` files. Data describing microarray layout, probe information and metadata for genes are stored as `ROOT` Trees in *scheme* files, accessible from `R` as *scheme* objects. Raw probe intensities, i.e. CEL-files for each project are stored as `ROOT` Trees in *data* files, accessible from `R` as *data* objects. All analysis is done independent of `R` such avoiding inherent memory limitations.

Note: Absolutely no knowledge of `ROOT` is required to use package xps. However, the interested user could use package xps independent of `R` by writing `ROOT` macros, examples of which can be found in file "macro4XPS.C", located in subdirectory examples.

3 Getting Started

First you need to load the xps package.

```
R> library(xps)
```

```
Welcome to xps version 1.10.2
    an R wrapper for XPS - eXpression Profiling System
    (c) Copyright 2001-2010 by Christian Stratowa
```

As an initial step, which needs to be done only once, you must import Affymetrix chip definition files, probe files and annotation files for all arrays that you are using, into `ROOT` *scheme* files. This is described in Appendix A1, here we use the `ROOT` *scheme* file supplied with the package.

Throughout this tutorial we will use a set of four *CEL* files supplied with the package. The necessary `ROOT` *scheme* file *SchemeTest3.root* for GeneChip *Test3.CDF* is also supplied as well as the `ROOT` *data* file *DataTest3_cel.root*. These files need to be loaded for every new `R`-session, unless the session has been saved.

Note: Please see Appendix A2 for many additional examples on how to use xps.

3.1 Reading CEL file information

The *CEL* files can be located in a common directory or in different directories, see `?import.data` how to import *CEL* files from different directories. *CEL* files will be imported into a `ROOT` *data* file as `ROOT` *Trees*. Once the `ROOT` *data* file is created, the *CEL* files are no longer needed, since subsequent R-sessions need only load the `ROOT` *data* file. However, it is possible to load only a subset of *CEL* files, and it is also possible to save new *CEL* files in the same `ROOT` *data* file at a later time. In this demo we will show how to achieve this.

First we load the xps package.

```
> library(xps)
```

For this demonstration *CEL* files are located in a common directory, in our case in:

```
> celdir <- file.path(.path.package("xps"), "raw")
```

Since our *CEL* files were created for GeneChip *Test3.CDF*, we need to load the corresponding `ROOT` *scheme* file first:

```
> scheme.test3 <- root.scheme(file.path(.path.package("xps"), "schemes",
+     "SchemeTest3.root"))
```

Now we can import the *CEL* files, in our case a subset first:

```
> celfiles <- c("TestA1.CEL", "TestA2.CEL")
> data.test3 <- import.data(scheme.test3, "tmpdt_DataTest3", celdir = celdir,
+     celfiles = celfiles, verbose = FALSE)
```

To see, which *CEL* files were imported as `ROOT` *Trees*, we can do:

```
> unlist(treeNames(data.test3))
```

```
[1] "TestA1.cel" "TestA2.cel"
```

Now we can import additional *CEL* files:

```
> celfiles <- c("TestB1.CEL", "TestB2.CEL")
> data.test3 <- addData(data.test3, celdir = celdir, celfiles = celfiles,
+     verbose = FALSE)
```

Instead of getting the imported tree names from the created instance *data.test3* of S4 class *DataTreeSet*, we can also get the tree names directly from the `ROOT` *data* file:

```
> getTreeNames(rootFile(data.test3))
```

```
[1] "TestA1.cel" "TestA2.cel" "TestB1.cel" "TestB2.cel"
```

Now we have all *CEL* files imported as `ROOT` *Trees*. In later R-sessions we only need to load the corresponding `ROOT` *data* file using function `root.data`. In this tutorial we will not use the file just created but the `ROOT` *data* file *DataTest3_cel.root*.

Note 1: It is also possible to import 'phenotypic-data' describing samples and further project–relevant data for the experiment, see S4 class *ProjectInfo*.

Note 2: Since `ROOT` *data* files contain the raw data, it is recommended to create them in a common system directory, e.g. 'rootdata', which is accessible to other users, too.

Note 3: In order to distinguish `ROOT` *data* files containing the raw data from other `ROOT` files, extension '_cel' is automatically added to the file name. Thus creating a raw data file with name *DataTest3* will result in a `ROOT` file with name *DataTest3_cel.root*. Extension 'root' is always added to each `ROOT` file.

Note 4: Usually, `ROOT` *data* files are kept permanently. Thus it is not possible to accidently overwrite a `ROOT` *data* file with another file of the same name; you will get an error message. If you want to create a temporary `ROOT` *data* file, which can be overwritten, the name must start with 'tmp_'. However, in the example above we needed to use 'tmpdt_' otherwise R CMD check would produce an error on Windows. Please note that 'tmpdt_' will not work with function `import.data` for the reason described in Note 3 above.

Note 5: It is highly recommended to keep the default setting `verbose=TRUE`, especially when working with exon arrays. On Windows you will see the verbose messages only when starting `R` from the command line, i.e. using RTerm.

3.2 Accessing raw data

Currently, the data from the imported *CEL* files are saved as `ROOT` *Trees* in the `ROOT` *data* file, however, they are not accessible from within `R`. The corresponding slot `data` of instance *data.test3* of S4 class *DataTreeSet*, a data.frame, is empty. This setting allows to import e.g. an (almost) unlimited number of *CEL* files from GeneChip Exon arrays on computers with 1GB RAM only.

When we try to access the raw data, we get:

```
> tmp <- intensity(data.test3)
> head(tmp)

data frame with 0 columns and 0 rows
```

Thus, we need to attach the raw data first to *data.test3*:

```
> data.test3 <- attachInten(data.test3)
```

Now we get:

```
> tmp <- intensity(data.test3)
> head(tmp)

  X Y TestA1.cel_MEAN TestA2.cel_MEAN TestB1.cel_MEAN TestB2.cel_MEAN
1 0 0          1319.1          1343.7           765.0           653.9
2 1 0         21304.9         21281.2          9742.5         18531.1
3 2 0          1009.9          1084.7          1162.6           466.8
4 3 0         21204.7         21233.9          6334.8         18896.0
5 4 0           960.7          1010.7           164.2           990.1
6 5 0          1078.0          1103.7           380.6           770.4
```

Alternatively, it is also possible to attach only a subset to the current object *data.test3*, or to a copy *subdata.test3*:

```
> subdata.test3 <- attachInten(data.test3, c("TestB1.cel", "TestA2"))
> tmp <- intensity(subdata.test3)
> head(tmp)

  X Y TestB1.cel_MEAN TestA2.cel_MEAN
1 0 0           765.0          1343.7
2 1 0          9742.5         21281.2
3 2 0          1162.6          1084.7
4 3 0          6334.8         21233.9
5 4 0           164.2          1010.7
6 5 0           380.6          1103.7
```

When we no longer need the raw data, we can remove them from *data.test3*, thus avoiding memory consumption of `R`:

```
> data.test3 <- removeInten(data.test3)
> tmp <- intensity(data.test3)
> head(tmp)

data frame with 0 columns and 0 rows
```

Now we want create some plots with the raw data. For this purpose we need not only attach the raw data, as shown above, but also slot `mask` of *scheme.test3*, since slot `mask` contains the information which oligos on the array are PM, MM, or control oligos, respectivly. See Appendix A1 for an explanation and how to avoid this step.

```
> data.test3 <- attachMask(data.test3)
> data.test3 <- attachInten(data.test3)
```

Note: We have applied method `attachMask` to *data.test3* and not to *scheme.test3*, since *data.test3* contains its own copy of *scheme.test3*.
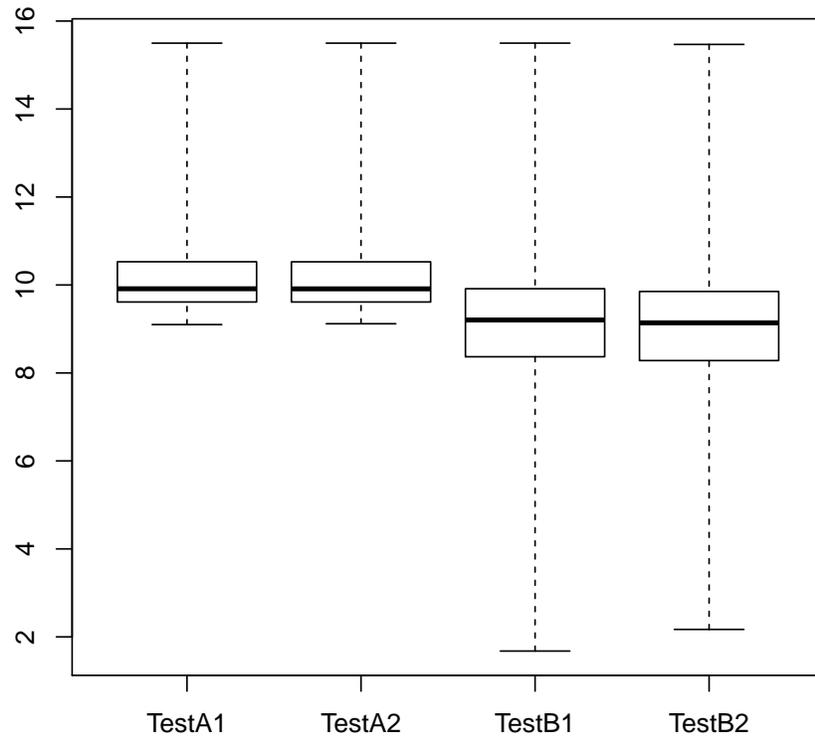
First, we create a density plot:

```
> hist(data.test3)
```
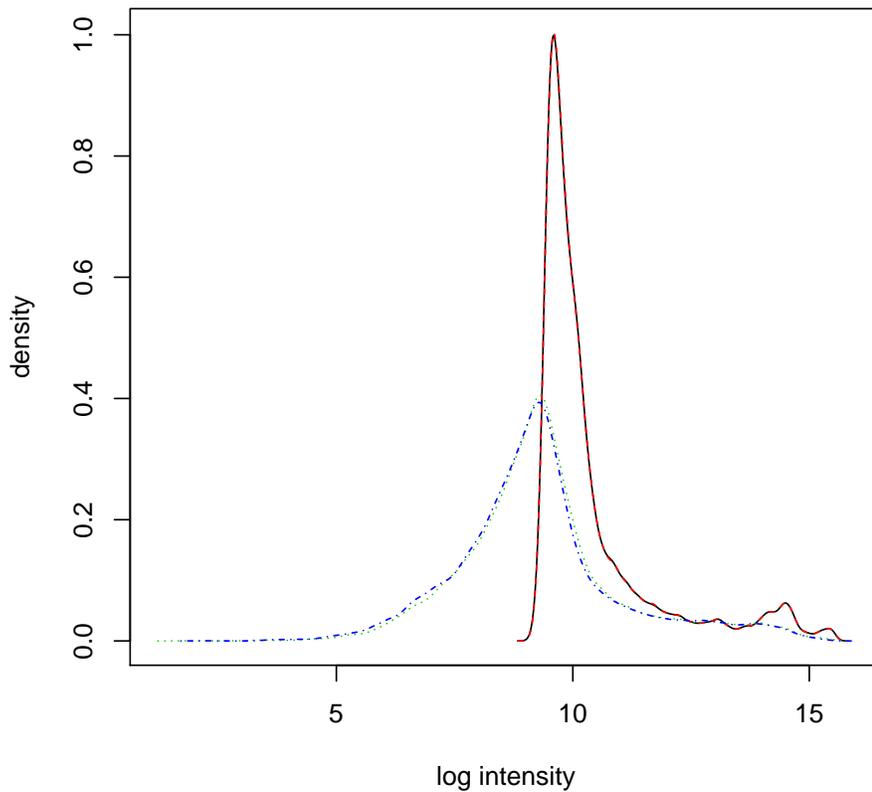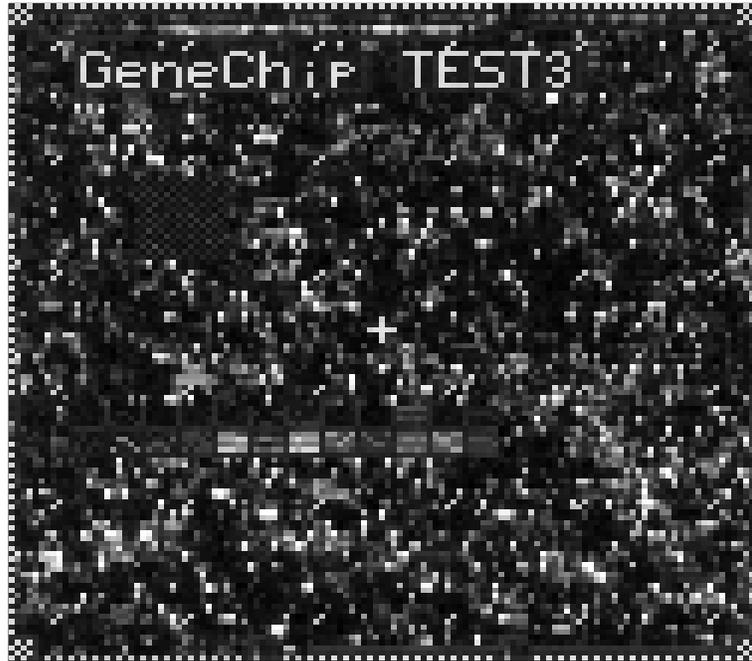
The corresponding boxplots are:

```
> boxplot(data.test3)
```

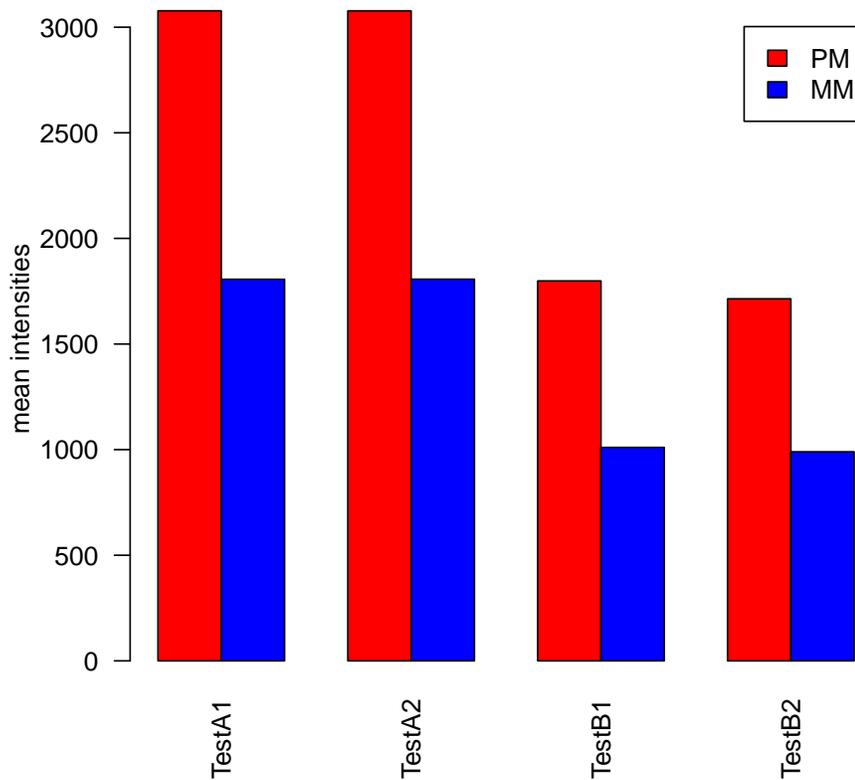It is also possible to create an image for e.g. sample TestA1:

```
> image(data.test3, names = "TestA1.cel_MEAN")
```

TestA1.cel_MEAN

Finally we include a PM-MM-plot of the data:

```
> pmplot(data.test3)
```


After we are done, we remove the data from *data.test3* to free `R` memory:

```
> data.test3 <- removeInten(data.test3)
> data.test3 <- removeMask(data.test3)
```

This step is not necessary for small datasets or if the computer has sufficient RAM.

Note: From the description above it is evident, that it is not possible to plot exon array data using e.g. functions `hist` or `image`, respectively, on computers with 1GB RAM only. To allow plotting under low memory conditions, package `xps` supports `ROOT` graphics, as described in Appendix A4.

4   Converting raw data to expression measures

When we start a new R-session, it is necessary to load the `ROOT` *scheme* and `ROOT` *data* files first:

```
> library(xps)
> scheme.test3 <- root.scheme(file.path(.path.package("xps"), "schemes",
```

```
+       "SchemeTest3.root"))
> data.test3 <- root.data(scheme.test3, file.path(.path.package("xps"),
+       "rootdata", "DataTest3_cel.root"))
```

This step is not necessary when objects *scheme.test3* and *data.test3* are already saved in an R-session.

Converting raw data to expression measures and computing detection calls is fairly simple. It is not necessary to attach any `data` or `mask` data.frames, since all computations are done independently from R.

4.1 Calculating expression levels

Let us first preprocess the raw data using method 'RMA' to compute expression levels, and store the results as ROOT *Trees* in ROOT file *tmpdt_Test3RMA.root*:

```
> data.rma <- rma(data.test3, "tmpdt_Test3RMA", verbose = FALSE)
```

Note: In this example and the following examples we suppress the usual output. Furthermore, once again we use 'tmpdt_', which adds date and time to the tmp-file, otherwise R CMD check would produce an error on Windows. Usually, you want to create a permanent file, however, if you want to create a temporary file it is recommended to use 'tmp_' as temporary file which will be overwritten.

Then we preprocess the raw data using method 'MAS5' to compute expression levels, and store the results in ROOT file *tmpdt_Test3MAS5.root*:

```
> data.mas5 <- mas5(data.test3, "tmpdt_Test3MAS5", normalize = TRUE,
+       sc = 500, update = TRUE, verbose = FALSE)
```

Now we want to compare the results by plotting the expression levels for the first sample. For this purpose we need to extract the expression levels from the resulting S4 classes *ExprTreeSet* as data.frames first:

```
> expr.rma <- validData(data.rma)
> expr.mas5 <- validData(data.mas5)
```

Now we can plot the results for the first sample:

```
> plot(expr.rma[, 1], expr.mas5[, 1], log = "xy", xlim = c(1, 20000),
+       ylim = c(1, 20000))
```
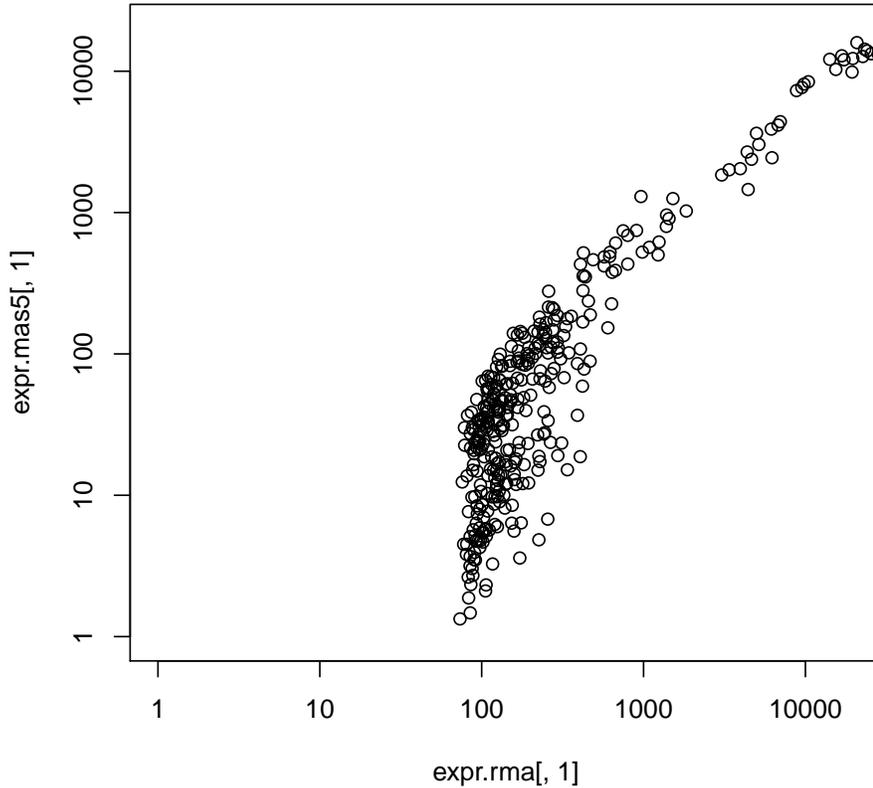
Note: For both methods, 'RMA' and 'MAS5', true expression levels are extracted, which is in contrast to other packages which extract the log2 values for 'RMA'.

4.2 Calculating detection calls

Let us now compute the MAS5 detection calls:

```
> call.mas5 <- mas5.call(data.test3, "tmpdt_Test3Call", verbose = FALSE)
```

Alternatively, let us compute the DABG (detection above background) calls:

```
> call.dabg <- dabg.call(data.test3, "tmpdt_Test3DABG", verbose = FALSE)
```

Note: YES, in principle it is indeed possible to compute the DABG call not only for exon arrays but for expression arrays, too. However, computation may take a long time, e.g. on a computer with 2.3GHz Intel Core 2 Duo processor and 2GB RAM, computing DABG calls for HG-U133_Plus_2 arrays will take about 45 min/array.

Both, detection call and detection p-value can be extracted as `data.frame`:

```
> pres.mas5 <- presCall(call.mas5)
> head(pres.mas5)
```

```
  UNIT_ID        UnitName TestA1.dc5_CALL TestA2.dc5_CALL TestB1.dc5_CALL
1       0 Pae_16SrRNA_s_at               A               A               A
2       1 Pae_23SrRNA_s_at               A               A               P
3       2    PA1178_oprH_at               A               A               A
4       3    PA1816_dnaQ_at               A               A               A
5       4     PA3183_zwf_at               A               A               A
6       5    PA3640_dnaE_at               A               A               A
  TestB2.dc5_CALL
1               A
2               A
3               A
4               A
5               A
6               A

> pval.mas5 <- pvalData(call.mas5)
> head(pval.mas5)

  UNIT_ID        UnitName TestA1.dc5_PVALUE TestA2.dc5_PVALUE
1       0 Pae_16SrRNA_s_at          0.837065          0.660442
2       1 Pae_23SrRNA_s_at          0.458816          0.418069
3       2    PA1178_oprH_at          0.975070          0.979305
4       3    PA1816_dnaQ_at          0.880342          0.805907
5       4     PA3183_zwf_at          0.863952          0.863952
6       5    PA3640_dnaE_at          0.950260          0.979305
  TestB1.dc5_PVALUE TestB2.dc5_PVALUE
1        0.56163900         0.872355
2        0.00564281         0.749276
3        0.62315800         0.291460
4        0.70854000         0.997629
5        0.78361600         0.975070
6        0.84608900         0.979305
```

4.3   Plotting results

Now we want to create some plots for the expression levels. In contrast to the raw data we do not
need to attach data, since expression levels and detection calls are already stored in the R objects, e.g.
in *data.rma*, an instance of S4 class *ExprTreeSet*, or in *call.mas5*, an instance of S4 class *CallTreeSet*,
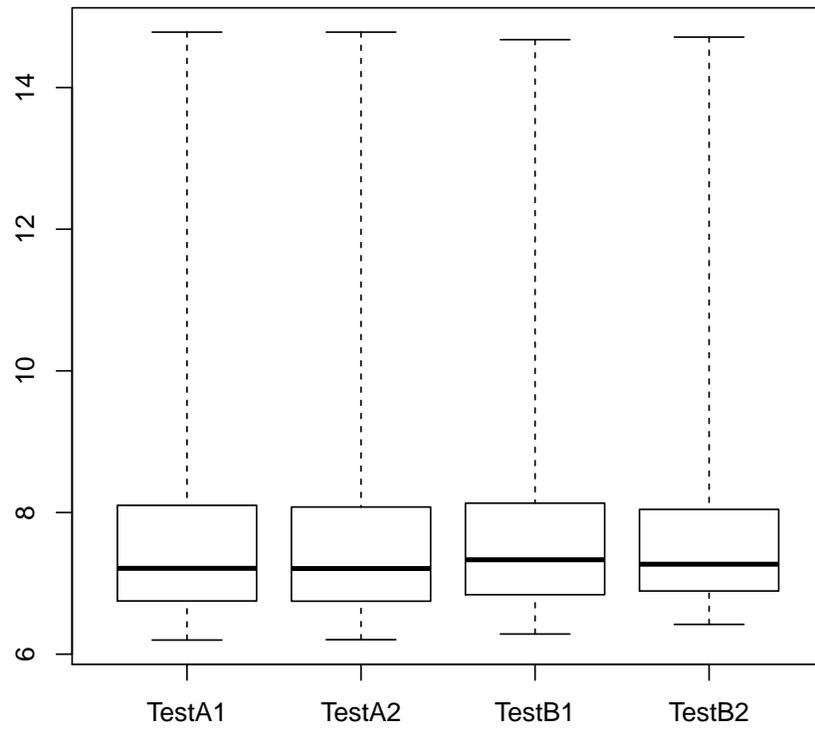respectively.

First, we create a density plot:
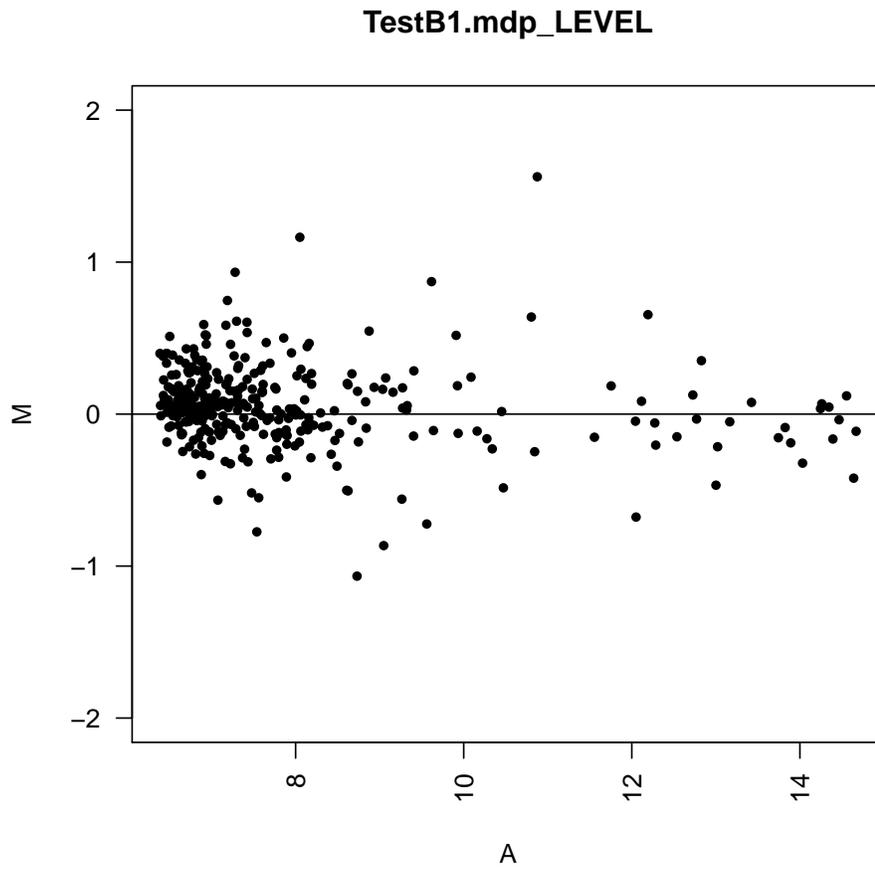
```
> hist(data.rma)
```
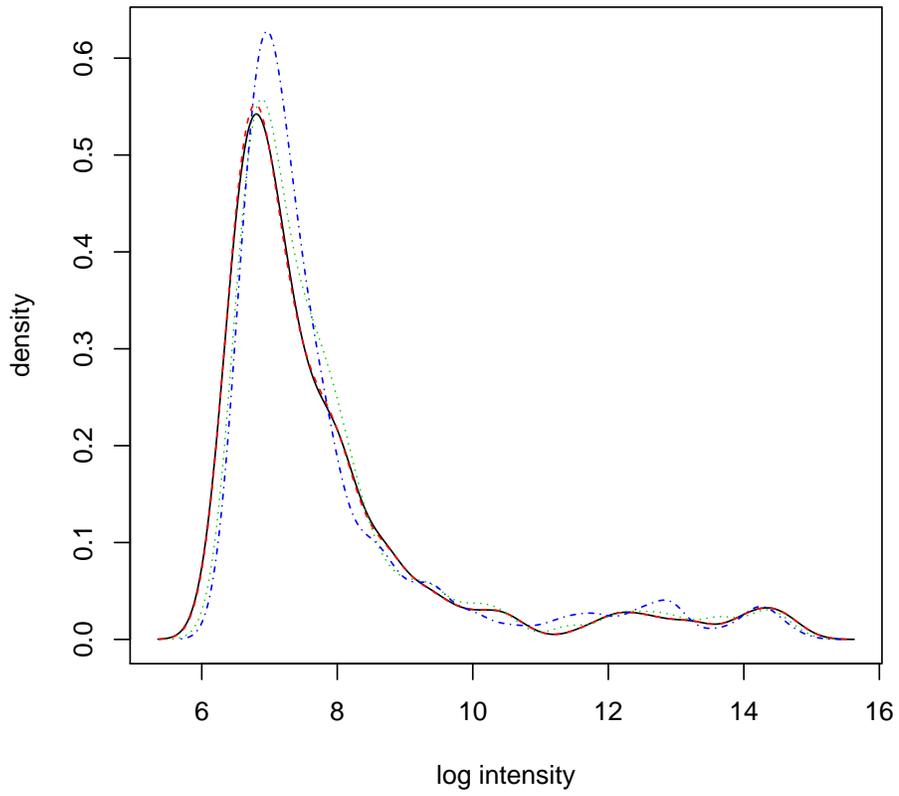
The corresponding boxplots are:
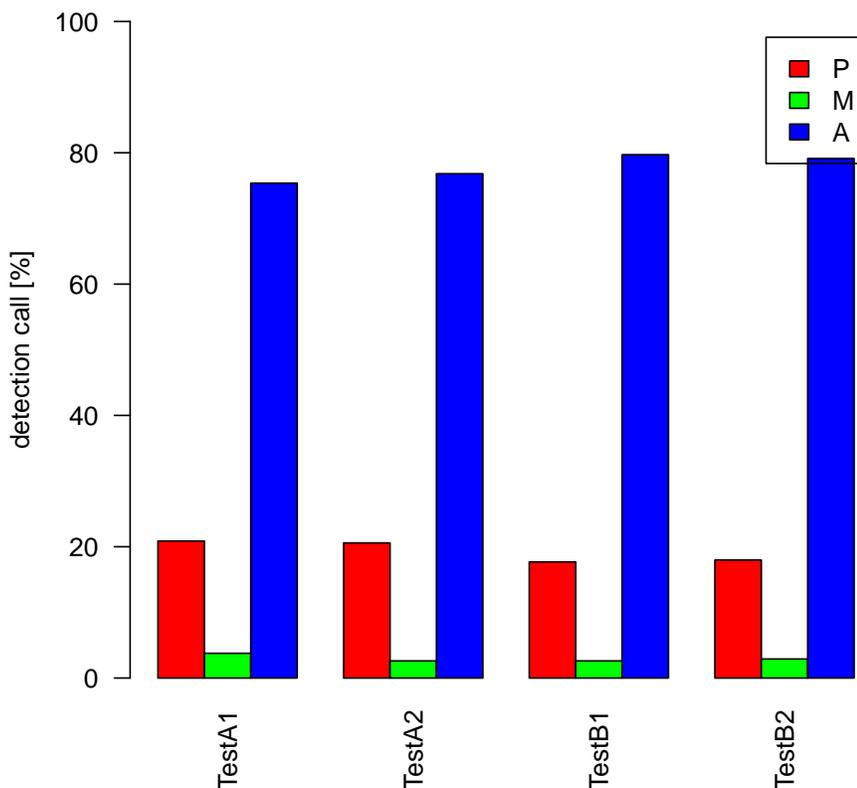
```
> boxplot(data.rma)
```

It is also possible to create M vs A plots for one or more samples:

```
> mvaplot(data.rma, pch = 20, ylim = c(-2, 2), names = "TestB1.mdp_LEVEL")
```

TestB1.mdp_LEVEL

Finally we create present call plots:

```
> callplot(call.mas5)
```


Since the `ROOT` expression files contain additional `ROOT` *Trees*, e.g. trees containing the computed background intensities, let us create an `image` showing the distribution of the background intensities on an array for MAS5.

First, we need to know the tree names stored in *data.mas5*

```
> getTreeNames(rootFile(data.mas5))
```

```
 [1] "TestA1.wbg" "TestA1.int" "TestA2.wbg" "TestA2.int" "TestB1.wbg"
 [6] "TestB1.int" "TestB2.wbg" "TestB2.int" "TestA1.tbw" "TestA2.tbw"
[11] "TestB1.tbw" "TestB2.tbw" "TestA1.rnk" "TestA1.tmn" "TestA2.rnk"
[16] "TestA2.tmn" "TestB1.rnk" "TestB1.tmn" "TestB2.rnk" "TestB2.tmn"
```

We decide to use tree 'TestA2.wbg' (extension 'wbg' stands for 'weighted background', see `?validTreetype`).

In this case we need to export the background intensities directly from `ROOT` file *tmp_Test3MAS5.root*. We can do this in the following way:
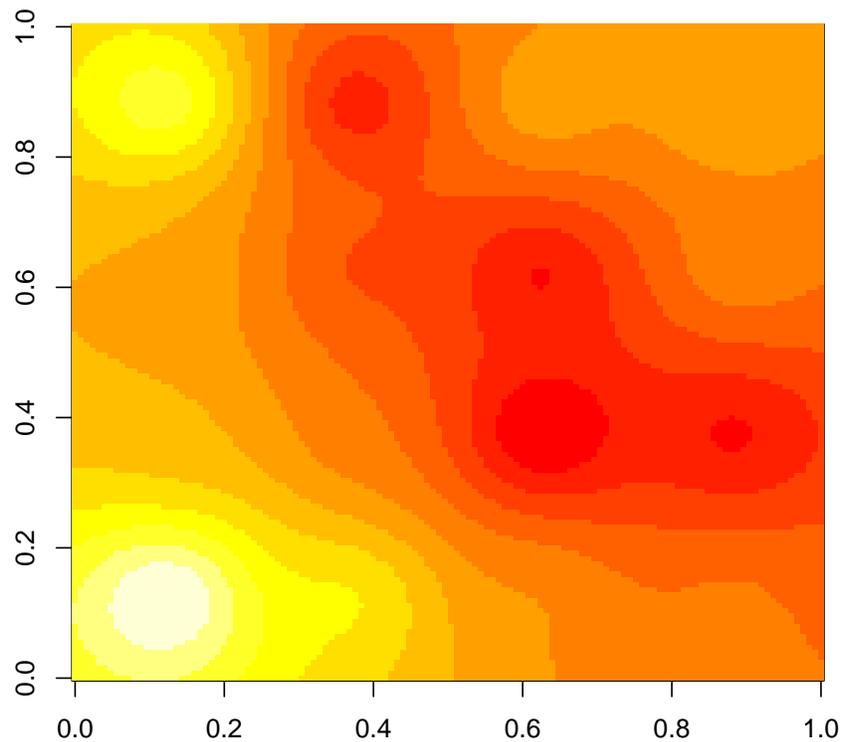
```
> bgrd <- export.root(rootFile(data.mas5), schemeFile(data.mas5),
+     "PreprocesSet", "TestA2", "wbg", "fBg", "BgrdROOTOut.txt",
+     as.dataframe = TRUE, verbose = FALSE)
```

Then we need to convert background intensities to an object of type `matrix`:

```
> wbg <- matrix(bgrd[, "BGRD"], ncol = ncols(schemeSet(data.mas5)),
+     nrow = nrows(schemeSet(data.mas5)))
```

Finally, we can draw the image:

```
> image(log2(wbg))
```


5 Filtering expression measures

The xps package can also be used to filter (select) genes according to a variety of different filtering mechanisms, similar to Bioconductor package genefilter.

It is important to note that filters can be split into the non–specific filters and the specific filters. Usually, non–specific filters are used to reduce the number of genes remaining for further analysis e.g. by reducing the noise in the dataset. In contrast, specific means that we are filtering with reference to a particular covariate. For example we want to select genes that are differentially expressed in two groups. Here we use the term 'prefilter' for non–specific filters and the term 'unifilter' for specific filters applied to two groups.

5.1 Applying non–specific filters: PreFilter

Applying non–specific filters is a simple two-step process: First, select the filters of interest using constructor `PreFilter`. Second, apply the resulting class *PreFilter* to an instance of class *ExprTreeSet* using function `prefilter`.

Currently it is possible to select up to ten non–specific filters which are defined in S4 class *PreFilter*. For this example let us initialize the following three non–specific filters:

1. `madFilter`: A 'median absolute deviation' filter, which selects only genes where `mad` across all samples is at least 0.5, i.e. `mad >= 0.5`.

2. `lowFilter`: A 'lower threshold' filter to select genes where the trimmed mean of the `log2`– expression levels is above 7.0 (with `trim = 0.02`).

3. `highFilter`: An 'upper threshold' filter to select genes that are `log2`–expressed below 10.5 in at least 80 percent of the samples.

Furthermore, a gene should be selected for further analysis only if it satisfies at least two of the three filters.

Initialization of the filters is done using the constructor `PreFilter`:

```
> prefltr <- PreFilter(mad = c(0.5), lothreshold = c(7, 0.02, "mean"),
+     hithreshold = c(10.5, 80, "percent"))
> str(prefltr)
```

This filter is then applied to expression data *data.rma* created earlier, using function `prefilter` with parameter `minfilters=2`:

```
> rma.pfr <- prefilter(data.rma, "tmpdt_Test3Prefilter", getwd(),
+     filter = prefltr, minfilters = 2, verbose = FALSE)
```

The resulting filter mask can be extracted as `data.frame`:

```
> tmp <- validData(rma.pfr)
> head(tmp)

                UNIT_ID FLAG
Pae_16SrRNA_s_at      0    1
Pae_23SrRNA_s_at      1    0
PA1178_oprH_at        2    0
PA1816_dnaQ_at        3    1
PA3183_zwf_at         4    1
PA3640_dnaE_at        5    0

> dim(tmp[tmp[, "FLAG"] == 1, ])

[1] 181   2
```

The data show that 181 genes of the 345 genes on the Test3 GeneChip are selected for further analysis.

5.2 Applying specific filters for two groups: UniFilter

Applying univariate filters is also a simple two-step process: First, select the filters of interest using constructor `UniFilter`. Second, apply the resulting class *UniFilter* to an instance of class *ExprTreeSet* using function `unifilter`.

Currently it is possible to select three univariate filters which are defined in S4 class *UniFilter*. For this example let us initialize the following two filters:

1. `fcFilter`: A 'fold–change' filter, which selects only genes with an absolute fold–change of at least 1.3, i.e. `abs(mean(GrpB)/mean(GrpA)) >= 1.3`.

2. `unitestFilter`: A 'unitest' filter to select genes where the p–value of the applied unitest, i.e. the `t.test` is less than 0.1 (`pval <= 0.1`).

Only genes satisfying both filters are considered to be differentially expressed.

Note: If you want to change the default settings for `t.test` and/or compute an adjusted p–value for multiple comparisons you need to initialize method `uniTest`, too.

Initialization of the filters is done using the constructor `UniFilter`:

```
> unifltr <- UniFilter(foldchange = c(1.3, "both"), unifilter = c(0.1,
+     "pval"))
```

This filter is then applied to expression data *data.rma* using function `unifilter` where parameter `group` allocates each sample to one of two groups. Furthermore, since we want to use only the pre–selected genes from `prefilter` we need to set `xps.fltr=rma.pfr`:

```
> rma.ufr <- unifilter(data.rma, "tmpdt_Test3Unifilter", getwd(),
+     unifltr, group = c("GrpA", "GrpA", "GrpB", "GrpB"), xps.fltr = rma.pfr,
+     verbose = FALSE)
```

The resulting data can be extracted as `data.frame`:

```
> tmp <- validData(rma.ufr)
> tmp
```

	UNIT_ID	Statistics	Mean1	Mean2	StandardError
AFFX-Ce_Gapdh_5_s_at	40	7.06687	298.668	209.920	0.0719846
rrlG_b2589_s_at	186	-10.74160	122.945	169.814	0.0433769
37189_at	214	-8.24096	241.766	369.666	0.0743373
AFFX-18SRNAMur/X00686_3_at	243	-7.66081	452.422	666.802	0.0730458
AFFX-hum_alu_at	277	-63.55890	674.325	5368.190	0.0470889
AFFX-HUMISGF3A/M97935_MA_at	283	-15.41420	341.779	458.275	0.0274521
AFFX-HUMRGE/M10098_5_at	286	-26.88740	149.584	199.014	0.0153200
AFFX-HUMRGE/M10098_M_at	287	-17.77530	125.175	176.651	0.0279579
AFFX-MurFAS_at	298	-7.14193	163.431	226.340	0.0657826

	DegreeOfFreedom	P-Value	P-Adjusted	FoldChange
AFFX-Ce_Gapdh_5_s_at	1.19543	0.06399260	0.06399260	0.702853
rrlG_b2589_s_at	1.49189	0.02160840	0.02160840	1.381220
37189_at	1.00864	0.07560790	0.07560790	1.529020
AFFX-18SRNAMur/X00686_3_at	1.05823	0.07425470	0.07425470	1.473850
AFFX-hum_alu_at	1.06781	0.00766898	0.00766898	7.960830
AFFX-HUMISGF3A/M97935_MA_at	1.61801	0.00952405	0.00952405	1.340850
AFFX-HUMRGE/M10098_5_at	1.00504	0.02329990	0.02329990	1.330450
AFFX-HUMRGE/M10098_M_at	1.04887	0.03139930	0.03139930	1.411240
AFFX-MurFAS_at	1.96059	0.02008720	0.02008720	1.384930

The data show that only 9 genes of the pre–selected 181 genes are considered to be differentially expressed.

Note: If you want to extract all data as `data.frame` as well as the resulting filter mask you can do:

```
> msk <- validFilter(rma.ufr)
> tmp <- validData(rma.ufr, which = "UnitName")
> tmp <- cbind(tmp, msk)
```

However, the recommended way to extract all data together with the filter mask as well as the gene annotation is:

```
> tmp <- export.filter(rma.ufr, treetype = "stt", varlist = "fUnitName:fName:fSymbol:fc:pval:flag",
+     as.dataframe = TRUE, verbose = FALSE)
> head(tmp)
```

```
  UNIT_ID          UnitName                              GeneName GeneSymbol
1       0 Pae_16SrRNA_s_at                                   <NA>       <NA>
2       3   PA1816_dnaQ_at    DNA polymerase III, epsilon chain       dnaQ
3       4    PA3183_zwf_at glucose-6-phosphate 1-dehydrogenase        zwf
4       6   PA4407_ftsZ_at            cell division protein FtsZ       ftsZ
5       7 Pae_16SrRNA_s_st                                   <NA>       <NA>
6       8 Pae_23SrRNA_s_st                                   <NA>       <NA>
    P-Value FoldChange Flag
1 0.7583680   0.946678    0
2 0.5312780   0.882740    0
3 0.1065430   0.589877    0
4 0.0857105   0.912617    0
5 0.6743540   0.878799    0
6 0.7226290   0.912209    0
```

Now all 181 pre–selected genes are extracted as `data.frame` together with the corresponding annotation and the filter mask.

It is also possible to create a fold-change vs p-value plot, called `volcanoplot`. Setting the parameter `labels="fSymbol"` allows us to draw the corresponding gene symbols, if known:

```
> volcanoplot(rma.ufr, labels = "fSymbol")
```

A Appendices

A.1 Importing chip definition and annotation files

In contrast to other packages, which rely on the Bioconductor method for creating cdf environments, we need to create ROOT *scheme* files directly from the Affymetrix source files, which need to be downloaded first from the Affymetrix web site. However, once created, it is in principle possible to distribute the ROOT *scheme* files, too.

Here we will demonstrate, how to create a ROOT *scheme* file for Affymetrix GeneChip *Test3.CDF*. We assume that the following files were downloaded, unzipped, and saved in subdirectories `libraryfiles` and `Annotation`, respectively:

- GeneChip chip definition file: *Test3.CDF*

- Probe sequence file: *Test3_probe.tab*

- Probeset annotation file: *Test3.na28.annot.csv*

In a new R-session we load our library and define the directories, where the library files and the annotation files are saved, respectively, and the directory, where the ROOT *scheme* files should be saved:

```
> library(xps)
> libdir <- "/path/to/Affy/libraryfiles"
> anndir <- "/path/to/Affy/Annotation"
> scmdir <- "/path/to/CRAN/Workspaces/Schemes"
```

Now we can create a ROOT *scheme* file:

```
> scheme.test3 <- import.expr.scheme("Scheme_Test3_na28", filedir = scmdir,
+     paste(libdir, "Test3.CDF", sep = "/"), paste(libdir, "Test3_probe.tab",
+         sep = "/"), paste(anndir, "Test3.na28.annot.csv", sep = "/"))
```

The R object *scheme.test3* is not needed lateron, since in every new R-session the ROOT *scheme* file need to be imported first, using:

```
> scmdir <- "/path/to/CRAN/Workspaces/Schemes"
> scheme.test3 <- root.scheme(paste(scmdir, "SchemeTest3.root",
+     sep = "/"))
```

Package xps includes a file "script4schemes.R" which contains code to import some of the main CDF and annotation files, which can be copied to an R-session, including code to create ROOT *scheme* files for the currently available Exon arrays and Gene arrays.

Note 1: Since ROOT *scheme* files need to be created only once, it is recommended to save them in a common system directory, e.g. 'Schemes', which is accessible to other users, too.

Note 2: As mentioned earlier, slot `mask` of *scheme.test3* needs to be attached to instances of S4 class *DataTreeSet* before accessing raw data, since slot `mask` contains the information which oligos on the array are PM, MM, or control oligos, respectivly. If you want to avoid this step you can create instances of *SchemeTreeSet*, which contain this information already, by setting parameter `add.mask` of function `import.expr.scheme` to `add.mask=TRUE`, e.g.:

```
> scheme.test3 <- import.expr.scheme("Scheme_Test3_na28", ...,
+     add.mask = TRUE)
```

Note 3: Please note that for the new GeneChip Exon array systems and Gene array systems Affymetrix no longer supports CDF-files, but uses the new CLF-files and PGF-files instead. For this reason package xps also uses CLF-, PGF-files to create the root scheme files, and does not use the inofficial CDF-files. See the help files `?import.exon.scheme` and `?import.genome.scheme` for more information.

A.2 Additional examples

Additional examples how to use package xps can be found in file "script4xps.R", located in subdirectory 'examples'. Most of these examples are easily adaptable to users need and can be copied with no or only minor modifications. Furthermore, a second file, "script4exon.R", shows how to use xps with the novel Affymetrix Gene and Exon arrays. Both files use the Affymetrix "Human Tissue Datasets" for arrays HG-U133_Plus_2, HuEx-1_0-st-v2 and HuGene-1_0-st-v1, respectively.

A.3 Using Biobase class ExpressionSet

Some users may prefer to use S4 class *ExpressionSet*, defined in the Biobase package of Bioconductor, for further analysis of expression measures.

Package Biobase contains a vignette "ExpressionSetIntroduction.pdf", which describes how to build an *ExpressionSet* from scratch. Here we create a minimal *ExpressionSet* containing the expression measures determined using RMA:

First, we need to load library Biobase, then extract the expression levels from instance `data.rma` of class *ExprTreeSet*, convert the data.frame to a matrix, and finally create an instance of class *ExpressionSet*:

```
> library(Biobase)
> expr.rma <- validData(data.rma)
> minimalSet <- new("ExpressionSet", exprs = as.matrix(expr.rma))
```

As described in vignette "ExpressionSetIntroduction.pdf", we can now access the data elements. For this example we create a new *ExpressionSet* consisting of the 5 features and the first 3 samples:

```
> vv <- minimalSet[1:5, 1:3]
> featureNames(vv)
> sampleNames(vv)
> exprs(vv)
```

This class *ExpressionSet* can now be used from within other Bioconductor packages.

A.4 ROOT graphics

As noted earlier, package xps allows to analyze Exon arrays on computers with only 1GB RAM. However, in this case it is not possible to use R-based plots such as `image`, `hist`, or `boxplot`, respectively. For this purpose xps takes advantage of the `ROOT` graphics capabilities, which do not suffer from such memory limitations.

In the following we will demonstrate some of the `ROOT` graphics capabilities using the 33 exon array data of all 11 tissues from the Affymetrix Exon Array Data Set "Tissue Mixture" (see file "script4exon.R").

Let us first create an image using function `root.image`:

```
> root.image(data.exon, treename = "BreastA.cel", zlim = c(3, 11),
+       w = 400, h = 400)
```
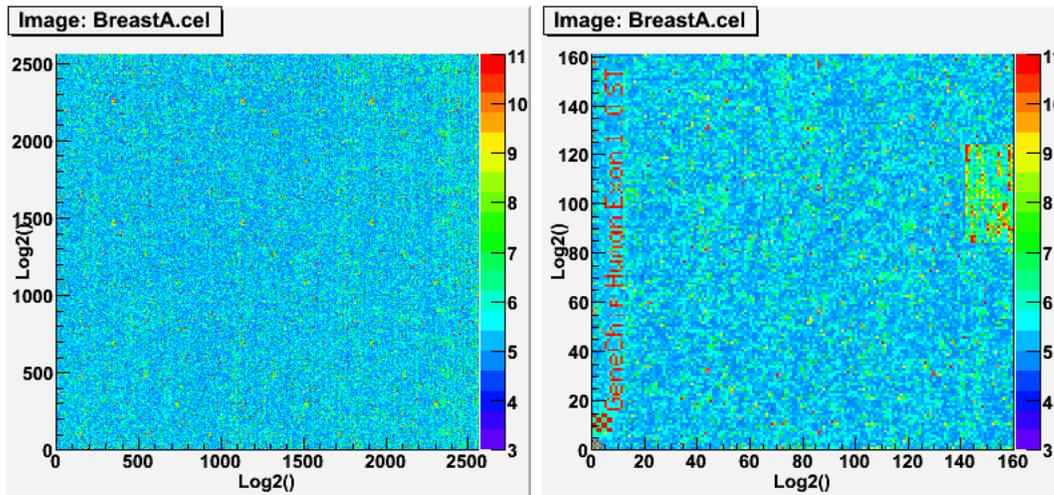
The left side of the figure shows the image created, while the right side shows the figure after zooming-in (see `?root.image` how to save the image and how to zoom-in).
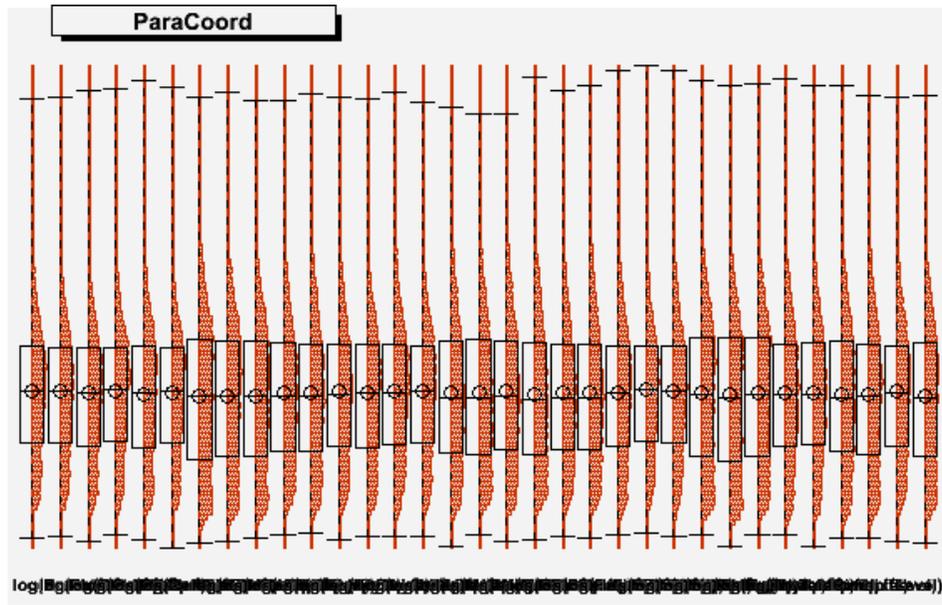
Now let us create density-plots for the raw intensities of all 33 exon arrays, as well as for the RMA-normalized expression levels:

```
> root.density(data.exon, "*", w = 400, h = 400)
> root.density(data.x.rma, "*", w = 400, h = 400)
```
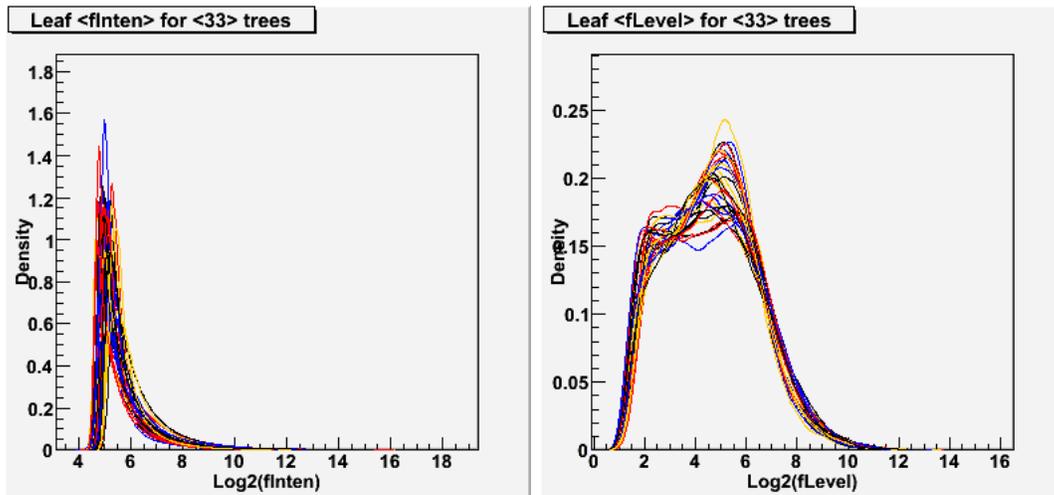

In addition we create profile plots for the RMA-normalized expression levels:

```
> root.profile(data.x.rma, w = 640, h = 400)
```

As you see, the profile plots shows both a histogram and a boxplot for each sample.

It is also possible to draw scatter-plots for the raw intensities between any two arrays, as well as between two RMA-normalized expression levels:
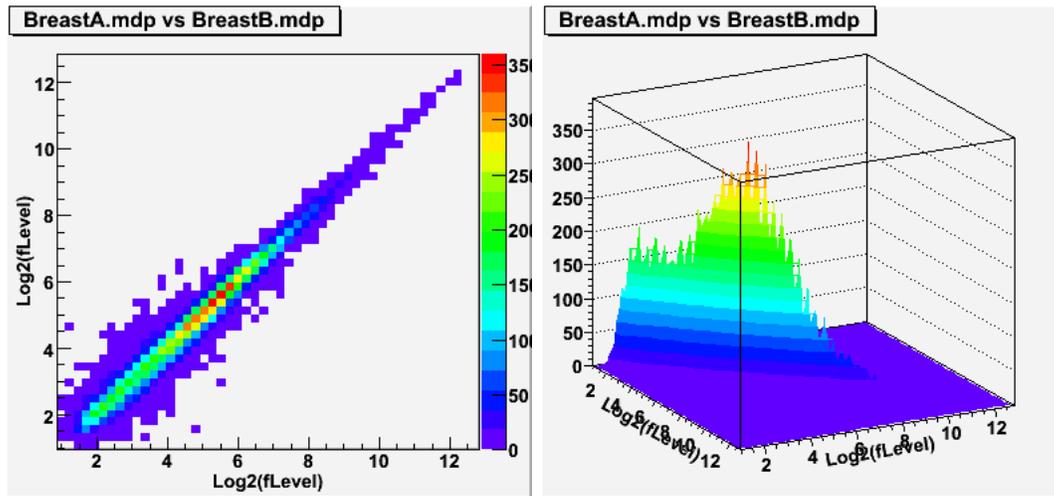
```
> root.graph2D(data.exon, "BreastA.cel", "BreastB.cel")
> root.graph2D(data.x.rma, "BreastA.mdp", "BreastB.mdp")
```
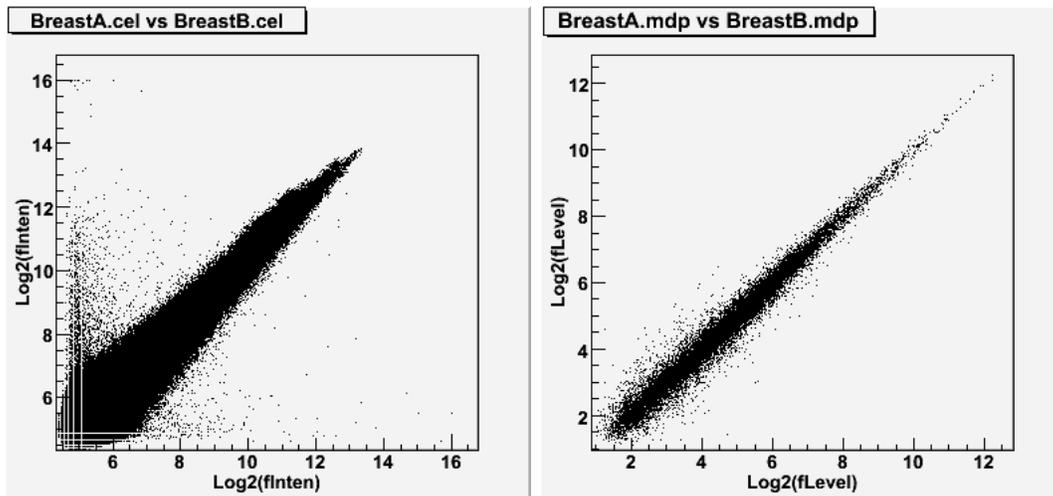

The left scatter-plot compares the raw intensities of two breast tissue replicas for all probes on the exon array, while the right scatter-plot compares the respective normalized expression levels.

Besides using scatter-plots it is also possible to plot the same data as 2D-histograms:

```
> root.hist2D(data.exon, "BreastA.cel", "BreastB.cel", option = "COLZ")
> root.hist2D(data.x.rma, "BreastA.mdp", "BreastB.mdp", option = "COLZ")
> root.hist2D(data.x.rma, "BreastA.mdp", "BreastB.mdp", option = "SURF2")
```


Here we show two different ways to plot the 2D-histogram for the normalized expression levels by simply changing the parameter option. The left histogram uses the default option="COLZ" while the right histogram was created using option="SURF2" to allow a 3-dimensional view of the expression level distribution.

Finally, it is also possible to create 3D-histograms:

```
> root.hist3D(data.exon, "BreastA.cel", "BreastB.cel", "BreastC.cel",
+     option = "SCAT")
> root.hist3D(data.x.rma, "BreastA.mdp", "BreastB.mdp", "BreastC.mdp",
+     option = "SCAT")
```
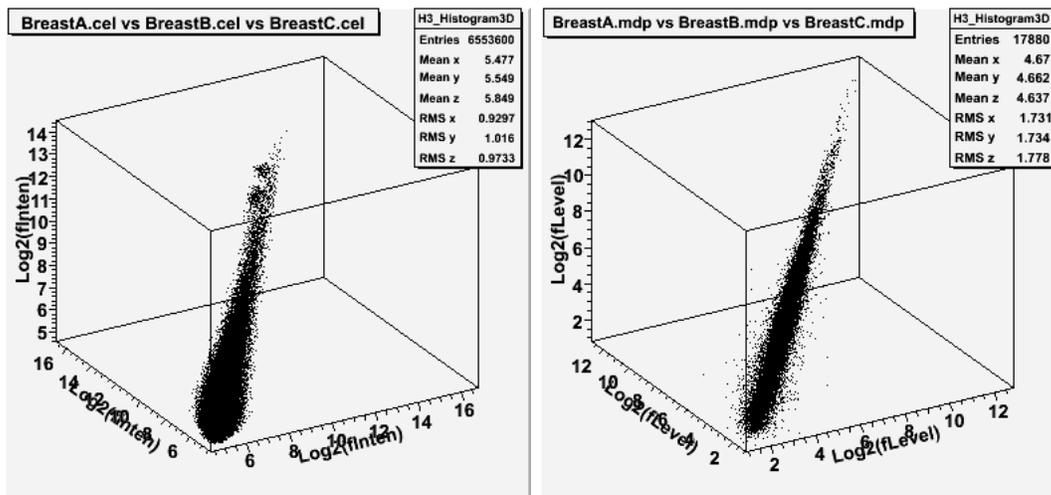
The left 3D-histogram compares the raw intensities of the breast tissue triplicates for all probes on the exon array, while the right scatter-plot compares the respective normalized expression levels.

Since quite often samples are hybridized onto arrays as triplicates, 3D-histograms are helpful in getting a first impression on the quality of the triplicates.

Note: The 3D-histograms can be rotated interactively, see `?root.hist3D`.

A.5   Using methods FARMS and DFW

Analogously to method `medianpolish`, used for `rma`, both `farms` and `dfw` are multichip summarization methods. The algorithm for FARMS (Factor Analysis for Robust Microarray Summarization) is described in (Hochreiter et al., 2006) and is available as package `farms`. The algorithm for DFW (Distribution Free Weighted Fold Change) is described in (Chen et al., 2007) and the R implementation can be downloaded from the web site of M.McGee. Both authors claim that their respective methods outperform method `rma` (see also Affycomp II: A benchmark for Affymetrix GeneChip expression measures).

The R implementation of both methods requires package `affy` since both methods must be registered with `affy`. In contrast, package `xps` implements both summarization methods in C++ and thus does not require any additional package.

In general, summarization methods are implemented in package `xps` as C++ classes derived from base class `XExpressor`. Thus summarization method `medianpolish` is implemented as class `XMedian-Polish`, while methods `farms` and `dfw` are implemented as classes `XFARMS` and `XDFW`, respectively.

To use FARMS you simply do:

```
> data.farms <- farms(data.test3, "tmp_Test3FARMS", verbose = FALSE)
```

To use DFW you simply do:

```
> data.dfw <- dfw(data.test3, "tmp_Test3DFW", verbose = FALSE)
```

Since the authors of both algorithms recommend to use their summarization methods with quantile normalization but without background correction, methods `farms` and `dfw` follow these suggestions. Users wanting to use both methods with a background correction method need to use the general

method `express` (see `?express`).

In addition to FARMS as summarization method the authors have also implemented a novel filtering method, called I/NI-calls, to exlcude non-informative genes, see (Talloen et al., 2007). This method cannot only be used with FARMS but also together with other methods to compute expression measures such as RMA.

To use I/NI-calls you simply do:

```
> call.ini <- ini.call(data.test3, "tmp_Test3INI", verbose = FALSE)
```

Note: Although package farms is available under the GNU General Public License, the authors state on their web site that: "This package (i.e. farms_1.x) is only free for non-commercial users. Non-academic users must have a valid license." Since I do not know if this statement applies for my C++ implementation, too, it is recommended that respective users contact the authors of the original package.

References

Affymetrix. *Affymetrix Microarray Suite User Guide.* Affymetrix, Santa Clara, CA, version 4 edition, 1999.

Affymetrix. *Affymetrix Microarray Suite User Guide.* Affymetrix, Santa Clara, CA, version 5 edition, 2001.

Z. Chen, M. McGee, Q. Liu, and R.H. Scheuermann. A distribution free summarization method for affymetrix genechip arrays. *Bioinformatics*, 23(3):321–327, Feb 2007.

Laurent Gautier, Leslie Cope, Benjamin Milo Bolstad, and Rafael A. Irizarry. affy - an R package for the analysis of affymetrix genechip data at the probe level. *Bioinformatics*, 20(3):307–315, Feb 2004.

S. Hochreiter, D.A. Clevert, and K. Obermayer. A new summarization method for affymetrix probe level data. *Bioinformatics*, 22(8):943–949, Apr 2006.

Rafael A. Irizarry, Bridget Hobbs, Francois Collin, Yasmin D. Beazer-Barclay, Kristen J. Antonellis, Uwe Scherf, and Terence P. Speed. Exploration, normalization, and summaries of high density oligonucleotide array probe level data. *Biostatistics*, 4(2):249–264, 2003.

W. Talloen, D.A. Clevert, K. Obermayer, D. Amaratunga, J. Bijnens, S. Kass, and H.W.H. Gohlmann. I/ni-calls for the exclusion of non-informative genes: a highly effective filtering tool for microarray data. *Bioinformatics*, 23(21):2897–2902, Nov 2007.

The ROOT team. ROOT User Guide. Technical report, CERN, 2009. URL `http://root.cern.ch/drupal/content/users-guide`.