

RPPanalyzer (Version 1.2)
Analyze reverse phase protein array data
User's Guide

Heiko Mannsperger¹, Stephan Gade¹,
Silvia von der Heyde² and Daniel Kaschek³

¹German Cancer Research Center, Heidelberg, Germany

²Medical Statistics, University Medical Center Göttingen, Germany

³Institute of Physics, Freiburg University, Germany

January 16, 2014

Contents

1	Introduction	2
2	Data preparation	2
2.1	Sample description	4
2.1.1	Columns plate, row and column	4
2.1.2	Column sample_type	4
2.1.3	Column sample	4
2.1.4	Columns concentration, dilution and dilSeriesID	5
2.2	Slide description	5
2.2.1	Column gpr	5
2.2.2	Columns pad, slide, incubation_run and spotting_run	5
2.2.3	Columns target and AB_ID	5
2.3	Image analysis result files	6
3	Data pre-processing	6
3.1	Read data	6
3.2	Export data as text file	7
3.3	Correct for background intensities	7
3.4	Data normalization with total protein dye	9
3.5	Quality control plots	9

4	Additional correction, quantification and normalization methods	10
4.1	Correct for background intensities	10
4.2	Quantify concentration	10
4.3	Data normalization	11
4.3.1	Total protein dyes	11
4.3.2	Housekeeping proteins	11
4.3.3	Median normalization	12
4.3.4	Protein quantification assays	12
5	Array and data selection	13
6	Visualizations	13
6.1	Time courses	13
6.2	Boxplots	14
6.3	Correlation plots	15
6.4	Heatmaps	15
7	Appendix	16

1 Introduction

In systems biology as well as in biomarker discovery reverse phase protein arrays (RPPAs) have emerged as a useful tool for the large-scale analysis of protein expression and protein activation (Paweletz et al., 2001). The method follows the basic principle of printing large numbers of raw protein extracts in parallel on a solid phase carrier to form a single array. Multiple slides are printed in parallel and each (sub)array is probed with a different monospecific antibody. To quantify protein expression or protein activation detectable signals are generated via fluorescence, dye precipitation, or chemiluminescence.

RPPAnalyzer is a compact tool, developed to perform the basic data analysis on RPPA data and to visualize the resulting biological information. It will help you with the evaluation of standard RPPA experiments. This vignette is a step by step instruction how to use the RPPAnalyzer especially written for people that are usually working in the lab and are not familiar using R. Figure 1 shows an overview of the data analysis steps.

2 Data preparation

To avoid errors during data analysis it is very important to prepare the input data exactly in the format as described in the following sections. It is not necessary to adjust the benchwork to the software but to describe exactly what you have done in the lab.

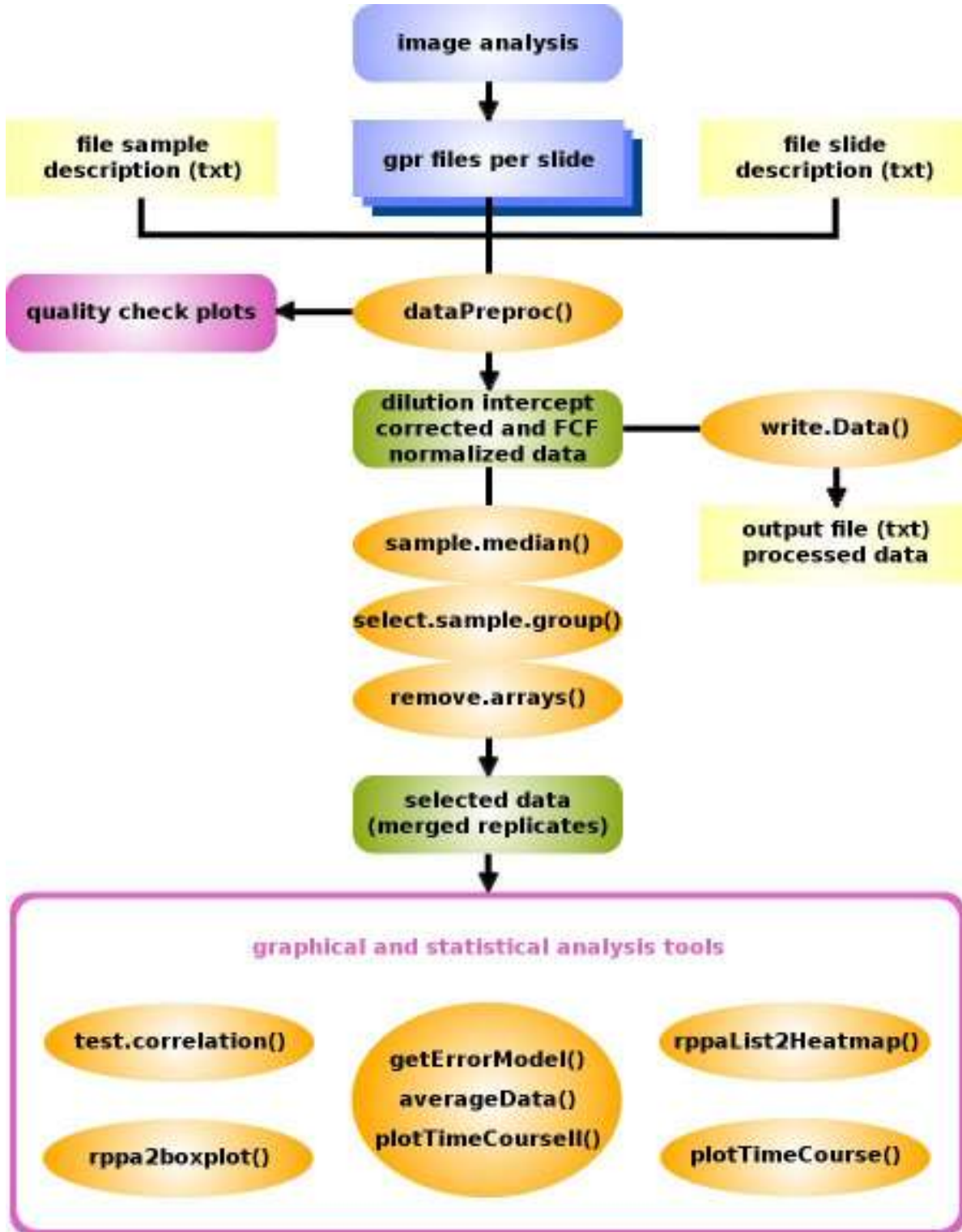


Figure 1: Recommended workflow for the analysis of reverse phase protein array data using the RPPanalyzer package

2.1 Sample description

Every information concerning the samples has to be stored in a tab delimited text file and named `sampledescription.txt` (use spreadsheet software like MS Excel or OOo Spreadsheet to generate the table). The *sampledescription* file contains eight mandatory columns that are required to identify the samples (described in detail below) and optional columns holding any information describing the samples in more detail. To select sample groups for separate analysis it is of advantage to store every type of information in a separate column. To access example files load the RPPanalyzer package:

```
> library(RPPanalyzer)
```

An example *sampledescription* file describing a serially diluted sample set is included.

```
> ## define path to example files
> dataDir <- system.file("extdata",package="RPPanalyzer")
> ## change working directory
> setwd(dataDir)
> ## store example sample description in a variable
> sampledescription <- read.delim("sampledescription.txt")
> ## show sample description header
> head(sampledescription)
```

2.1.1 Columns plate, row and column

These columns describe the location of the samples in the spotting source well plate. The column *plate* contains the number of the source well plate stored as an integer (1, 2, 3, ...). The Column *row* contains capital letters (e.g. A-P) and the column *column* integers (e.g. 1-24) to identify the position within one source well plate.

2.1.2 Column sample_type

The column *sample_type* holds information about the type of the appropriate sample. Entries ‘measurement’ indicate an experimental measurement whereas ‘control’ denotes spots for investigation of antibody binding dynamics. Accordingly ‘neg_control’ is reserved for control spots (e.g. BSA) which can be used to investigate unspecific binding. Finally, ‘blank’ indicates empty spots (e.g. only buffer).

2.1.3 Column sample

Provide an identifier for your samples in this column. It is of advantage to keep this term unique in case of clinical samples, for cell culture experiments put in the name of the cell line and add more columns describing every experimental parameter.

2.1.4 Columns concentration, dilution and dilSeriesID

The column *concentration* provides numeric data with information of the sample concentration. In case of serially diluted samples describe the dilution steps (starting with a 1 for the highest concentration) in column *dilution*. The column *dilSeriesID* contains the values that should be used for dilution intercept correction via `correctDilinterc`, e.g. cell line names. It should have at least two distinct values which must be matchable in ‘control’ and ‘measurement’. NA values denote that control values should not be used.

2.2 Slide description

Write all information describing the slides and arrays in a tab delimited text file and name it `slidedescription.txt`. Seven obligatory columns have to be provided and any optional column can be added.

```
> ## change directory to example files
> dataDir <- system.file("extdata",package="RPPanalyzer")
> setwd(dataDir)
> ## store example slide description in a variable
> slidedescription <- read.delim("slidedescription.txt")
> ## show slide description header
> head(slidedescription)
```

2.2.1 Column gpr

To find the GenePix result files (*gpr* files) in the current folder, the terms stored in the column *gpr* are used as identifier. That means you have to use exactly identical terms for the names of the *gpr* files and in the *gpr* column. If you print multiple arrays on one slide describe the arrays using the same order like on the slide. That means start with describing the uppermost array, than the array below in the next row of the *slidedescription* file and so on.

2.2.2 Columns pad, slide, incubation_run and spotting_run

The column *pad* holds the number of the pad or array on the slides. The column *slide* holds the number of the slide. Arrays that were analyzed in parallel are identified via the *incubation_run* column. Make sure that you have exact one blank array (incubated with 2nd antibodies only) for each incubation run. The column *spotting_run* specifies the arrays that were printed in parallel. You have to provide at least one array with normalizer signals per print run for the normalization method *housekeeping*. In case of normalizing using a protein dye (method *proteinDye*), a whole slide has to be provided.

2.2.3 Columns target and AB_ID

In order to be able to assign the right proteins to the arrays the column *target* holds the protein name and *AB_ID* the corresponding antibody ID. Please use only regular characters (letters, digits, ‘_’, and ‘-’).

2.3 Image analysis result files

So far the software is restricted to read GenePix result files (*gpr* files). For spot identification grid in the image analysis software (here GenePix) use the GenePix array list (*gal* file) that is produced by the spotting device (e.g. Aushon 2470 or ArrayJet).

3 Data pre-processing

Among all possible correction and normalization methods prior to the actual data analysis, we recommend the correction based on serially diluted samples via `correctDilinterc`, combined with normalization according to total protein concentration via FCF. These important pre-processing steps are implemented in the `dataPreproc` function. It imports the raw data, corrects and normalizes it and generates plots for quality checks. The function returns a list with four different elements. The first element contains four raw data elements, i.e. foreground and background expression matrices as well as data frames holding the array and sample description. The second element is analogously built up but with foreground expression data corrected to dilution intercepts via the `correctDilinterc` function. Therefore the default input parameter `correct` must stay 'both'. In case of 'none', the measurements stay as in `rawdat`. In case of 'noFCF', the FCF measurements stay as in `rawdat`.

In case of resulting negative values the absolute minimum plus one is added. The third element is also structured like the first two but holds dilution intercept corrected and FCF normalized foreground data. The last element defines the directory for storing the generated outputs. All output files are stored in a folder labelled with the date of analysis at the input files location. This also holds for the raw data, exported to a text file in table format.

The integrated functions, which can also be applied separately, are explained in the following subsections. For the usage of the `dataPreproc` function we refer to the help page which can be accessed with

```
> ?dataPreproc

> ## change directory to example files
> dataDir <- system.file("extdata",package="RPPanalyzer")
> setwd(dataDir)
> ## pre-process data
> preprocessedData <- dataPreproc(dataDir=dataDir,blocks=12,
+                               spot="aushon",exportNo=2,correct="both")
```

3.1 Read data

Change to the directory where your data files are stored. This can be done using the R working menu (File > change directory...) or by using the command `setwd`. For usage within `dataPreproc`, this is defined in the parameter `dataDir`.

```
> setwd(dataDir)
```

The data analysis starts with reading the data from the current working directory. The argument *blocksperarray* gives the number of blocks that are printed in one array. For usage within `dataPreproc`, this is defined in the parameter *blocks*. This number is used to separate multiple arrays on one slide that are incubated individually. With the argument *spotter* the package takes in account the difference in the column ID which is used to identify the samples. To get information about the manually flagged spots, set the *printFlags* argument to ‘TRUE’ to export these flags to CSV file. For usage within `dataPreproc`, the spotter information has to be provided in the parameter *spot*.

```
> dataDir <- system.file("extdata",package="RPPanalyzer")
> setwd(dataDir)
> rawdata <- read.Data(blocksperarray=12,spotter="aushon",
+                       printFlags=FALSE)
```

After reading the RPPA data, an R-object (list with four elements) is created. The first element holds a matrix with the foreground (expression) intensity data, the second a matrix with background intensities. The columns of the matrix are representing the individual arrays described by the third element of the data list, a data frame holding the array information. The rows of the matrix are described by the fourth element holding the sample information.

3.2 Export data as text file

It is possible to export the RPPA data set as tab delimited text file at any point during data analysis for further inspection using spreadsheet software. The data will be stored in two files, representing the expression and background or expression and error, depending on the analysis step. The rows of the table will be annotated with sample information, the columns with array information.

```
> write.Data(rawdata,FileNameExtension="test_data")
```

The text files will be stored in the current working directory. For usage within `dataPreproc`, the tables are stored in the related analysis output folder.

3.3 Correct for background intensities

To correct for background signals, we recommend to apply the `correctDilinterc` function either directly or within the data pre-processing function `dataPreproc`.

It derives intercepts of dilution series in dependence of *dilSeriesID*, defined in the *sampledescription* file, as well as *slide*, *pad*, *incubation_run* and *spotting_run*, defined in the data frame holding the array information.

To apply this function, a dilution series of a representative sample has to be spotted on each slide in addition to the samples of interest. The latter are defined as ‘measurement’ in the *sampledescription.txt*, while the serially diluted samples are defined as ‘control’. To link a sample of interest to the respective control dilution series, the same identifier has to be entered in the column *dilSeriesID* of the *sampledescription.txt*.

A smoothing spline is used to extrapolate to zero. Nonparametric bootstrap is used to estimate uncertainty of the intercept estimate. Linear models are applied to the intercepts as response variables in dependence of diverse predictors, namely simply a constant, the antibody, antibody + slide or antibody + slide + sample (*dilSeriesID*). Via Analysis of Variances (ANOVA) it is tested which model fits best. The estimated uncertainties of the intercepts are used as weights.

The user should use the provided bar plot ('anovaIntercepts_Output.pdf') of the residual sum of squares (RSS) to choose the model with the smallest RSS, favouring less complexity. For example, if the bars of model 'antibody + slide' and 'antibody + slide + sample' are the smallest and equally high, model 'antibody + slide' should be preferred, as the sample does not provide additional information. The chosen model then is used to predict the intercepts which are subtracted from the original foreground expression. For usage within `dataPreproc`, the model information has to be provided in the parameter `exportNo`. The default is set to four, i.e. the most complex model 'antibody + slide + sample'. The function additionally generates plots of the dilution series and related intercept estimations ('getIntercepts_Output.pdf')

```
> ## import raw data
> fgRaw.tmp <- read.delim("test_dataexpression.txt",stringsAsFactors=FALSE,
+                         row.names=NULL,header=TRUE)
> fgRaw <- read.delim("test_dataexpression.txt",
+                     skip=max(which(fgRaw.tmp[,1]=="")+1,
+                                 stringsAsFactors=FALSE,row.names=NULL,header=TRUE)
> ## remove NAs
> fgNAVec <- which(is.na(fgRaw[, "ID"]))
> if(length(fgNAVec) > 0){
+   fgRaw <- fgRaw[-fgNAVec,]
+ }
> colnames(fgRaw) <- sub("X", "",gsub("\\.", "-", colnames(fgRaw)))
> ## correct data for BG noise
> correctedData <- correctDilinterc(
+   dilseries=fgRaw[which(fgRaw$sample_type=="control" &
+                         !is.na(fgRaw$dilSeriesID)),],
+   arraydesc=rawdata$arraydescription,
+   timeseries=fgRaw[which(fgRaw$sample_type=="measurement"),],exportNo=2)
> ## correct negative values
> if(min(correctedData[,colnames(rawdata$arraydescription)]) < 0){
+   correctedData[,colnames(rawdata$arraydescription)] <-
+     correctedData[,colnames(rawdata$arraydescription)] +
+     abs(min(correctedData[,colnames(rawdata$arraydescription)]))+1
+ }
```

Further information on correction for background intensities is provided in later sections.

3.4 Data normalization with total protein dye

Within `dataPreproc`, the background corrected signal intensities are normalized spot-wise to the total protein concentration using the Fast Green FCF method (Loebke et al., 2007).

In short, replicate slides for one print run in an experiment are stained with the total protein dye ‘Fast Green FCF’ to determine the total protein concentration of each individual lysate spot. In case of multiple arrays on one slide, the normalization is working array (pad) wise. That means each array is normalized by the corresponding array on the normalizer slide. The normalization method `proteinDye` of `normalizeRPPA` requires one normalizer slide per print run which will be identified as ‘protein’ in the `target` column of the `slidedescription` file.

A correction factor is determined for each individual spot that reflects the deviation of the protein concentration determined from the median of all FCF-spots. The target protein specific signal intensities are then corrected for technical variance by division by the correction factors. Afterwards the corrected spot intensities can be multiplied by the median of the corresponding normalizer subarray to scale the data back to the native range. The function `normalizeRPPA` within `dataPreproc` uses the method `proteinDye` and returns normalized values in native scale (instead of \log_2), setting the `vals` attribute to ‘native’. Further information on normalization methods is provided in later sections.

3.5 Quality control plots

Signal validity and antibody dynamics can be checked by comparing the target specific signals to the corresponding blank value of the serially diluted control samples (column `sample_type` in the `sampledescription` file). For this function it is necessary to have one blank array (incubated only with secondary antibodies) for each incubation run (column `incubation_run` in the `slidedescription` file). We included an additional data set containing an experiment with siRNA transfected cell lines to demonstrate the plotting routines.

```
> ## load data set
> dataDir <- system.file("data",package="RPPanalyzer")
> setwd(dataDir)
> load("HKdata.RData")
> data(HKdata)

> plotQC(HKdata,file="control_samples.pdf",arrays2rm=c("protein"))
```

Within `dataPreproc`, this function is applied to the raw signal intensities. Additionally you can plot the blank signals against the target signal of the measurements (column `sample_type` in the `sampledescription` file).

```
> plotMeasurementsQC(HKdata,file="control_measurements.pdf",
+                    arrays2rm=c("protein"))
```

Within `dataPreproc`, this function is applied to the background corrected and FCF normalized signal intensities.

To check the data distribution for each measured target you can generate a PDF file with a quantile-quantile plot. This can be done before and after normalization of the data.

```
> plotqq(HKdata, fileName="qqplot_measurements.pdf")
```

Within `dataPreproc`, this function is applied to the background corrected and FCF normalized signal intensities.

4 Additional correction, quantification and normalization methods

In this section we introduce further data processing functions, apart from the recommended ones integrated in `dataPreproc`. Those may be appropriate in special situations.

4.1 Correct for background intensities

To correct for background signals, you can use all methods from the `backgroundCorrect` functions of the *limma* package (Smyth, 2005) or use the method *admin* which subtracts the local background and adds a small constant value to avoid negative signals.

```
> dataBGcorrected <- correctBG(HKdata, method="normexp")
```

4.2 Quantify concentration

In case of serially diluted samples you have to calculate the (relative) concentration of the samples. You can use either a linear model (function `calcLinear`) or a logistic three parameter model (function `calcLogistic`). We recommend to use the Serial Dilution Curve algorithm (Zhang et al., 2009) which is the most recent development and produces very robust concentration values (function `calcSdc`). Another possibility of quantification is the *SuperCurve* package (Coombes et al., 2009) which can be accessed using the wrapper function `calcSuperCurve`. This function is not part of the `RPPanalyzer` package, but can be found in the appendix of this vignette. To use the `calcSuperCurve` function you have to download and install the package from the MD Anderson Bioinformatics home page (<http://bioinformatics.mdanderson.org/Software/OOMPA/>).

```
> ## To run the serial dilution curve algorithm it is necessary
> ## to aggregate replicate spots first.
> dataDir <- system.file("data", package="RPPanalyzer")
> setwd(dataDir)
> load("ser.dil.samples.RData")
> data(ser.dil.samples)
```

```

> ser.dil_median <- sample.median(ser.dil.samples)
> ## calculate concentration (for the attributes see help pages)
> c_Values <- calcSdc(ser.dil_median,D0=2,sel=c("measurement"),
+                     dilution="dilution")

```

For the arguments of the `calcSdc` function we refer to the help page which can be accessed with

```

> ?calcSdc

```

4.3 Data normalization

Normalization is a crucial step in RPPA data analysis to ensure sample comparability and to yield high quality data. The reference value to normalize RPPA is the total protein amount per spot. There are different possibilities to generate this reference value that will be described in detail below. The following signal normalization steps can be applied directly to background corrected data if the samples are spotted in only one concentration. For serially diluted samples the normalization step is performed on the quantified data. Otherwise the information of the signal dynamics in one dilution series is lost.

4.3.1 Total protein dyes

The most common method to normalize RPPA data is to stain one slide representative for one print run with a total protein dye like Fast Green FCF (also integrated in `dataPreproc`), Sypro Ruby or colloidal Gold (see also Spurrier et al. (2008)).

After calculating the \log_2 intensities, the normalizer value can simply be subtracted from the target signal to obtain the *relative protein expression*. In case of multiple arrays on one slide the normalization is working array wise (pad wise). That means each array is normalized by the corresponding array on the normalizer slide. The normalization method *proteinDye* requires one normalizer slide per print run which will be identified as ‘protein’ in the *target* column of the *slidedescription* file. If you want to obtain values in native scale (instead of \log_2 scale) you have to change the *vals* attribute to ‘native’.

```

> ## load data set
> dataDir <- system.file("data",package="RPPanalyzer")
> setwd(dataDir)
> load("HKdata.RData")
> data(HKdata)
> ## normalize
> norm_values_pd <- normalizeRPPA(HKdata,method="proteinDye",
+                                vals="logged")

```

4.3.2 Housekeeping proteins

Proteins that are expected to be expressed at a constant level, not effected by the experimental conditions, can be used as housekeeping proteins for normalization. This

method is established for quantitative Western blots and can be utilized to normalize RPPA. To obtain the normalizer value, the mean of all arrays identified with the ‘normalizer’ attribute (column *target* in the *slidedescription* file) is calculated within one print run.

```
> norm_values_hk <- normalizeRPPA(HKdata,method="housekeeping",  
+                               normalizer="housekeeping",vals="logged")
```

In case of a fluorescent readout it is possible to incubate antibodies against housekeeping proteins after the target specific antibodies and label them for detection at a different wavelength. Using this approach it is possible to generate the normalizer signal from the same spot as the target specific signal. This enables to correct for spotting imprecisions that could not be identified on just one (or a few) representative slides per print run.

```
> norm_values_hk_sbs <- normalizeRPPA(HKdata,method="spotbyspot",  
+                                    normalizer="housekeeping",vals="logged")
```

4.3.3 Median normalization

Assuming that all proteins measured in the RPPA experiment are reflecting the total protein amount this can be used as a normalizer value. The median value of all protein signals of each spot or sample is calculated and used as normalizer signal.

```
> norm_values_row <- normalizeRPPA(HKdata,method="row")
```

4.3.4 Protein quantification assays

The method *extValue* provides the possibility to utilize protein concentration values determined with protein quantification assays (e.g. Bradford, BCA) as normalizer value. The protein concentration has to be provided in a column in the sample description file and will be accessed with the attribute *useCol*. This method needs very precise spotting device since the value does not include spotting imprecisions.

```
> norm_values_eV <- normalizeRPPA(HKdata,method="extValue",  
+                               useCol="concentration",vals="logged")
```

```
> ## all normalization methods were performed on a sample set that was  
> ## spotted in replicates (not serially diluted).  
> ## In this case you can aggregate the replicate spots after the  
> ## normalization:  
> norm_data <- sample.median(norm_values_pd)
```

5 Array and data selection

To select a sample group of interest for further analysis it is possible to access the samples using any column (attribute *params*) of the *sampledescription* and define the samples of interest (attribute *sel*).

```
> selectedSamples <- select.sample.group(norm_data,  
+                                     params=list("replicate"=c("1")))
```

Furthermore, it is possible to exclude arrays from further analysis which you have identified as not valid or not necessary. They will be identified using the *target* information in the *slidedescription* file.

```
> selectedData <- remove.arrays(selectedSamples,param="target",  
+                               arrays2rm=c("protein","blank","housekeeping"))
```

6 Visualizations

RPPanalyzer provides several standard visualization tools to get an overview of the biological relevance of the data set.

6.1 Time courses

RPPAs allow the measurement of the phosphorylation status of proteins. Therefore they capacitate, in contrast to mRNA based techniques, to investigate signaling pathways in a time resolved manner. Such time course experiments can be visualized with the `plotTimeCourse` or `plotTimeCourseII` function.

The `plotTimeCourse` function will generate a PDF in the current working directory. The argument *tc.identifier* combines the sample attributes which will identify the individual time course experiments whereas the *plot.split* argument will be used to define which time course experiments will be plotted in one graph. The argument *plotformat* defines the way the data will be plotted: ‘rawdata’ will plot the time points connected with dashed lines, ‘splines’ will plot a smoothed spline calculated using the package *gam* (Hastie, 2009). To plot both set *plotformat* to ‘both’.

```
> ## load a time course data set  
> dataDir <- system.file("data",package="RPPanalyzer")  
> setwd(dataDir)  
> load("dataII.RData")  
> data(dataII)  
> ## plot time course data  
> plotTimeCourse(dataII,tc.identifier=c("sample","stimulation",  
+                                     "inhibition","stim_concentration"),  
+               plot.split="experiment",file="Timeplot.pdf",  
+               arrays2rm=c("protein","Blank"),plotformat="spline")
```

The `plotTimeCourseII` function visualizes time courses after data transformation by the `getErrorModel` and `averageData` functions, which estimate noise in the data and average replicates.

```
> ## pre-process the data
> dataDir <- system.file("extdata",package="RPPanalyzer")
> setwd(dataDir)
> res <- dataPreproc(dataDir=dataDir,blocks=12,spot="aushon",exportNo=2)
> ## remove arrays
> normdat_rm <- remove.arrays(res$normdat,param="target",
+                             arrays2rm=c("protein","blank"))
> ## select samples and export data
> sel_sampels_A549 <- select.sample.group(normdat_rm,
+                                         params=list("cell_line"="A549"),
+                                         combine=F)
> write.Data(sel_sampels_A549,FileNameExtension="HGF_sample_data_A549")
> ## read selected data
> dataexpression_1 <- read.table("HGF_sample_data_A549expression.txt")
> ## use getErrorModel function
> dataexpression_2 <- getErrorModel(dataexpression_1,verbose=FALSE)
> ## use averageData function
> dataexpression_3 <- averageData(dataexpression_2,
+                                 scaling=c("slide","replicate"),
+                                 distinguish=c("cell_line","treatment"))
> ## plot time course data
> plotTimeCourseII(dataexpression_3,
+                  filename="timecourse_HGF_sample_data_A549.pdf",
+                  legpos="top",xname="time [min]",yname="signal [a.u.]",
+                  linecolor=c("red","green","blue","black","orange","grey"))
```

6.2 Boxplots

The (differential) expression of proteins between distinct groups can be visualized in boxplots, including related p-values. Therefore, the function `rppa2boxplot` offers two possible testing scenarios. You can compare expression values to a reference group (control), if provided, via a Wilcoxon rank sum test. Otherwise a test on general differences is performed via a Kruskal-Wallis rank sum test. For simple boxplots of groups without any statistical testing, the function `simpleBoxplot` can be applied. A PDF is generated in all cases and saved in the current working directory.

```
> ## load data set
> dataDir <- system.file("data",package="RPPanalyzer")
> setwd(dataDir)
> load("dataIII.RData")
> data(dataIII)
```

```

> ## aggregate replicates
> dataIII_median <- sample.median(dataIII)
> ## draw simple boxplot and generate PDF
> simpleBoxplot(x=dataIII_median,param="rank",
+               orderGrp=c("vx","zx","yzz","rxi"),
+               file="simpleBoxplot.pdf")
> ## draw boxplot, test for (differential) expression in comparison to
> ## control group "vx", and generate PDF
> rppa2boxplot(x=dataIII_median,param="rank",control="vx",
+             orderGrp=c("vx","zx","yzz","rxi"),
+             file="wilcoxonBoxplot.pdf")
> ## draw boxplot, test for general differences in group expressions,
> ## and generate PDF
> rppa2boxplot(x=dataIII_median,param="rank",control=NULL,
+             orderGrp=c("vx","zx","yzz","rxi"),
+             file="kruskalBoxplot.pdf")

```

6.3 Correlation plots

If you want to correlate the protein expression or phosphorylation status to numeric sample attributes, you can use the `test.correlation` function (a wrapper for `cor.test`). Define the correlation method with the `method.cor` argument and the method to correct the p-values for multiple testing in `method.padj`. A PDF will be generated in the current working directory.

```

> ## load data set
> dataDir <- system.file("data",package="RPPanalyzer")
> setwd(dataDir)
> load("dataIII.RData")
> data(dataIII)
> ## normalize data
> n.data <- normalizeRPPA(dataIII,method="row")
> ## aggregate replicates
> cl.data <- sample.median(n.data)
> ## test correlation
> test.correlation(cl.data,param="concentration",method.cor="kendall",
+                 method.padj="BH",file="correlation_plot.pdf")

```

6.4 Heatmaps

A common method to present high dimensional biological data are heatmaps. The RPPanalyzer provides a function to plot heatmaps annotated with any sample attribute in order to check if the sample attribute corresponds to the clustering. Thereby the parameter `sampledescription` defines which information is used for grouping the samples. To ensure a stable and meaningful clustering, removing control arrays and arrays of bad

quality by `remove.arrays` is a recommended preceding step. It is also recommended to apply the `logList` function to the data before plotting to logarithmize (\log_2) the first two RPPA list elements, i.e. foreground and background signal intensities.

```
> ## load data set
> dataDir <- system.file("data",package="RPPanalyzer")
> setwd(dataDir)
> load("dataIII.RData")
> data(dataIII)
> ## aggregate replicates
> dataIII_median <- sample.median(dataIII)
> ## plot heatmap
> rppaList2Heatmap(dataIII_median)
```

7 Appendix

This appendix contains the source code to the Super Curve functionality. If one wants to use this, just paste the functions below into your R-script, load the RPPanalyzer package and go on.

```
#'
#' calcSuperCurve.R
#'
#' Uses the package SuperCurve to perform the quantification
#'

#'
#' calcSuperCurve
#' uses the package SuperCurve for quantification
#'
calcSuperCurve <- function(x,model="cobs", method="nlrq", sample.id=c("sample","sample.
dilution="dilution", block.design, plot=T) {

  # at first use create.ID.col to create unique identifiers
  xi <- create.ID.col(x,sample.id=sample.id)

  # get only the real measurements, don't fit blank spots and buffer
  xi <- select.sample.group(xi,params=list("sample_type"=sel))

  # get the unique IDs
  id <- unique(as.character(xi[[4]]$identifier))

  # create a matrix of the fittet relative expression values
  # since we do a fit for every array the no. of columns is equal to
```



```

# the no. of columns of the raw expression matrix
# the no. of rows is equal to the length of unique(id)
vals <- matrix(NA, ncol=ncol(xi[[1]]), nrow=length(id))
colnames(vals) <- colnames(xi[[1]])
rownames(vals) <- id

# iterate over all arrays and get an individual fit
# this is the main loop
for(i in 1:ncol(xi[[1]])) {

  cat("Fit SuperCurve model for array:", xi[[3]]["target",i], "\n")
  cat("SuperCurve model:", model, "\n")
  cat("SuperCurve fit method:", method, "\n")

  # create an RPPA object
  rppa <- createRPPAObject(xi, i, block.design)

  # create an RPPA Design Object
  rppaDesign <- createRPPDesignObject(rppa, xi, dilution)

  # do the fit
  fit <- RPPAFit(rppa=rppa, design=rppaDesign, measure="Mean.Net",
model=model, method=method)
  #fit <- try(do.call("RPPAFit", args=list(rppa=rppa, design=rppaDesign,
#measure="Mean.net", model=model, method=method)))

  #if(inherits(fit,"try-error")) {

#stop("This function requires the package SuperCurve.
#Please install it
#from 'http://bioinformatics.mdanderson.org/Software/OOMPA/'.")

  #}

  # plot the fit if wanted
  if(plot) {

    main=c(paste("Target:",xi[[3]]["target",i]),
paste("Antibody:", xi[[3]]["AB_ID",i]), paste("Model:", model))

    plot(fit, main=main)

  }
}

```

```

# order the results according to our sampledescription
conc <- as.numeric(fit@concentrations)
m <- match(rownames(vals), names(fit@concentrations), nomatch=0)
if(any(m==0)) {
  stop("For some samples there was no model fit. Something went wrong!")
}

conc <- conc[m]

vals[,i] <- conc

cat("Finished!\n\n")
}

# get only these entries from each dilution series with the highest concentration
# we will need the sampledescription of this
tempDat <- pick.high.conc(xi,sample.id = sample.id)

# pick the lines of the sample description matching to our fitted values
m <- match(rownames(vals), tempDat[[4]]$identifier)
# discarding columns like concentration, dilution, Block, etc.
# these information don't make sense any longer
n <- colnames(tempDat[[4]]) %in% c("concentration", "dilution",
"Block", "Column", "Row")
sampleDesc <- tempDat[[4]][m,!n]

# set rownames of vals to the sample IDs
rownames(vals) <- sampleDesc$ID

# assemble new RPPA data list for return
ret <- tempDat
ret[[1]] <- vals
ret[[2]] <- vals
ret[[4]] <- sampleDesc

# set new names
namesRet <- names(ret)
namesRet[1] <- "expression"
namesRet[2] <- "dummy"
names(ret) <- namesRet

# return the result
return(ret)

```

```

}

#'
#' createRPPAObject
#'
#' creates an RPPA object used in SuperCurve
#' @param x: the RPPA data list
#' @param arrayIdx: the index of the array, the RPPA object is created
#'
createRPPAObject <- function(x, arrayIdx, block.design) {

  # the bg corrected expression values of the samples
  # of the specified array
  Mean.Net <- x[[1]][,arrayIdx]

  # the names of the samples
  Sample <- x[[4]][,"identifier"]

  # now we calculate the main row and the main col
  # we use this terms to determine the rows and cols of the clocks
  # note: the blocks are ordered rowwise, e.g 16 blocks
  # 1 2 3 4
  # 5 6 7 8
  # 9 10 11 12
  # 13 14 15 16

  # first the Main.Row, it can be expressed as
  # (block-1) div (number of cols in the blockdesign) + 1
  Main.Row <- ((x[[4]][,"Block"]-1) %/% block.design[2]) + 1

  # second the columns which can be expressed as
  # (block-1) mod (number of cols in the blockdesign) + 1
  Main.Col <- ((x[[4]][,"Block"]-1) %% block.design[2]) + 1

  # now we can copy our Col and Row information as Sub.Col and Sub.Row
  Sub.Row <- x[[4]][,"Row"]
  Sub.Col <- x[[4]][,"Column"]

  # finally create data.frame
  data <- data.frame(Main.Row=Main.Row, Main.Col=Main.Col,
Sub.Row=Sub.Row, Sub.Col=Sub.Col, Sample=Sample,
Mean.Net=Mean.Net)

```

```

# now define the file and the antibody attribute of the RPPA object
file <- x[[3]][["gpr",arrayIdx]
antibody <- x[[3]][["target",arrayIdx]

# return the new created class RPPA
return(new("RPPA", data=data, file=file, antibody=antibody))

}

```

And here is the Rd source for the calcSuperCurve function:

```

\name{calcSuperCurve}
\alias{calcSuperCurve}

\title{ Calculates the concentration of serial diluted samples}

\description{
Calculates the protein concentration of a serial diluted sample
stored in an RPPA data list using quantification methods implemented
in the SuperCurve package (Coombes et. al, 2009).
}

\usage{
calcSuperCurve(x,model="cobs", method="nlrq",
sample.id=c("sample","sample.n"),sel=c("measurement","control"),
dilution="dilution", block.design, plot=T)
}

\arguments{
\item{x}{RPPA data list with replicates aggregated with median }
\item{model}{The model used by the SuperCurve package. Has to
be either "cobs", "logistic" or "loess".}
\item{method}{The method used for fitting the model. Has to be
"nlrq", "nlrob" or "nls".}
\item{sample.id}{Attributes to identify the samples}
\item{sel}{The sample type that should be calculated. Has to be
"measurements","control", "neg_control",or "blank".}
\item{dilution}{Name of the column in the feature data matrix
describing the dilution steps of the samples.}
\item{block.design}{Vector with two elements. Describes how the blocks
are arranged on the array. The first element specifies the number of
rows, the second the number of columns. The blocks are supposed to be
arranged row by row in the grid.}

```

```

  \item{plot}{Logical. If true, model fits are plotted }
}

\details{
This method is wrapper function to give the user access to the
quantification methods implemented in the SuperCurve package. The
package can be installed from
\url{http://bioinformatics.mdanderson.org/Software/OOMPA/}.
}

\value{

  \item{expression}{matrix with expression values}
  \item{error}{matrix with error values}
  \item{arraydescription}{data frame with feature data}
  \item{sampledescription}{data frame with pheno data}
}

\references{Coombes et. al, 2009, SuperCurve: SuperCurve Package

Hu et al., Bioinformatics 2007,
Non-parametric quantification of protein lysate arrays
}

\author{Stephan Gade <s.gade@dkfz.de> }
\examples{
\dontrun{
library(RPPanalyzer)
data(ser.dil.samples)
}

\dontrun{
predicted.data <- calcSuperCurve(ser.dil.samples, sel=c("measurement"),
block.design=c(1,4))}

}
\keyword{ manip }

```

References

K. R. Coombes, S. Neeley, C. Joy, J. Hu, K. Baggerly, , and P. Roebuck. *SuperCurve: SuperCurve Package*, 2009. R package version 1.3.3.

- T. Hastie. *gam: Generalized Additive Models*, 2009. URL <http://CRAN.R-project.org/package=gam>. R package version 1.01.
- C. Loebke, H. Sueltmann, C. Schmidt, F. Henjes, S. Wiemann, A. Poustka, and U. Korf. Infrared-based protein detection arrays for quantitative proteomics. *Proteomics*, 7(4): 558–64, Feb 2007. doi: 10.1002/pmic.200600757. URL <http://www3.interscience.wiley.com/journal/114123572/abstract>.
- C. P. Paweletz, L. Charboneau, V. E. Bichsel, N. L. Simone, T. Chen, J. W. Gillespie, M. R. Emmert-Buck, M. J. Roth, E. F. P. III, and L. A. Liotta. Reverse phase protein microarrays which capture disease progression show activation of pro-survival pathways at the cancer invasion front. *Oncogene*, 20(16):1981–1989, Apr 2001. doi: 10.1038/sj.onc.1204265. URL <http://dx.doi.org/10.1038/sj.onc.1204265>.
- G. K. Smyth. Limma: linear models for microarray data. In R. Gentleman, V. Carey, S. Dudoit, and W. H. R. Irizarry, editors, *Bioinformatics and Computational Biology Solutions using R and Bioconductor*, pages 397–420. Springer, New York, 2005.
- B. Spurrier, S. Ramalingam, and S. Nishizuka. Reverse-phase protein lysate microarrays for cell signaling analysis. *Nat Protoc*, 3(11):1796–808, Jan 2008. doi: 10.1038/nprot.2008.179. URL <http://www.nature.com/nprot/journal/v3/n11/abs/nprot.2008.179.html>.
- L. Zhang, Q. Wei, L. Mao, W. Liu, G. Mills, and K. Coombes. Serial dilution curve: a new method for analysis of reverse phase protein array data. *Bioinformatics*, Jan 2009. doi: 10.1093/bioinformatics/btn663. URL <http://bioinformatics.oxfordjournals.org/cgi/reprint/btn663v1>.