CALIB package User's Guide

Hui Zhao, Kristof Engelen, Bart DeMoor and Kathleen Marchal
Katholieke University Leuven, Belgium

July, 2006

Table of contents

1. Introduction

Normalization of microarray measurements is the first step in a microarray analysis flow. It aims at removing consistent sources of variations to make measurements mutually comparable. Reliable normalization is essential since the results of all subsequent analyses (such as e.g. clustering) might largely be influenced by the normalization procedure. For normalization of two-color arrays different methods have been described. Although some approaches inherently work with absolute intensities (e.g. ANOVA [1]), in general, preprocessing of two-color microarrays largely depends on the calculation of the log-ratios of the measured intensities. A common normalization step consists of the linearization of the Cy3 vs. Cy5 intensities (e.g. loess [2]). It assumes the distribution of gene expression is balanced and shows little change between the biological samples tested (to which we refer as Global Normalization Assumption or GNA). Global mRNA changes that result in an uneven distribution of expression changes however, have been shown to occur more frequently than what is currently believed [3,4], and could have a significant impact on the interpretation of data normalized according to the Global Normalization Assumption.

Recently, a different way of normalizing two-color microarray data was proposed that obviates the GNA and that poses several advantages over ratio based approached (for details see Engelen *et al*., 2006 [5]). Briefly, the normalization is based on a physically motivated model, explicitly modeling the hybridization of transcript targets to their corresponding DNA probes, and the relation between the measured fluorescence and the amount of hybridized, labeled target. The parameters of this model and incorporated error distributions are estimated from external control spikes: targets that are added to the hybridization solution in known concentrations. This, together with the inherent nonlinearity of the model, allows normalizing the data without making any assumptions on the distribution of gene expression (as opposed to procedures relying on the GNA). More importantly, since our model links target concentration to measured intensity, estimating absolute expression levels of transcript targets in the hybridization solution becomes possible.

Here we describe the implementation of this method as a Bio-Conductor package [6,] called CALIB. This package allows normalizing two-color microarray data, using the method mentioned above. A spike-based calibration model is used to estimate absolute transcript levels for each combination of a gene and tested biological condition, irrespective of the number of microarray slides or replicate spots on one slide.

2. Classes

Three data objects are used for storing data in the CALIB package.

RGList_CALIB: A list used to store raw measurement data after they are read in from an image analysis output file, usually by *read.rg*(). The *RGList_CALIB* in this package is an extended *limma::RGList* from the *Limma* package [7,8]. As compared to the *limma::RGList* it contains two additional fields, *RArea* and *GArea* . These two additional fields are meant to store the spot areas, which in some cases are needed to calculate measured intensities. While the natural measure of spot intensity is the sum of pixel intensities within the spot mask, often only mean or median pixel intensities are reported as the ratio of averages is equivalent to the ratio of sums. Since CALIB does not work with intensity ratios, in cases were only mean or median pixel intensities are reported, the spot area is required to calculate reliable spot intensity values.

SpikeList: A list used to store raw measurement data of all external control spikes spotted on the arrays. An object of this class is created by *read.spike()*. It is a subset of the object of *RGList_CALIB* plus two fields, *RConc* and *Gconc* to indicate known concentrations for the control spikes' targets added to the hybridization solution and labeled in red and green respectively.

ParameterList: A list used to store parameters of the calibration model for each array. An object of this class is created by *estimateParameter()*.

RGList_CALIB, SpikeList and *ParameterList* are inherited directly from the R data type *LIST*. Therefore, methods that work on the LIST data type (e.g. *summary, dim, length, ncol, nrow, dimnames, rownames, colnames*) can be used on these three classes. For example,

```
> dim(spike)

[1] 600  2
```

Shows that the *SpikeList* object spike contains data for 600 spikes and 2 arrays.

```
> colnames(spike)
```

will return the column names for the spike object, i.e. the names of the arrays.
Combination methods can be used on multiple data objects of these classes: *cbind*, *rbind* and *merge* can be used on the *RGList_CALIB* class and the *SpikeList* class. On the *ParameterList* class, only *cbind* and *merge* can be used, as it makes no sense to row combine different *ParameterList* objects. Combination methods can be used as shown below:

```
> RG1 <- read.rg(files[1:2], source = "genepix")
> RG2 <- read.rg(files[3:5], source = "genepix")
> RG <- cbind(RG1,RG2)
```

Since the *RGList_CALIB* was adopted from the *Limma* package, we used an example of the *Limma* package to explain the usage of this class. Some of the *limma::RGList* functions of the *Limma* package were extended in *RGList_CALIB* to better suit our needs.

3. Example data file

In order to illustrate the workings and principles of the CALIB method and the usage of the functions in the package, we included a test set containing two out of fourteen hybridizations of a publicly available benchmark data set [9]. The experimental design of these two arrays consists of a color-flip of two conditions. The usage of the package is illustrated in this guide by means of this test example.

Raw data, spikes, model parameters and normalized data are stored in the data files *RG.rda, spike.rda, parameter.rda and normdata.rda* respectively. These examples can be read into R and then used as input of the appropriate function in the package (note that the raw data files (.txt file) are not included in the package). For example,

```
> data(RG)
> data(spike)
> par <- estimateParameter(RG, spike)
```

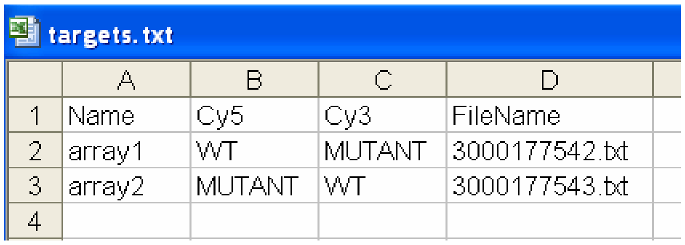
By using *data(RG)*, *RG.rda* which stores an *RGList_CALIB* object is read into R. This R object is called RG and can be used as input of function *estimateParameter()*, whose usage will be illustrated in section six of this guide. The function *data(spike)* is used in the same way for reading *spike.rda* into R in this example.

4. Reading data

4.1 Reading two-color microarray Data

Since the *RGList_CALIB* used to store microarray data in the CALIB package is an extension of the *limma::RGList*, the functions used for reading raw data into an *RGList* object in *Limma* are also applicable in the CALIB package.

Following the steps in *Limma*, at first a target file needs to be created. It should be a tab-delimited text file with the default name "targets.txt", and basically describe the experiment design, listing for each array, its name, the file where the data corresponding to this array can be found and the condition/dye combinations measured on this array. Figure 1 gives an example of the target file.



targets.txt

	A	B	C	D
1	Name	Cy5	Cy3	FileName
2	array1	WT	MUTANT	3000177542.txt
3	array2	MUTANT	WT	3000177543.txt
4				

Figure 1. An example of the target file "targets.txt'

The following command is used to read the target file into R.

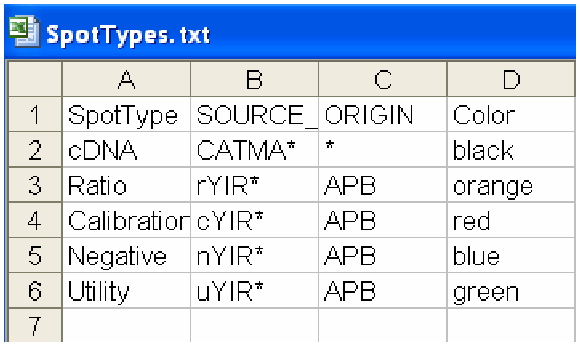
```
> targets <- readTargets()
```

The *targets* object contains a column labeled "FileName". The entries in this column correspond for each array to the name of the image analysis output file. This column is used as input argument of the function to read in the intensity data:

```
> RG <- read.rg(targets$FileName, source="<imageanalysisprogram>",path="<filedirectory>")
```

where <imageanalysisprogram> is the name of the image analysis program used to generate the raw datafile and <filedirectory> is the full path of the directory containing the raw data files. At present time, the CALIB package supports three image analysis programs: "genepix", "genepix.median" and "quantarray". *RG* is an *RGList_CALIB* object containing the imported microaray data.

After importing the raw data into R, a specific spot type will be assigned to each of the different spots on the array (or different rows of the *RG*). This is done by first reading in the spot file (an example is given in figure 2) which specifies the different user specified spot types (descriptive expression used to make a distinction the regular cDNA probes and the different types of control spots (ratio, calibration,…)) and has the default name "SpotType.txt" and secondly, by setting the status of each spot on the array to one of the user specified spot types. This allows discriminating in the *RG* between regular cDNA spots, as opposed to control spots from which the model parameters will be estimated. Note that while these steps are optional in *Limma* package, they are required in the CALIB package.



	A	B	C	D
1	SpotType	SOURCE_	ORIGIN	Color
2	cDNA	CATMA*	*	black
3	Ratio	rYIR*	APB	orange
4	Calibratior	cYIR*	APB	red
5	Negative	nYIR*	APB	blue
6	Utility	uYIR*	APB	green
7				

Figure 2. An Example of spot type file "SpotType.txt"

The spot type file uses simplified regular expressions to match patterns. For example, AA* means any string starting with AA, AA. means AA followed by exactly one other character and AA\. means AA followed by a period and no other characters.

The status of the calibration controls, the ratio controls and the negative controls should be Calibration, Ratio and Negative respectively.

The function used to read user-specified spot type files is *readSpotType()*.The function used to control the status of each spot on the array is *controlStatus()*. They are used as follows:

```
> spottypes <- readSpotType()
> RG$genes$Status <- controlStatus(spottypes, RG)
```

The function *readSpotType()* takes the spot type file name as input argument. In the example, default name "SpotType.txt" is given to the spot type file. Therefore, *readSpotType()* just takes the default argument.

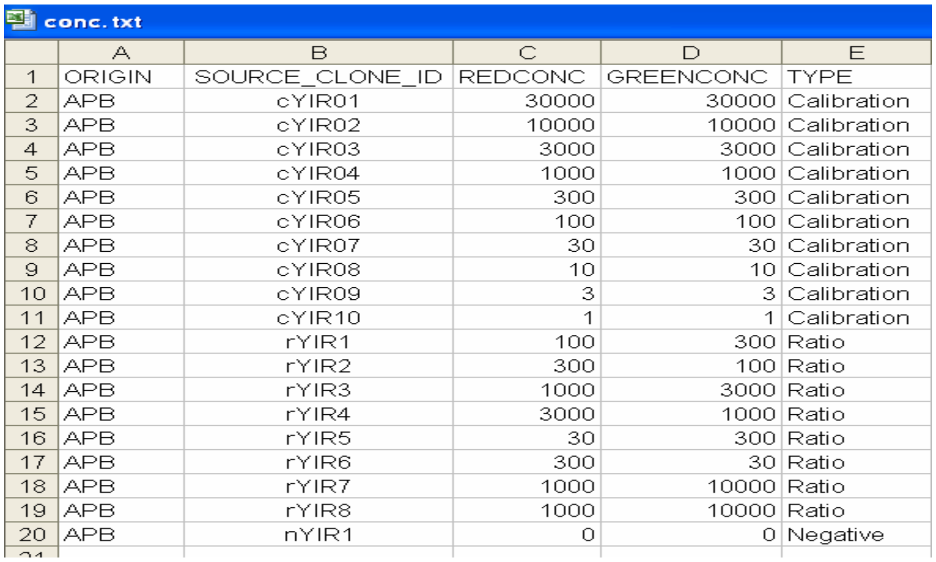
For details regarding the usage of these functions and the format of the target file and the spot type file and for some more optional functions, we refer to the limma user's guide in the *Limma* package.

4.2 Reading Spike Data

Spike data are stored in a *SpikeList* object. The data in the *SpikeList* object correspond to a subset of the data stored in the *RGList_CALIB* object, i.e. to these data that correspond to the measurements of the externally added control spikes as specified by their user defined spot status.

In addition, the *SpikeList* object contains two more fields: *RConc* and *GConc*. The entries in these fields correspond to the absolute concentrations of labeled mRNAs for each of the control spikes. *RConc* and *GConc* are set by reading information from a user-specified concentration file.

An example of a concentration file is given below in figure 3: it is a tab-delimited text file in which each row corresponds to a specific control spike, identified by the rows' entry in an identifier column, in this case with the name "SOURCE_CLONE_ID". This name of the identifier column should match the name of a column in the *RG$genes* (where *RG* is an *RGList_CALIB* object) of which the entries contain patterns or regular expressions sufficient to uniquely identify each different spike. The entries of each row in the columns REDCONC and GREENCONC by default contain the absolute concentrations of the spike, labeled with respectively the red and the green dye added to the hybridization solution. For each row, the entry in the TYPE column indicates the type of the particular spike. The names used to indicate spike types in the TYPE column should match the names used to indicate the spike status (or spot type) in the *RGList_CALIB* object.


conc.txt

	A	B	C	D	E
1	ORIGIN	SOURCE_CLONE_ID	REDCONC	GREENCONC	TYPE
2	APB	cYIR01	30000	30000	Calibration
3	APB	cYIR02	10000	10000	Calibration
4	APB	cYIR03	3000	3000	Calibration
5	APB	cYIR04	1000	1000	Calibration
6	APB	cYIR05	300	300	Calibration
7	APB	cYIR06	100	100	Calibration
8	APB	cYIR07	30	30	Calibration
9	APB	cYIR08	10	10	Calibration
10	APB	cYIR09	3	3	Calibration
11	APB	cYIR10	1	1	Calibration
12	APB	rYIR1	100	300	Ratio
13	APB	rYIR2	300	100	Ratio
14	APB	rYIR3	1000	3000	Ratio
15	APB	rYIR4	3000	1000	Ratio
16	APB	rYIR5	30	300	Ratio
17	APB	rYIR6	300	30	Ratio
18	APB	rYIR7	1000	10000	Ratio
19	APB	rYIR8	1000	10000	Ratio
20	APB	nYIR1	0	0	Negative

Figure 3. An example of concentration file "conc.txt"

In this example, two identifier columns ORIGIN and SOURCE_CLONE_ID are used.
For a concentration file named conc.txt a *SpikeList* object can be created by

```
> spike <- read.spike(RG, file = "conc.txt", path = <filedirectory>)
```

When the concentration column names differ from the default names “REDCONC” and “GREENCONC” (for example rconc and gconc) the function can be called as follows:

```
> spike <- read.spike(RG, file = “conc.txt”, conccol = list (RConc = “rconc”, GConc = “gconc”), path = <filedirectory>)
```

In both previous examples, we assumed that the same spike set (e.g. a commercial kit) was hybridized to all arrays. When this is not the case, for instance, when the same spike set is added in different concentrations on the different arrays, or when different dye configurations are used for the same spike set hybridized to the different arrays (e.g. an experiment where ratio spikes are ‘color-flipped’ from one array to another), the user has to specify a separate concentration file for each array. For example, if we have two different arrays with different sets of spikes hybridized to them, a single concentration file is defined for each array; say “conc1.txt” and “conc2.txt”. In case the default concentration column names are used, the *read.spike()* function should be called as follows:

```
> spike <- read.spike(RG, file = c(“conc1.txt”, “conc2.txt”), different = TRUE, path = <filedirectory> )
```

where conc1.txt and conc2.txt correspond to the concentration files of the first and second array in the *RG* respectively.
In all previous examples, a *SpikeList* called *spike* is created, containing the spike data of all the arrays.

5. Quality control

In the CALIB package, calibration model parameters are estimated from external control spikes. The final normalization thus depends on the quality of spotted spikes on the arrays. Checking the spike quality prior to estimating the model parameters is definitely advisable.
Two functions to visually inspect the quality of the spikes are provided i.e., *plotSpikeCI()* and *plotSpikeRG()*. In the following, we will use the example dataset provided by the package to explain how to create plots with these functions.
The function *plotSpikeCI()* plots for one specified array the known concentration of the spikes (corresponding to the amount of externally added labeled cDNA of each spike) against their measured intensity and this for both the red and green channel.

```
> plotSpikeCI(spike)
```

An additional argument *array* of the function *plotSpikeCI()* takes the first array as default value. Therefore, this function gives the plot (shown in figure 4) of the first array of *spike*. If the second array needs to be plotted, the argument *array* has to be specified.

```
> arraynum<- 2
> plotSpikeCI(spike, array=arraynum)
```

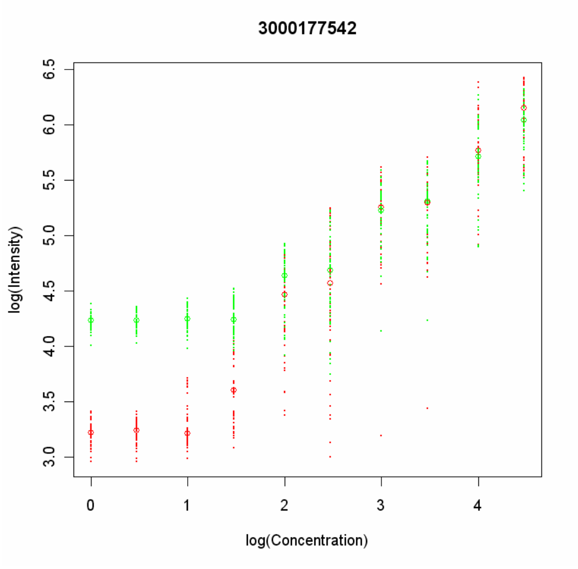


Figure 4

From this plot, a sigmoidal relationship between the measured intensities and added concentrations is to be expected. Indeed, in a certain range the relationship will be linear, but at the highest and lowest concentration levels saturation effects will occur, which might be different for the red and green channel. The function *plotSpikeRG()* plots for all spikes on a specified array the red versus the green intensities. For example, if we want to plot the red versus green intensities of the first array, the function is used as follows:

```
> plotSpikeRG(spike)
```

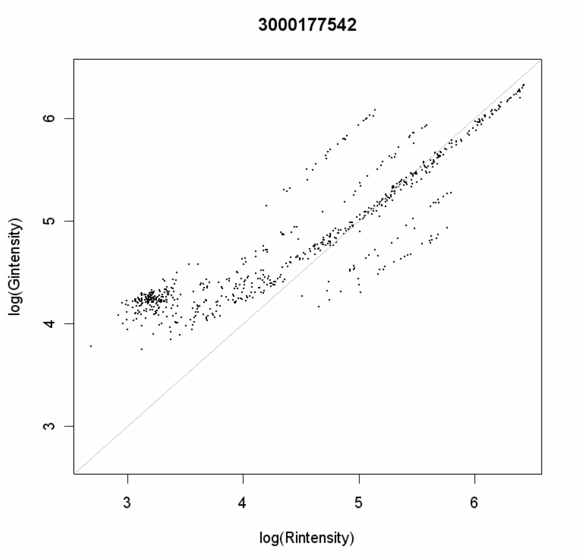


Figure 5

In this plot we expect the ratio 1:1 spikes on the one-one axis. The higher intensity levels in this plot give an idea of the multiplicative error variance on the measured intensities.

6. Estimation of model parameters

Once the quality of the spikes is assessed, spike measurements can be used to estimate the parameters of the calibration model. This model describes the relationship between measured intensities and target levels added to the hybridization solution. The function *estimateParameter()* is used to estimate the parameters of the calibration models based on the measured intensities and known concentrations of the external control spikes. It takes the *RGList_CALIB* and *SpikeList* objects as input arguments and estimates a set of parameters for each array. The function can be called as follows

```
> parameter <-  estimateParameter (RG, spike)
```

Besides the required inputs *RGList_CALIB* object and *SpikeList* object, there are three optional input arguments for this function.

- ***bc*** and ***area*** (both logical values) are used to specify how intensity values per probe need to be calculated: *bc* indicates whether a background correction is used , *area* indicates whether the foreground intensity will be multiplied with the spot area The default value of these two arguments are *bc* = FALSE (no background correction) and *area* = TRUE (multiplication performed).
- ***errormodel*** specifies the type of spot error distribution used to model the spot sizes. As described in Engelen *et al*. 2006, the calibration model contains a hybridization reaction which is modeled as

  $$\frac{x_s}{x_0(s_0 - x_s)} = K_A$$

  In this equation, the spot capacity s_0 is assumed to follow a certain distribution around an average spot capacity. Two possible distributions are provided: either the spot capacity is additive $\mu_s : s_0 = \mu_s + \varepsilon_s$ (errormodel =”A”), or the spot capacity error is multiplicative $s_0 = \mu_s e^{\varepsilon_s}$ (errormodel =”M”) in both cases with the spot error $\varepsilon_s \sim N(0, \sigma_s)$.

If all arguments are given by the user, *estimateParameter()* can be used like

```
> parameter <- estimateParameter(RG, spike, bc = FALSE, area = TRUE, errormodel = “M”)
```

The output of this function is an object of *ParameterList* (named *parameter* in the above examples), containing for each array the model parameters and the user specified input arguments (the bc and area values and the type of errormodel).

Results can be accessed by,

```
> show (parameter)
```

or

```
> summary (parameter)
```

which allow viewing the compacted print-out and the summary of the *parameter* object.

After the parameter estimation, an option is provided to adjust the estimated lower saturation level (indicated by P2 in the parameter list) for either one of both channels. This is needed in these cases, where the lower saturation limit for the spike measurements is seen significantly below the saturation for other data points. This is illustrated in the Figure 6 below, where the lower green intensity levels for the spikes (black dots) are generally below those of the other data points (grey dots; the blue curve represents the estimated parameters), indicating that the estimated parameter P2 for the green channel (based on the spikes) is not ideal for normalizing the other data points.

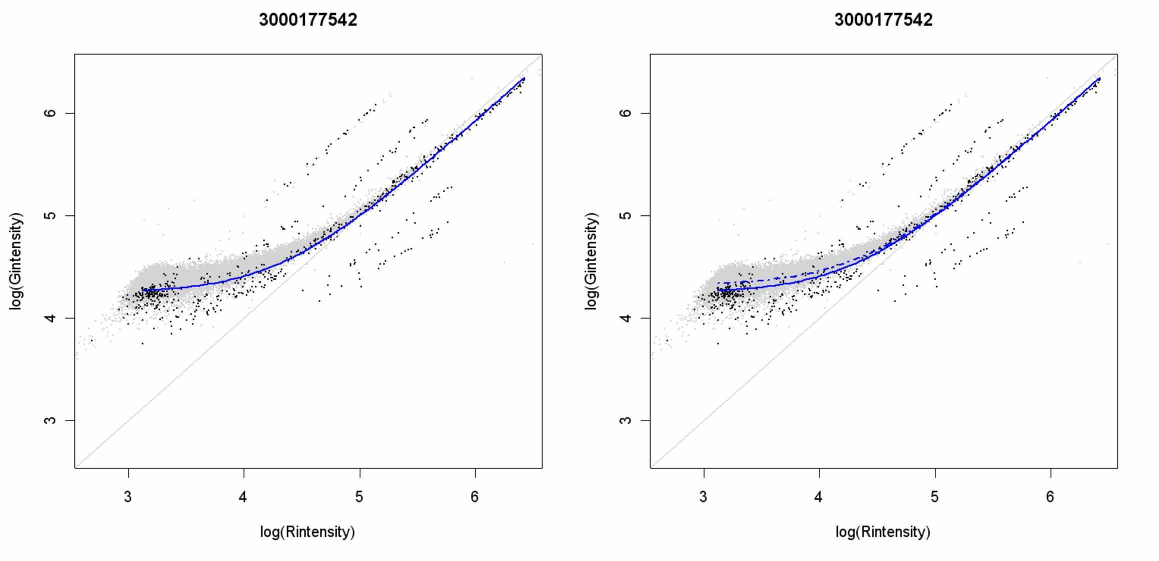


Figure 6.

The cause for this discrepancy in lower intensity level between the spike intensities and the intensities of the other data points is not known. A possible explanation might be that fractions of degraded target more or less specifically bind to their corresponding probes raising the background. Industrially synthesized spikes are purer (less degradation), so the effect would be limited to the measurements of the actual gene probes. A higher incorporation rate of Cy3 labels could explain the difference between both channels (i.e. smaller, degraded targets might still get labeled with Cy3, but not with Cy5).

As describing in section five, the function *plotSpikeRG()* can be used to check the quality of the spikes. It can also be used to evaluate the necessity for an adjustment of P2 values if it is called with different arguments as follows:

```
> plotSpikeRG(spike, parameter, RG)
```

This function will give plot shown in figure 6 (left hand panel). In the example, this plot indicates that adjustment of cy3 is necessary.

When deemed necessary, function *adjustmentP2()* is available for the adjustment. The user can specify the array and channel for which the adjustment will be performed by giving the arguments *arrayindex* and *colorindex* respectively. For the *colorindex,* 1 means red and 2 means green. For example,

```
> parameter -> adjustP2(RG, parameter, arrayindex = c(1,2), colorindex = c(1,2))
```

means that the arrays with index 1 and 2 (according to the data in RG) should be adjusted. For array 1, P2 of the red channel should be adjusted, while for array 2 P2 of the green channel should be adjusted. The *ParameterList* object *parameter* gets an additional field called *AdjustFactor* for the adjustment factor of P2. After adjustment, the evaluation function can also be called to check the result of adjustment. For example,

```
> plotSpikeRG(spike, parameter, RG)
```

Figure 6 (right hand panel) gives the plot created by this function. These two plots show that after adjustment (right hand panel) the estimated model indicated by dotted line on the plot fits the data better than without adjustment (left hand panel).

7. Normalization

Once the calibration curves for the red and green channels have been estimated for each array, they can be used to normalize the data. The function *normalizeData()* is used to this and estimates absolute expression levels for each combination of a gene and condition in the experiment design, regardless of the number of replicates. This is conceptually shown for a loop design of three conditions in the figure 7 below.

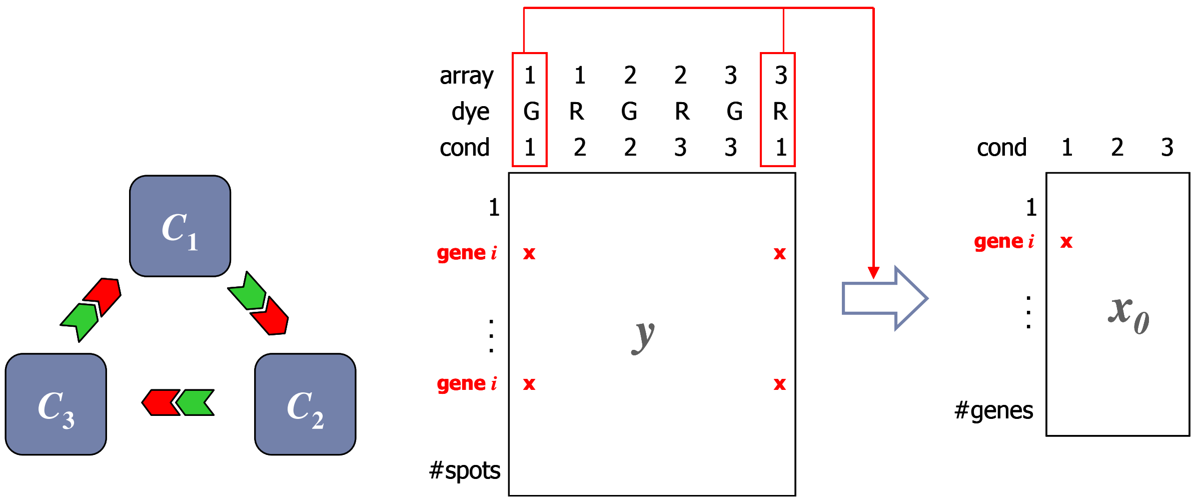
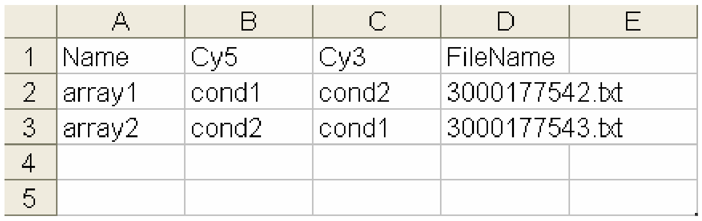


Figure 7.

The function *normalizeData()* takes as input arguments the raw data in the form of an *RGList_CALIB* object, and the estimated model parameters in the form of a *ParameterList* object. The settings of the *bc*, *area* and *errormodel*, also needed to perform the normalization are also obtained from the *ParameterList* object.

The design of the array is specified by the input arguments ***array***, ***condition*** and ***dye***. Each of them is an integer vector with equal length.

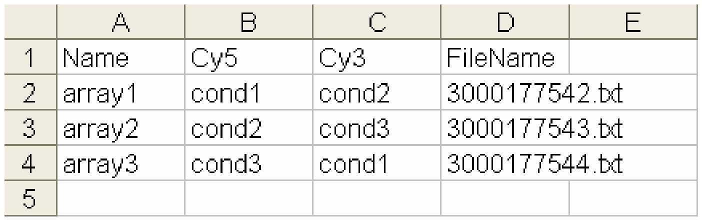


	A	B	C	D	E
1	Name	Cy5	Cy3	FileName	
2	array1	cond1	cond2	3000177542.txt	
3	array2	cond2	cond1	3000177543.txt	
4					
5					

Figure 8.

For example, for a two array color-flip design as specified in figure 8, values of array, condition and dye should be given as:

```
> varray <- c(1,1,2,2)
> vcondition <- c(1,2,2,1)
> vdye <- c(1,2,1,2)
```



	A	B	C	D	E
1	Name	Cy5	Cy3	FileName	
2	array1	cond1	cond2	3000177542.txt	
3	array2	cond2	cond3	3000177543.txt	
4	array3	cond3	cond1	3000177544.txt	
5					

Figure 9.

For or a loop design with three different conditions as specified in figure 9, values of array, condition and dye should be given as:

```
> varray <- c(1,1,2,2,3,3)
> vcondition <- c(1,2,2,3,3,1)
> vdye <- c(1,2,1,2,1,2)
```

From the examples given above, it is clear that:

- Each entry in the vector corresponds to an array channel
- Each entry of *array* indicates on which array this channel was measured. Numbers are duplicated because in a two color design, each array contains two channels.

- Each entry of *condition* gives the numeric representation of the condition measured in this channel
- Each entry in dye indicates the dye used to label this channel. Because we are dealing with two-color arrays, only two numbers are used in vector *dye*. By default 1 represents the red dye and 2 represents the green dye.

The input argument ***idcol*** specifies the column name of unique identifiers for each probe. It is possible for the same gene to occur multiple times within one array. However, during the calculation, these replicates will be combined and only one normalized value is calculated from these replicates, as indicated in figure 7. The argument is required since different arrays have different annotations. For example, this argument can be specified as follows:

```
> id_col <- “CLONE_ID”
```

After specifying all these arguments, the function can be called as follows:

```
> normdata <- normalizeData(RG, parameter,  array = varray, condition = vcondition, dye = vdye, idcol = id_col,)
```

When the function is called like this, all genes in the *RGList_CALIB* object are normalized. However, since normalization is calculated gene by gene, the normalization can also be performed on individual genes or on a group of interesting genes instead of on the whole gene set. The user can enter the set of selected genes on which normalization has to be performed by another argument of this function – ***cloneid***.

For example, if we only want to know the estimated expression level of clone “200001”, we can type

```
> cloneid_interested <- “200001”
> normdata <- normalizeData(RG, parameter,  array = varray, condition = vcondition, dye = vdye, cloneid = cloneid_interested, idcol = id_col,)
```

Or if we are interested in a group of clones, we can type

```
> cloneid_interested <- c(“200001”, “200002”, “200003”, “200004”, “200005”)
> normdata <- normalizeData(RG, parameter,  array = varray, condition = vcondition, dye = vdye, cloneid = cloneid_interested, idcol = id_col)
```

In all examples, *normdata* is the output of this function. It is a numeric matrix with rows representing individual genes and columns representing different condition. Namely, every value in this result matrix represents the expression level in a different gene-condition combination (as illustrated by the figure 7 at the beginning of this section).

8. Diagnostics and data visualization

The CALIB package provides different visualization functions that facilitate quality control and data exploration before and after parameter estimation.

The estimation of model parameters is dependent on the quality of the external control spikes. In order to ensure good normalization results, it is advisable to check the quality of the external controls prior to normalization, by using visualization functions described in section five.

Besides checking external control quality, the functions *plotSpikeCI()* and *plotSpikeHI()* can also be used for evaluating the model fit after parameter estimation (if the additional argument “parameter” is added). When including the estimated calibration parameters, both the data and the model fit (red and green curves) are plotted.

The function *plotSpikeCI()* should then be called as follows (where arraynum is the index of the array that will be plotted.):

```
> arraynum <- 1
> plotSpikeCI(spike, parameter, array = arraynum)
```

Figure 10 shows the plot created by this function. The red and green curves represent the estimated calibration models for the red and green channels respectively.

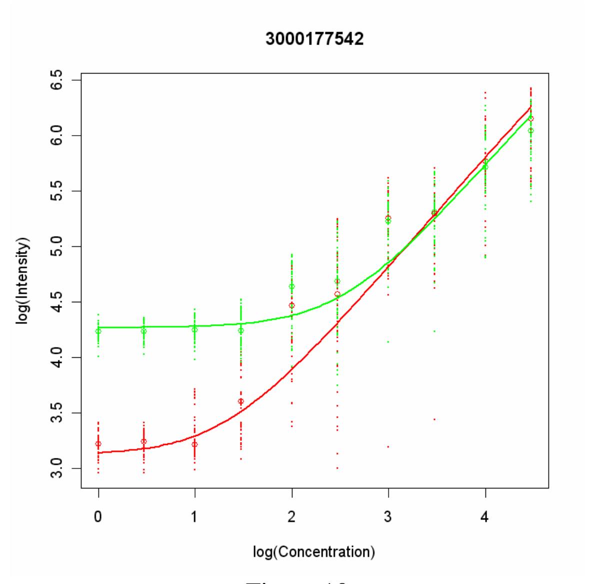


Figure 10

The function *plotSpikeHI()* should then be called as follows (again, arraynum is the index of the array that will be plotted.):

```
> arraynum <- 1
> plotSpikeHI(spike, parameter, array = arraynum)
```

The red and green curves represent the estimated calibration models for the red and green channel respectively. Light red and green dots indicate the amount of hybridized target for the measured intensities of the external control spikes if all spot capacities were equal (ε_s=0). Black dots represent the estimated amount of hybridized target by taking into account the estimated spot capacity errors, i.e. the black dots illustrate how the incorporation of spot errors into the model is an adequate tool for explaining the large variation in measured intensities. In general, the more tight and smooth (no visible artifacts) the black dots fit the model curves, the more suitable the model is for further normalization.

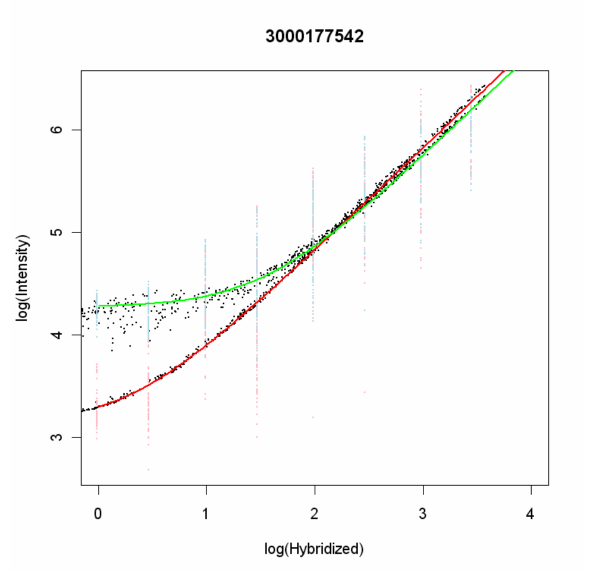


Figure 11

Note that for both *plotSpikeCI()* and *plotSpikeHI()*the default array index argument *array* is set to 1.

The function *plotSpikeSpotError()*plots the distribution of the estimated spot capacity errors for all spikes of the specified array, and can be used to ascertain the number of outliers or 'broken' spots (spots with unusually small spot capacity errors). If there are a substantial amount of these outliers, it is advisable to re-estimate the model parameters after omitting these spots from the spike set. Depending on the value of the argument *plottype* ("hist", "boxplot" and "dens") this distribution will be plotted as a histogram, a boxplot or a density plot. Resulting plots are shown in figure 12, 13 and 14.

Function *plotSpikeSpotError(),* can thus be called in the following three ways,

```
> ## plot histogram of the first array, which is the default value of the argument array.
> plotSpikeSpotError(parameter, plottype = "hist")
> ## plot boxplot of both arrays.
> plotSpikeSpotError(parameter, plottype = "boxplot", plotnames = NULL)
> ## plot density function of the first array, which is the default value of the argument array.
> plotSpikeSpotError(parameter, plottype = “dens”, width = 1)
```

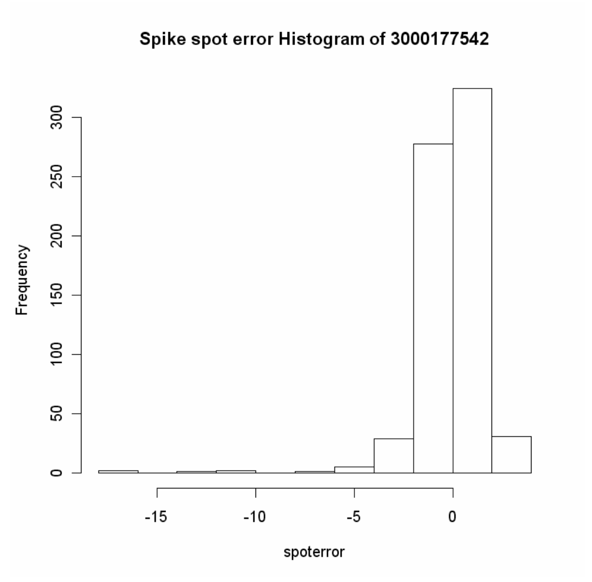


Figure 12

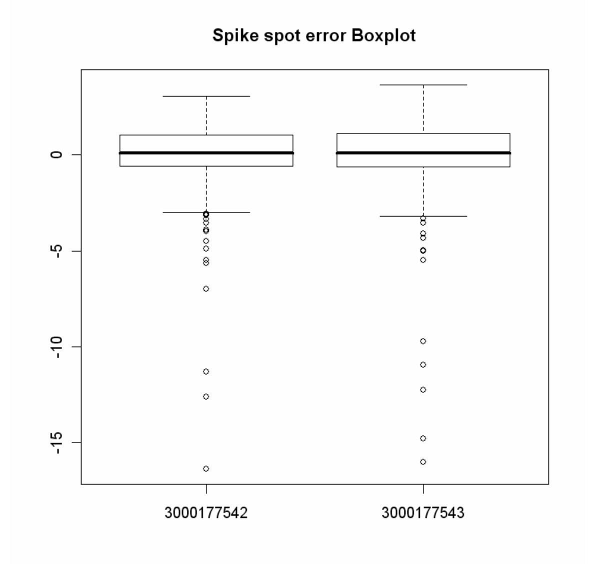


Figure 13

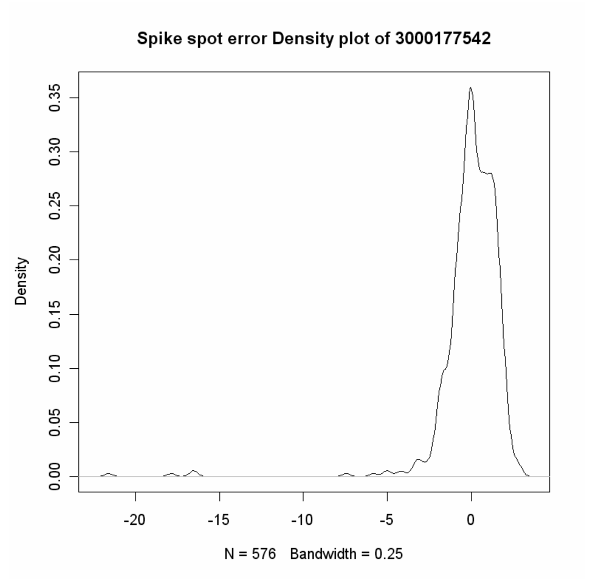


Figure 14

The function *plotNormalzedData()* can be called after data normalization and allows comparing the estimated expression levels of two selected conditions. It provides an image of the overall similarity between two conditions that were present in the experimental design, i.e. more similar conditions center more tightly around the bisector, as is the case for the example used in this guide (identical conditions). The function is called as follows:

```
> ## specify the two conditions to be plotted.
> cond <- c(1,2)
> ## use the default values for other parameters.
> plotNormalizedData(normdata,condition = cond)
```

The plot created by this function is shown as Figure 15.

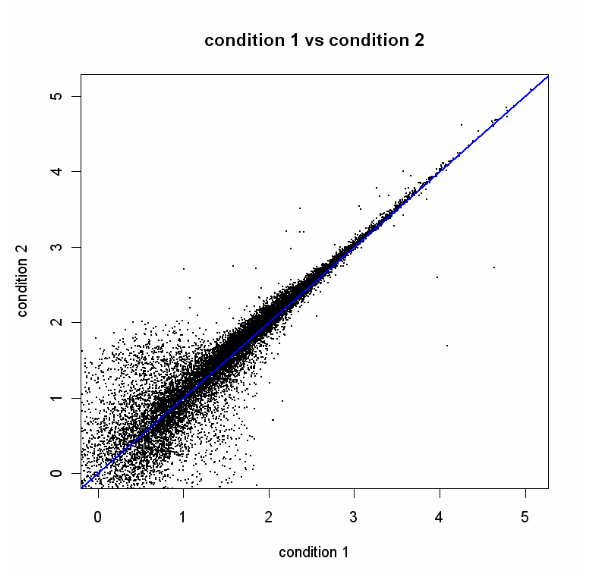


Figure 15

9. References

1. Kerr, M. K., Martin, M. & Churchill, G. A. Analysis of variance for gene expression microarray data. *J. Comput. Biol* **7**, 819-837 (2000).

2. Yang, Y. H. *et al.* Normalization for cDNA microarray data: a robust composite method addressing single and multiple slide systematic variation. *Nucleic Acids Res.* **30**, e15 (2002).

3. van Bakel,H. and Holstege,F.C. (2004) In control: systematic assessment of microarray performance. *EMBO Rep.*, 5, 964-969.

4. van de Peppel,J. et al. (2003) Monitoring global messenger RNA changes in externally controlled microarray experiments. *EMBO Rep.*, 4, 387-393.

5. Engelen, K., Naudts, B., De Moor, B. & Marchal, K. A calibration method for estimating absolute expression levels from microarray data. *Bioinformatics* **22**, 1251-1258 (2006).

6. Gentleman, R. C. *et al.* Bioconductor: open software development for computational biology and bioinformatics. *Genome Biology* **5**, (2004).

7. Smyth, G. K. Linear models and empirical bayes methods for assessing differential expression in microarray experiments. *Stat. Appl. Genet. Mol. Biol.* **3**, Article3 (2004).

8. Wettenhall, J. M. & Smyth, G. K. limmaGUI: a graphical user interface for linear modeling of microarray data. *Bioinformatics* **20**, 3705-3706 (2004).

9. Hilson, P. *et al.* Versatile gene-specific sequence tags for Arabidopsis functional genomics: transcript profiling and reverse genetics applications. *Genome Res.* **14**, 2176-2189 (2004).