RQdeltaCT - Relative Quantification of Gene Expression using Delta Ct Methods

Daniel Zalewski daniel.piotr.zalewski@gmail.com

13 September 2025

If you use RQdeltaCT package in your work, please cite this related article:

Zalewski, D., Bogucka-Kocka, A. RQdeltaCT: an open-source R package for relative quantification of gene expression using delta Ct methods. Sci Rep 15, 29762 (2025). https://doi.org/10.1038/s41598-025-11822-0

Table of contents

Introduction

`RQdeltaCT` is an R package developed to perform relative quantification of gene expression using delta Ct methods proposed by Kenneth J. Livak and Thomas D. Schmittgen in Article1 and Article2.

These methods were designed to analyse gene expression data (Ct values) obtained from real-time PCR experiments. The main idea is to normalise gene expression values using endogenous control gene (or genes), present gene expression levels in linear form using the $2^{-(value)}$ transformation, and calculate differences in gene expression levels between groups of samples (or technical replicates of a single sample).

Two main delta Ct methods are used for relative quantification. The choice of the best method depends on the study design. A short description of these methods is provided below; for more details, refer to the articles in the links provided.

1. **2^{-dCt} method.** In this method, Ct values are normalised by the endogenous control gene (often GAPDH, beta-actin, or other) by subtracting the Ct value of the endogenous control in each sample from the Ct value of the gene of interest in the same samples, obtaining delta Ct (dCt) values. Subsequently, the dCt values are transformed using the 2^{-dCt} formula, summarised by means in the compared study groups, and a ratio of means (fold change) is calculated for a study group. This method is useful in scenarios where samples should be analysed as individual data points, e.g., in comparison between patients and healthy subjects. See example no 5. in Article2.
2. **2^{-ddCt} method.** Similar to the 2^{-dCt} method, Ct values are normalised by endogenous control gene, but the obtained delta Ct (dCt) values are not exponentially transformed, but are summarised by means in the compared study groups, and the mean dCt in a control group is subtracted from the mean dCt in a study group, giving the delta delta Ct (ddCt) value. Subsequently, ddCt values are transformed using the 2^{-ddCt} formula to obtain the fold change value (also called the RQ value). This method is useful when a compared groups contain technical rather than biological replicates, e.g. where samples of cell line before adding stimulant are compared to samples of the same cell line after stimulation. See examples no. 1 and 2 in Article2.

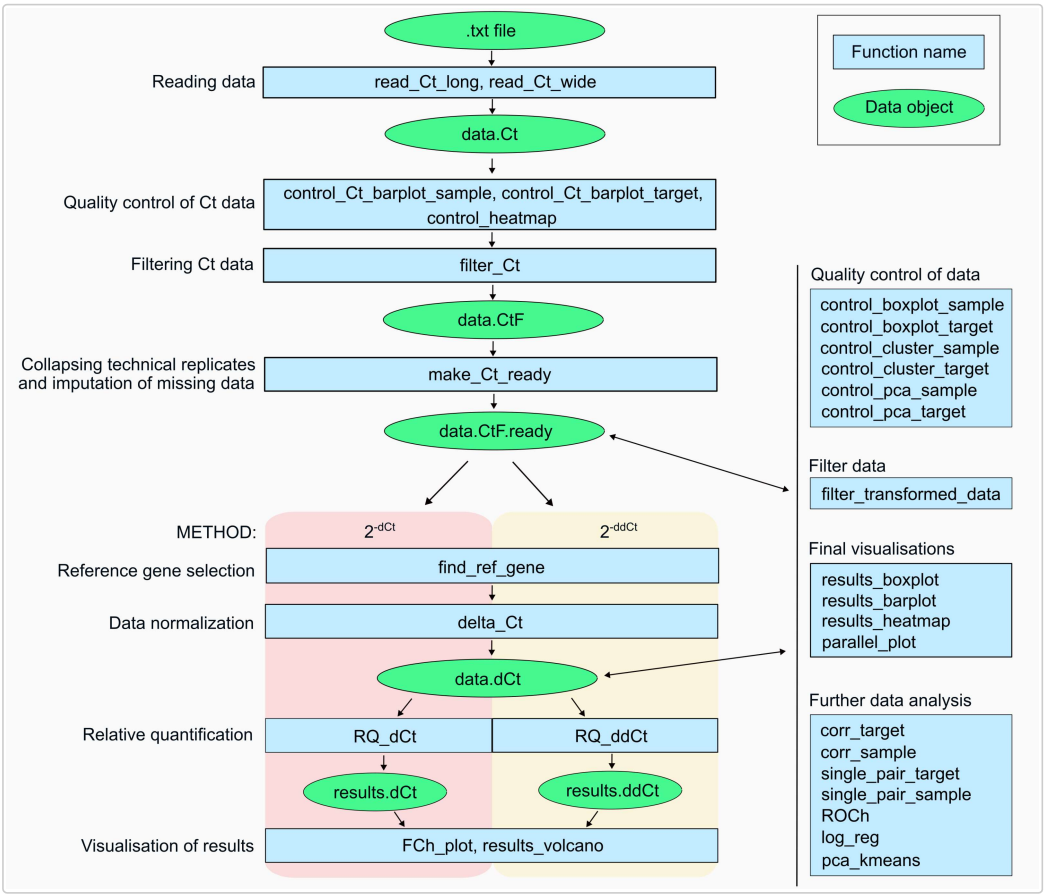
The presented `RQdeltaCT` package includes functions that encompass both of these methods, either for comparison of independent groups of samples or groups with paired samples (pairwise analysis). The selection of a suitable method for analysis is up to the user.

To install and load the `RQdeltaCT` package simply run:

```
remotes::install_github("Donadelnal/RQdeltaCT")
library(RQdeltaCT)
```

The functions developed within the `RQdeltaCT` package are designed to be maximally easy to use, even for users who are beginners to R. The parameters of functions were prepared to sufficiently range all essential tasks and options, and no additional, extensive coding steps are necessary in standard workflow. The package was developed with the intention of being user-friendly and providing an opportunity to perform relative quantification analysis of gene expression using the `RQdeltaCT` package by non-experts in R programming (only basic programming skills are required).

The summary of standard workflow



The entire standard workflow of analysis performed using the `RQdeltaCT` package requires the following external packages:

- `tidyverse` - main package for data processing (`dplyr`, `tidyr`) and visualisation (`ggplot2`),
- `coin` - used to perform the Mann-Whitney U test (`wilcox_test()` function),
- `ctrlGene` - used to calculate gene expression stability score using geNorm algorithm,
- `ggsignif` - used to add significance labels to plots,
- `Hmisc` - used to perform correlation analysis (`rcor()` function),
- `corrplot` - used to visualise results of correlation analysis (`corrplot()` function),
- `ggpmisc` - used to add linear regression results to the plot (`stat_poly_eq()` function),
- `pROC` - used to perform analysis (`roc()` function),
- `oddsratio` - used to compute odds ratio values (`or_glm()` function),
- `GGally` - used to illustrate a pairwise changes in gene expression (`ggparcoord()` function).

All plots created by the functions of the `RQdeltaCT` package can be saved as a .tiff files in the working directory (if `save.to.tiff` parameter is set to `TRUE`). The user can specify the image resolution, dimensions, and name. Firthermore, all generated tables can be saved as .txt files (if `save.to.txt` parameter is set to `TRUE`) with name specified by the user.

This vignette includes instructions and examples of the usage of the main parameters of functions from the `RQdeltaCT` package. The description of all available parameters is provided in the documentation of a particular function.

Data import

Data analysed using the `RQdeltaCT` package should be in tabular form and contain the following information: group names, sample names, gene names, and Ct values. Flag information can also be included for data-filtering purposes. Any other information could exist in the data (the user does not have to remove them), but will not be used for analysis.

Files with such tables can typically be exported from software coupled with PCR devices and used to analyse raw data files generated during real-time PCR experiments. Such files are also returned by external software, such as ExpressionSuite Software, or even R packages, e.g. qpcR

For user convenience, the `RQdeltaCT` package provides two functions that are useful to importing tables in .txt or csv. format:

- `read_Ct_long()` - to import tables with a long-format structure (each information in columns),
- `read_Ct_wide()` - to import tables with a wide-format structure (samples by columns, genes by rows).

NOTE: Imported tables must be free of empty lines.

All datasets used in this chapter are a part of real data used in this article. The tables contain samples divided into two groups: AAA group (patients with abdominal aortic aneurysm) and Control group (subjects without AAA). For each sample, the expression of 19 genes was determined as Ct values, obtained using real-time PCR and TaqMan assays.

Reading long-format data using the `read_Ct_long()` function

Example of a long-format table with a structure suitable for the `read_Ct_long()` function:

Group	Sample	Gene	Ct	Flag
Disease	Disease1	Gene1	25.6	OK
Disease	Disease2	Gene2	32.9	Undetermined
Control	Control1	Gene1	Undetermined	OK
Control	Control2	Gene2	27.5	OK
…	…	…	…	…

For the purpose of presentating the `read_Ct_long()` function, the long-format .txt table (data_Ct_long.txt) located in `RQdeltaCt` package directory will be imported:

```
# Set path to file:
path <- system.file("extdata",
                    "data_Ct_long.txt",
                    package = "RQdeltaCT")

# Import file using path; remember to specify proper separator, decimal character, and number of
         necessary columns:
library(RQdeltaCT)
library(tidyverse)
data.Ct <- read_Ct_long(path = path,
                        sep = "\t",
                        dec = ".",
                        skip = 0,
                        add.column.Flag = TRUE,
                        column.Sample = 1,
                        column.Gene = 2,
                        column.Ct = 5,
                        column.Group = 9,
                        column.Flag = 4)
```

Let's look at the data structure:

```
str(data.Ct)
#> 'data.frame':    1288 obs. of  5 variables:
#>  $ Sample: chr  "AAA1" "AAA10" "AAA12" "AAA13" ...
#>  $ Gene  : chr  "ANGPT1" "ANGPT1" "ANGPT1" "ANGPT1" ...
#>  $ Ct    : chr  "32.563" "34.648" "35.059" "37.135" ...
#>  $ Group : chr  "AAA" "AAA" "AAA" "AAA" ...
#>  $ Flag  : num  1.38 1.35 1.34 1.2 1.39 ...
```

The data were imported properly; however, the Flag variable is numeric, but a character or factor is required. This variable contains a numeric AmpScore parameter, which is often used to evaluate the quality of the amplification curve - curves with AmpScore below 1 are typically considered as low quality and removed from data during analysis. Therefore, the Flag variable can be changed into character by transforming the numeric AmpScore parameter into a binary variable that contains values “OK” and “Undetermined” according to the applied AmpScore criterion:

```
library(tidyverse)
data.Ct <- mutate(data.Ct,
                  Flag = ifelse(Flag < 1, "Undetermined", "OK"))
str(data.Ct)
#> 'data.frame':    1288 obs. of  5 variables:
#>  $ Sample: chr  "AAA1" "AAA10" "AAA12" "AAA13" ...
#>  $ Gene  : chr  "ANGPT1" "ANGPT1" "ANGPT1" "ANGPT1" ...
#>  $ Ct    : chr  "32.563" "34.648" "35.059" "37.135" ...
#>  $ Group : chr  "AAA" "AAA" "AAA" "AAA" ...
#>  $ Flag  : chr  "OK" "OK" "OK" "OK" ...
```

In this transformation, all AmpScore values in the Flag column that are below 1 were changed to “Undetermined”, otherwise to “OK”. The Flag variable now is a character, as required, and the data are ready for further analysis.

Reading wide-format data using the `read_Ct_wide()` function

The `read_Ct_wide()` function was designed to import data with Ct data in the form of a wide-format table (with sample names in the first row and gene names in the first column). Because such a structure does not include names of groups, an additional file containing group names and assigned samples is required. This second file must contain two columns: column named “Sample” with the names of the samples and column named “Group” with the names of the groups assigned to the samples. The names of the samples in this file must be the same as those of the columns in the file with Ct values (the order does not have to be kept).

Example structure of a wide-format table suitable for the `read_Ct_wide()` function:

Gene	Sample1	Sample2	Sample3	…
Gene1	25.4	24.9	25.6	…
Gene2	21.6	22.5	20.8	…
Gene3	33.7	Undetermined	Undetermined	…
Gene4	15.8	16.2	17.5	…
…	…	…	…	…

Example structure of an additional file suitable for the `read_Ct_wide()` function:

Sample	Group
Sample1	Control
Sample2	Control
Sample3	Disease
Sample4	Disease
…	…

In the following example, the `read_Ct_wide()` function was used to import and merge a wide-format file with Ct values (data_Ct_wide.txt) and a file with names of groups (data_design.txt) located in `RQdeltaCt` package directory:

```
# Set paths to required files:
path.Ct.file <- system.file("extdata",
                            "data_Ct_wide.txt",
                            package = "RQdeltaCT")
path.design.file <- system.file("extdata",
                                "data_design.txt",
                                package = "RQdeltaCT")

# Import files:
library(tidyverse)
data.Ct <- read_Ct_wide(path.Ct.file = path.Ct.file,
```

```
                        path.design.file = path.design.file,
                        sep ="\t",
                        dec = ".")

# Look at the structure:
str(data.Ct)
#> tibble [1,216 × 4] (S3: tbl_df/tbl/data.frame)
#>  $ Gene  : chr [1:1216] "ANGPT1" "ANGPT1" "ANGPT1" "ANGPT1" ...
#>  $ Sample: chr [1:1216] "AAA1" "AAA10" "AAA12" "AAA13" ...
#>  $ Ct    : chr [1:1216] "32.563" "34.648" "35.059" "37.135" ...
#>  $ Group : chr [1:1216] "AAA" "AAA" "AAA" "AAA" ...
```

The table imported from the package data object can be directly subjected to further analysis.

Other methods

It can also be a situation in which an imported wide-format table has samples by rows and genes by columns. There is no need to develop a separate function to import files with such a table, the following code can be used to imports such a file:

```
# Import file, be aware of specifying parameters that fit the imported data:
data.Ct.wide <- read.csv(file = "data/data.Ct.wide.vign.txt",
                         header = TRUE,
                         sep = ",")
str(data.Ct.wide)
#> 'data.frame':    64 obs. of  22 variables:
#>  $ X     : int  1 2 3 4 5 6 7 8 9 10 ...
#>  $ Group : chr  "AAA" "AAA" "AAA" "AAA" ...
#>  $ Sample: chr  "AAA1" "AAA10" "AAA12" "AAA13" ...
#>  $ ANGPT1: num  32.6 34.6 35.1 37.1 34 ...
#>  $ ANGPT2: chr  "36.554" "37.262" "36.977" "38.295" ...
#>  $ CCL2  : chr  "31.334" "31.161" "32.077" "33.982" ...
#>  $ CCL5  : num  23.4 22.4 24.3 24.9 24.1 ...
#>  $ CSF2  : chr  "35.608" "33.385" "36.374" "36.997" ...
#>  $ FGF2  : num  32.8 33.3 31.3 35.5 31.9 ...
#>  $ GAPDH : num  21.5 22.9 24.7 24 21.6 ...
#>  $ IL1A  : chr  "35.037" "36.36" "35.946" "36.885" ...
#>  $ IL1B  : num  26.6 27.1 26.7 27.7 26.8 ...
#>  $ IL6   : num  36.7 34.1 35.4 37.1 35.2 ...
#>  $ IL8   : num  28.2 30.2 29.4 32.5 29.6 ...
#>  $ PDGFA : num  28.2 30.5 29.9 33 29.8 ...
#>  $ PDGFB : num  27.9 28.5 28.1 30.8 30.9 ...
#>  $ TGFA  : num  30 31.2 32 32.1 29.4 ...
#>  $ TGFB  : num  21.7 22.5 23.6 24.5 22.7 ...
#>  $ TNF   : num  26.3 27.1 27.6 28.5 27.6 ...
#>  $ VEGFA : num  28.2 28.5 29.6 30.2 26.6 ...
#>  $ VEGFB : chr  "28.263" "27.358" "28.867" "29.951" ...
#>  $ VEGFC : num  32.7 32.9 31 35.9 32 ...

# The imported table is now transformed into a long-format structure.
library(tidyverse)
data.Ct <- data.Ct.wide %>%
             select(-X) %>% # The "X" column is unnecessary and is removed.
             mutate(across(everything(), as.character)) %>% # All variables also are converted to a
         character to unify the class of variables.
             pivot_longer(cols = -c(Group, Sample), names_to = "Gene", values_to = "Ct")
str(data.Ct)
#> tibble [1,216 × 4] (S3: tbl_df/tbl/data.frame)
#>  $ Group : chr [1:1216] "AAA" "AAA" "AAA" "AAA" ...
#>  $ Sample: chr [1:1216] "AAA1" "AAA1" "AAA1" "AAA1" ...
#>  $ Gene  : chr [1:1216] "ANGPT1" "ANGPT2" "CCL2" "CCL5" ...
#>  $ Ct    : chr [1:1216] "32.563" "36.554" "31.334" "23.415" ...
```

NOTE: At this stage, the Ct values do not have to be numeric.

NOTE: Data can also be imported to R using the user's own code, but the final object must be a data frame and contain a table with a column named "Sample" with sample names, column named "Gene" with gene names, column named "Ct" with raw Ct values, column named "Group" with group names,

and optionally column named "Flag" containing flag information (this column should be a class of character or factor).

For package testing purposes, it is also convenient to use the data objects included in the `RQdeltaCT` package, named `data.Ct` (the data with independent groups of samples) and `data.Ct.pairwise` (with dependent groups of samples, can be used for a pairwise analysis):

```
data(data.Ct)
str(data.Ct)
#> 'data.frame':    1288 obs. of  5 variables:
#>  $ Sample: chr  "AAA1" "AAA10" "AAA12" "AAA13" ...
#>  $ Gene  : chr  "ANGPT1" "ANGPT1" "ANGPT1" "ANGPT1" ...
#>  $ Ct    : chr  "32.563" "34.648" "35.059" "37.135" ...
#>  $ Group : chr  "AAA" "AAA" "AAA" "AAA" ...
#>  $ Flag  : chr  "OK" "OK" "OK" "OK" ...

data(data.Ct.pairwise)
str(data.Ct.pairwise)
#> tibble [756 × 4] (S3: tbl_df/tbl/data.frame)
#>  $ Sample: chr [1:756] "Sample01" "Sample01" "Sample01" "Sample01" ...
#>  $ Group : chr [1:756] "After" "After" "After" "After" ...
#>  $ Gene  : chr [1:756] "Gene1" "Gene2" "Gene3" "Gene4" ...
#>  $ Ct    : chr [1:756] "32.563" "36.554" "31.334" "23.415" ...
```

The `data.Ct.pairwise` data are a part of the `data.Ct` data, with a structure transformed to fit a pairwise analysis. Because the data did not come from paired experiments, the sample names, gene names, and group names were encoded. The number of samples in the study group (named "Before") has been equated with the number of the reference group (named "After"). These data are used in this vignette to demonstrate a pairwise approach to the relative quantification of gene expression (see **Part B: A pairwise analysis** chapter).

Part A: The workflow for analysis of independent groups of samples

The `RQdeltaCT` package allows to perform analysis in two variants: a comparison of independent groups of samples and a pairwise comparison of dependent groups of samples. Independent groups contain different samples without inter-group relations. An example of such an analysis is the comparison of patients with a disease vs. healthy controls. In the pairwise approach, the groups contain either the same set of subjects or different, but paired subjects. An example of pairwise analysis is the comparison between the same patients before and after medical intervention. This part of the vignette contains instructions for analysis of independent groups, for the pairwise variant of the workflow refer to **Part B: A pairwise analysis**.

Quality control of raw Ct data

The crucial step of each data analysis is an assessment of the quality and usefulness of the data used to investigate the studied problem. The `RQdeltaCT` package offers two functions for quality control of raw Ct data: `control_Ct_barplot_sample()` (for quality control of samples) and `control_Ct_barplot_gene()` (for quality control of genes). Both functions require specifying quality control criteria to be applied to Ct values, in order to label each Ct value as reliable or not reliable. These functions return numbers of reliable and unreliable Ct values in each sample or each gene, as well as total number of Ct values obtained from each sample and each gene. These results are presented graphically on barplots. The obtained results are useful for inspecting the analysed data in order to identify samples and genes that should be considered to be removed from the data (based on the applied reliability criteria).

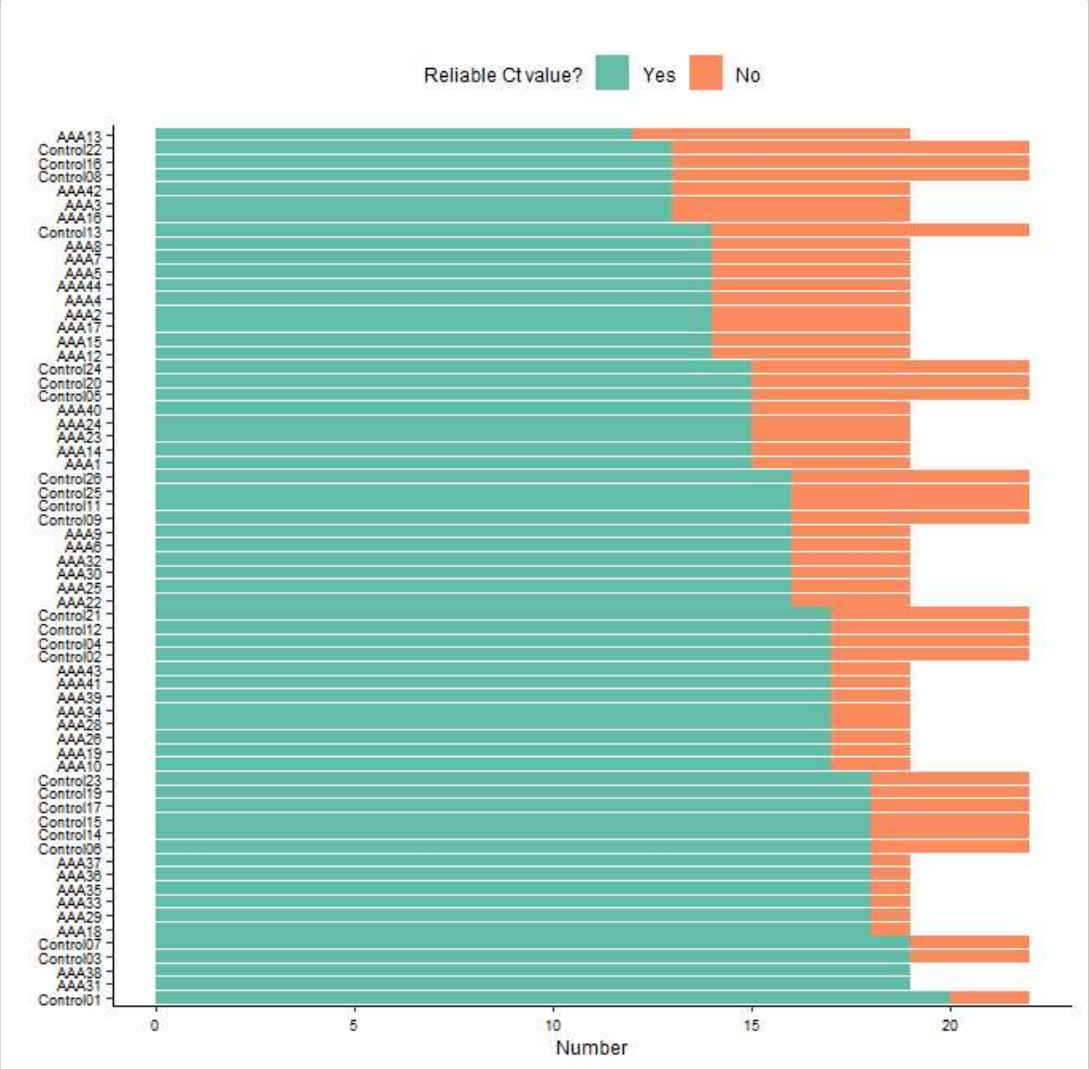
Three selection criteria can be set for these functions:

- a flag used for undetermined Ct values. Default to "Undetermined".
- a maximum of Ct value allowed. Default to 35.
- a flag used in the Flag column for values which are unreliable. Default to "Undetermined".

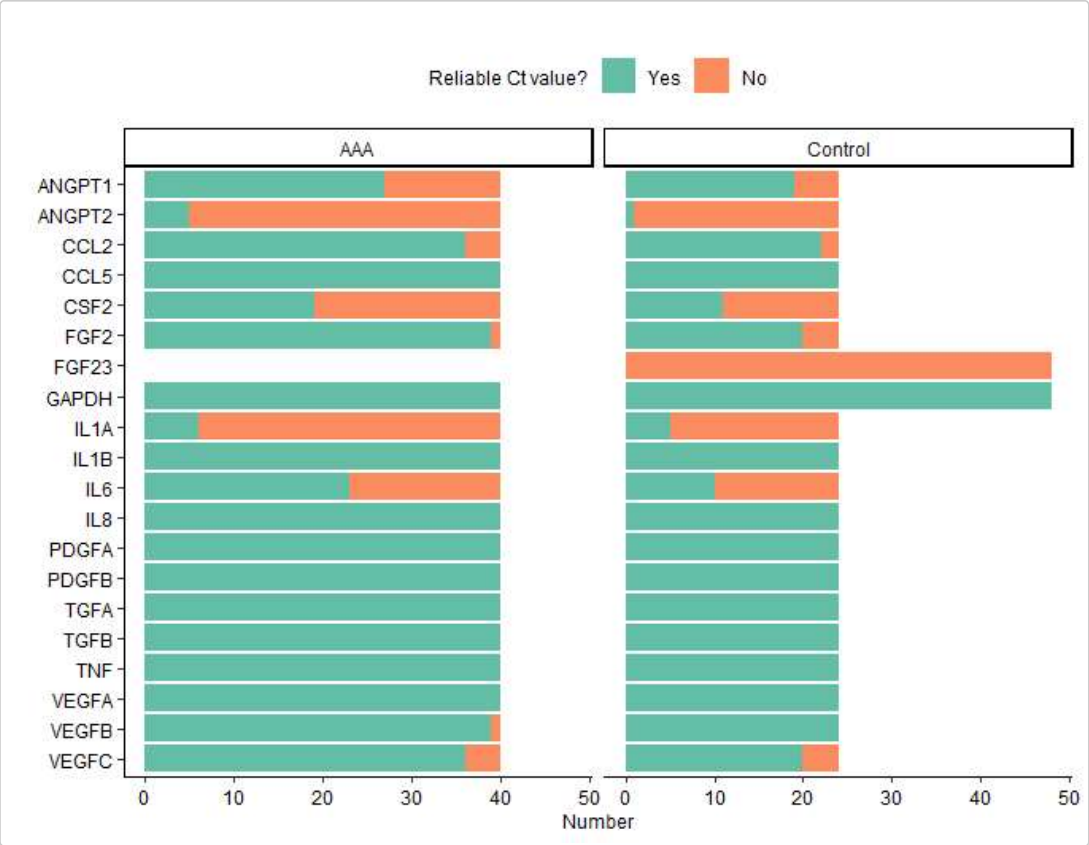
NOTE: These functions do not perform data filtering, but only report numbers of Ct values labelled as reliable or not and present them graphically.

An example of the use of these functions is provided below:

```
sample.Ct.control <- control_Ct_barplot_sample(data = data.Ct,
                                               flag.Ct = "Undetermined",
                                               maxCt = 35,
                                               flag = c("Undetermined"),
                                               axis.title.size = 9,
                                               axis.text.size = 7,
                                               plot.title.size = 9,
                                               legend.title.size = 9,
                                               legend.text.size = 9)
```




```
gene.Ct.control <- control_Ct_barplot_gene(data = data.Ct,
                                           flag.Ct = "Undetermined",
                                           maxCt = 35,
                                           flag = c("Undetermined"),
                                           axis.title.size = 9,
                                           axis.text.size = 9,
                                           plot.title.size = 9,
                                           legend.title.size = 9,
                                           legend.text.size = 9)
```



The created plots are displayed on the graphic device, and short information about the returned tables appears. Returned objects are lists that contain two elements: an object with a plot and a table with numbers of Ct values labelled as reliable (Yes) and unreliable (No), as well as a fraction of unreliable Ct values in each gene. To easily identify samples or genes with high number of unreliable values, tables are sorted to show them at the top. To access the returned tables, the second element of returned objects should be called:

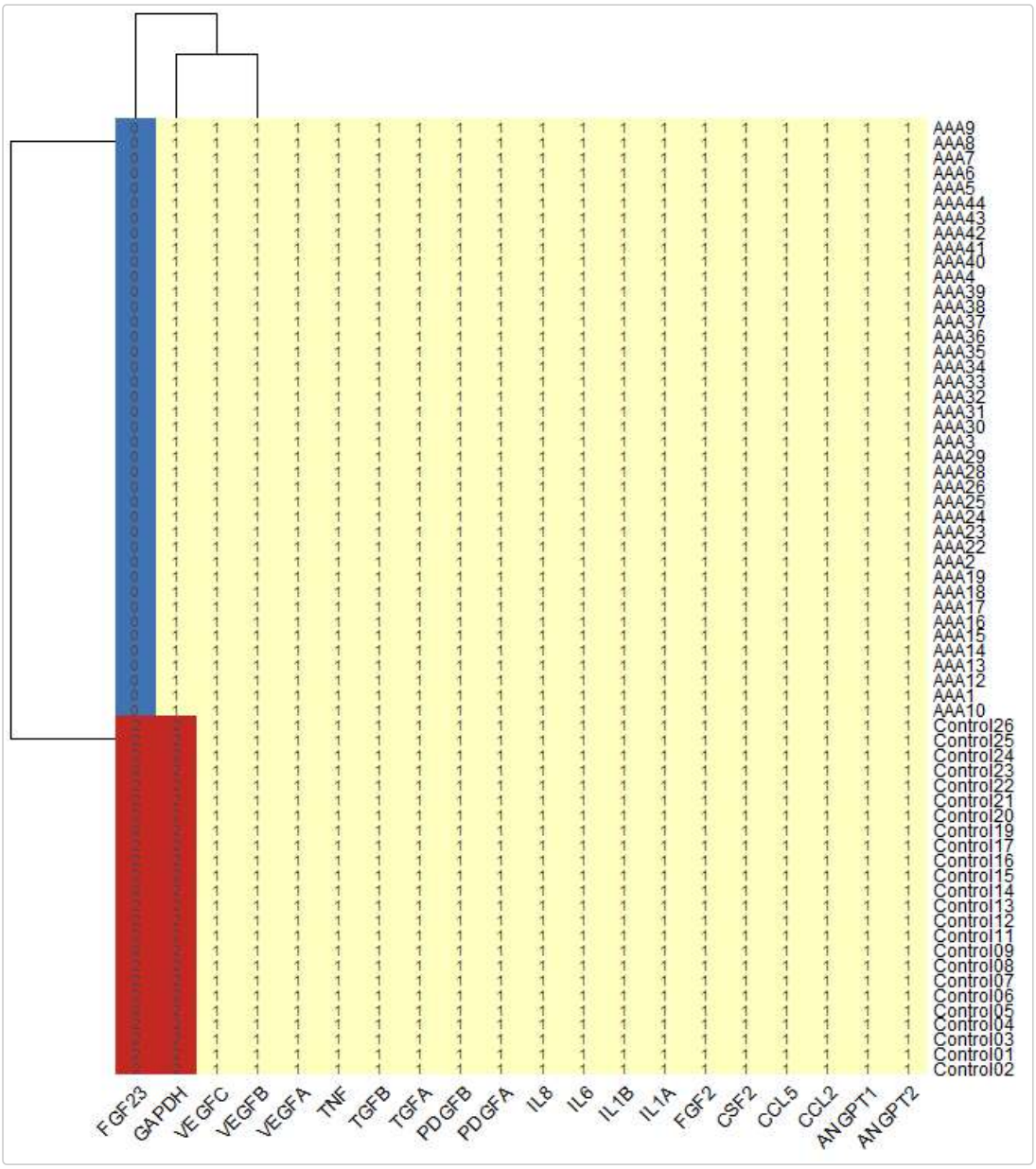
```
head(sample.Ct.control[[2]])
#> # A tibble: 6 × 4
#>   Sample    Not.reliable Reliable Not.reliable.fraction
#>   <fct>            <int>    <int>                <dbl>
#> 1 Control08            9       13                0.409
#> 2 Control16            9       13                0.409
#> 3 Control22            9       13                0.409
#> 4 Control13            8       14                0.364
#> 5 AAA13                7       12                0.368
#> 6 Control05            7       15                0.318
```

```
head(gene.Ct.control[[2]])
#> # A tibble: 6 × 5
#>   Gene   Group   Not.reliable Reliable Not.reliable.fraction
#>   <fct>  <fct>          <int>    <int>                <dbl>
#> 1 FGF23  Control           48        0                1
#> 2 ANGPT2 AAA               35        5                0.875
#> 3 IL1A   AAA               34        6                0.85
#> 4 ANGPT2 Control           23        1                0.958
#> 5 CSF2   AAA               21       19                0.525
#> 6 IL1A   Control           19        5                0.792
```

To gain deeper insight into the number of replicates by genes and samples, the `control_heatmap()` function can be used:

```
library(tidyverse)
library(pheatmap)
data(data.Ct)

# Vector of colors to fill the heatmap can be specified to fit the user's needs:
colors <- c("#4575B4","#FFFFBF","#C32B23")
control_heatmap(data.Ct,
                sel.Gene = "all",
```

```
colors = colors,
show.colnames = TRUE,
show.rownames = TRUE,
fontsize = 9,
fontsize.row = 9,
angle.col = 45)
```



The created plot is displayed on the graphic device (if `save.to.tiff = FALSE`) or saved to .tiff file (if `save.to.tiff = TRUE`).

NOTE: The `control_heatmap()` works only if various numbers of replicates are in the data, otherwise error will appear because the inherited `pheatmap()` function can not deal with the situation where all values are equal.

Visual inspection of returned plots and obtained tables gives a clear image of data quality. The results obtained in the examples show that the AAA group contains more samples than the Control group. Some samples have more Ct values (more technical replicates) than other samples. Furthermore, in all samples, the majority of Ct values are reliable.

Regarding genes, GAPDH and FGF23 were investigated in duplicates in the Control group, while other genes have single Ct values in both groups. Furthermore, FGF23 was analysed only in the Control group and has all values labeled as unreliable; therefore, it is obvious that this gene should be excluded from the analysis. Some other genes also have many unreliable Ct values (e.g. ANGPT2, IL1A) and maybe should be considered to be removed from the data.

In some situations, a unified fraction of unreliable data need to be established and used to make decision which samples or genes should be excluded from the analysis. This can be done using the following code, in which the second element of object returned by the `control_Ct_barplot_sample()` and `control_Ct_barplot_gene()` functions can be used directly, and a vector with samples or genes for which the fraction of unreliable Ct values is higher than a specified threshold is received:

```
# Finding samples with more than half of the unreliable Ct values.
low.quality.samples <- filter(sample.Ct.control[[2]], Not.reliable.fraction > 0.5)$Sample
low.quality.samples <- as.vector(low.quality.samples)
low.quality.samples
#> character(0)


# Finding genes with more than half of the unreliable Ct values in given group.
low.quality.genes <- filter(gene.Ct.control[[2]], Not.reliable.fraction > 0.5)$Gene
low.quality.genes <- unique(as.vector(low.quality.genes))
low.quality.genes
#> [1] "FGF23"  "ANGPT2" "IL1A"   "CSF2"   "IL6"
```

In the above examples, there is no sample with more than half of the unreliable data. Furthermore, this criterion was met by 5 genes (FGF23, ANGPT2, IL1A, CSF2, and IL6); therefore, these genes will be removed from the data in the next step of analysis.

Filtering of raw Ct data

When reliability criteria are finally established for Ct values, and some samples or genes are decided to be excluded from the analysis after quality control of the data, the data with raw Ct values can be filtered using the `filter_Ct()` function.

As a filtering criteria, a flag used for undetermined Ct values, a maximum of Ct threshold, and a flag used in Flag column can be applied. Furthermore, vectors with samples, genes, and groups to be removed can also be specified:

```
# Objects returned from the `low_quality_samples()` and `low_quality_genes()`functions can be used
        directly:
data.CtF <- filter_Ct(data = data.Ct,
                      flag.Ct = "Undetermined",
                      maxCt = 35,
                      flag = c("Undetermined"),
                      remove.Gene = low.quality.genes,
                      remove.Sample = low.quality.samples)

# Check dimensions of data before and after filtering:
dim(data.Ct)
#> [1] 1288    5
dim(data.CtF)
#> [1] 946   5
```

NOTE: If data contain more than two groups, it is good practice to remove groups that are out of comparison; however, a majority of other functions can deal with more groups unless only two groups are indicated to be given in function's parameters. For more details refer to chapter [**Analysis of more than two groups**].

Collapsing technical replicates and imputation of missing data - `make_Ct_ready()` function

In the next step, filtered Ct data can be subjected to collapsing of technical replicates and data imputation by means within groups using the `make_Ct_ready()` function.

The term 'technical replicates' means observations with the same group name, gene name, and sample name. In the scenario when data contain technical replicates but they should not be collapsed, these technical replicates must be distinguished by different sample names, e.g. Sample1_1, Sample1_2, Sample1_3, etc.

The parameter `imput.by.mean.within.groups` can be used to control data imputation. If it is set to `TRUE`, imputation will be done, otherwise missing values will remain in the data. For a better view of the amount of missing values in the data, the information about the number and percentage of missing values is displayed automatically:

```
# Without imputation:
data.CtF.ready <- make_Ct_ready(data = data.CtF,
                                imput.by.mean.within.groups = FALSE)
# A part of the data with missing values:
```

```r
as.data.frame(data.CtF.ready)[19:25,]
#>    Group Sample ANGPT1   CCL2   CCL5   FGF2  GAPDH   IL1B    IL8  PDGFA  PDGFB
#> 19   AAA  AAA36 28.309 31.564 19.434 30.017 20.479 23.996 26.244 25.358 26.934
#> 20   AAA  AAA37 27.741 29.860 19.295 28.410 20.283 25.341 26.172 25.402 26.537
#> 21   AAA  AAA38 28.034 29.755 20.255 29.559 18.314 23.927 23.977 24.790 26.758
#> 22   AAA  AAA39 31.755 28.885 22.475 32.465 23.639 26.675 27.323 28.822 28.353
#> 23   AAA  AAA40 30.513     NA 31.508 31.957 22.866 26.108 26.635 28.001 29.048
#> 24   AAA  AAA41 28.967 33.670 22.517 30.274 22.233 27.056 32.805 28.671 29.105
#> 25   AAA  AAA43 33.471 32.162 22.556 33.499 22.804 25.799 28.884 29.082 27.531
#>      TGFA   TGFB    TNF  VEGFA  VEGFB  VEGFC
#> 19 28.529 20.162 25.334 26.879 26.524 28.720
#> 20 28.891 20.963 26.187 28.048 26.558 29.192
#> 21 26.624 19.997 24.447 26.307 26.054 28.878
#> 22 30.426 22.551 27.124 28.610 28.112 32.293
#> 23 30.730 22.867 27.943 29.338 29.138 32.024
#> 24 28.992 21.658 27.137 27.937 27.208 30.535
#> 25 30.866 22.849 26.406 28.878 27.926 32.749

# With imputation:
data.CtF.ready <- make_Ct_ready(data = data.CtF,
                                imput.by.mean.within.groups = TRUE)
# Missing values were imputed:
as.data.frame(data.CtF.ready)[19:25,]
#>    Group Sample ANGPT1     CCL2   CCL5   FGF2  GAPDH   IL1B    IL8  PDGFA
#> 19   AAA  AAA36 28.309 31.56400 19.434 30.017 20.479 23.996 26.244 25.358
#> 20   AAA  AAA37 27.741 29.86000 19.295 28.410 20.283 25.341 26.172 25.402
#> 21   AAA  AAA38 28.034 29.75500 20.255 29.559 18.314 23.927 23.977 24.790
#> 22   AAA  AAA39 31.755 28.88500 22.475 32.465 23.639 26.675 27.323 28.822
#> 23   AAA  AAA40 30.513 31.58061 31.508 31.957 22.866 26.108 26.635 28.001
#> 24   AAA  AAA41 28.967 33.67000 22.517 30.274 22.233 27.056 32.805 28.671
#> 25   AAA  AAA43 33.471 32.16200 22.556 33.499 22.804 25.799 28.884 29.082
#>     PDGFB   TGFA   TGFB    TNF  VEGFA  VEGFB  VEGFC
#> 19 26.934 28.529 20.162 25.334 26.879 26.524 28.720
#> 20 26.537 28.891 20.963 26.187 28.048 26.558 29.192
#> 21 26.758 26.624 19.997 24.447 26.307 26.054 28.878
#> 22 28.353 30.426 22.551 27.124 28.610 28.112 32.293
#> 23 29.048 30.730 22.867 27.943 29.338 29.138 32.024
#> 24 29.105 28.992 21.658 27.137 27.937 27.208 30.535
#> 25 27.531 30.866 22.849 26.406 28.878 27.926 32.749
```

NOTE: The data imputation process can significantly influence the data; therefore, no default value was set to the `imput.by.mean.within.groups` parameter in order to force the specification by the user. If there are missing data for a certain gene in the entire group, they will not be imputed and will remain NA.

In general, a majority of functions in the `RQdeltaCT` package can deal with missing data; however, some used methods (e.g. PCA) are sensitive to missing data (see PCA analysis section).

NOTE: The `make_Ct_ready()` function should be used even if the collapsing of technical replicates and data imputation is not required, because this function also prepares the data structure to fit to further functions.

Reference gene selection

Ideally, the reference gene should have an identical expression level in all samples, but in many situations it is not possible to achieve this, especially when biological replicates are analysed. Therefore, differences between samples are allowed, but the variance should be as low as possible, and it is also recommended that Ct values should not be very low (below 15) or very high (above 30) Article.

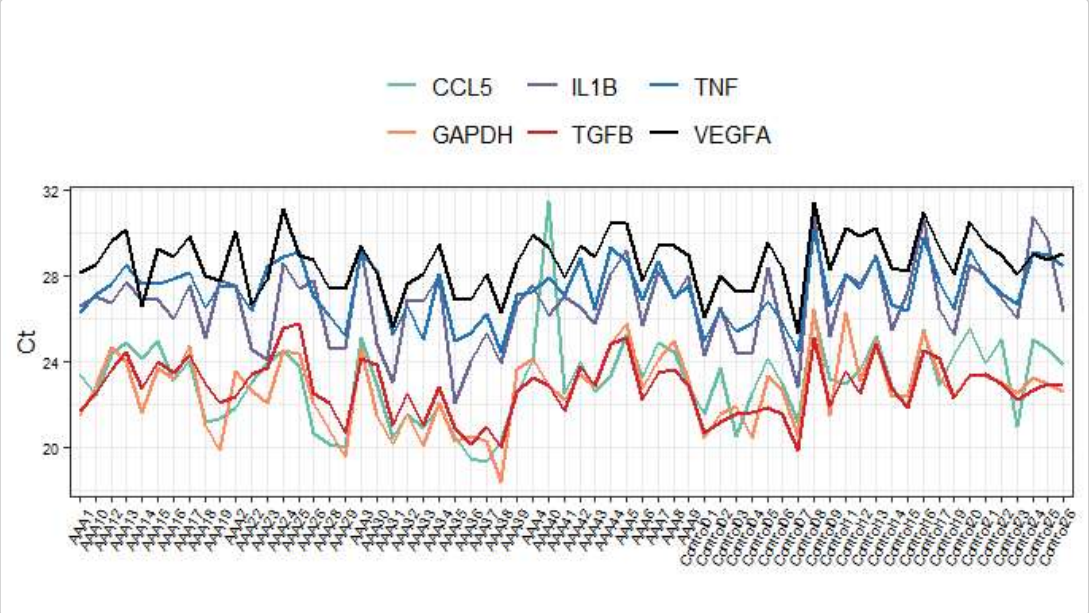
The `RQdeltaCT` package includes `find_ref_gene()` function that can be used to select the best reference gene for normalisation. This function calculates descriptive statistics, such as minimum, maximum, standard deviation, and variance, as well as stability scores calculated using the NormFinder (Article) and geNorm (Article) algorithms. Ct values are also presented on a line plot.

NormFinder scores are computed using the internal `RQdeltaCT::norm_finder()` function working on the code adapted from the original NormFinder code. To calculate NormFinder scores, at least two samples must be present in each group. For the geNorm score, the `geNorm()` function of the `ctrlGene` package is used.

The `find_ref_gene()` function in the `RQdeltaCT` package allows one to choose which of these algorithms should be done by setting the logical parameters: `norm.finder.score` and `genorm.score`. The returned object is a list that

contains two elements: an object with plot and a table with results. In the example below, six genes are tested for suitability to be a reference gene:

```
library(ctrlGene)
# Remember that the number of colors in col parameter should be equal to the number of tested genes:
ref <- find_ref_gene(data = data.CtF.ready,
                     groups = c("AAA","Control"),
                     candidates = c("CCL5", "IL1B","GAPDH","TGFB","TNF", "VEGFA"),
                     col = c("#66c2a5", "#fc8d62","#6A6599", "#D62728", "#1F77B4", "black"),
                     angle = 60,
                     axis.text.size = 7,
                     norm.finder.score = TRUE,
                     genorm.score = TRUE)
```




```
ref[[2]]
#>       Gene    min     max       sd      var NormFinder_score geNorm_score
#> 1     CCL5 19.295 31.5080 1.998908 3.995635             0.33    1.2335325
#> 2    GAPDH 18.314 26.4465 1.724890 2.975244             0.22    0.9001852
#> 3     IL1B 22.043 30.8920 1.894611 3.589551             0.31    1.0511715
#> 4     TGFB 19.801 25.7370 1.359134 1.847245             0.34           NA
#> 5      TNF 24.436 30.1330 1.370355 1.877874             0.19           NA
#> 6    VEGFA 25.323 31.3990 1.329008 1.766262             0.14    0.8127723
#> 7 TGFB-TNF    NA      NA       NA       NA               NA    0.6930721
```

The created plot is displayed on the graphic device. NA values are presented because geNorm method returns a pair of genes with the highest stability.
Among tested genes, GAPDH, TNF, TGFB, and VEGFA seem to have the best characteristics to be a reference gene (they have low variance, low NormFinder and geNorm scores).

Data normalization using reference gene

Data normalization can be performed using `delta_Ct()` function that calculates delta Ct (dCt) values by subtracting Ct values of reference gene (or mean of the Ct values of reference genes, if more than one reference gene is used) from Ct values of gene of interest across all samples. If 2^{-dCt} method is used, `transform` should be set to `TRUE` to tansform dCt values using 2^{-dCt} formula:

```
# For 2^-dCt^ method:
data.dCt.exp <- delta_Ct(data = data.CtF.ready,
                         normalise = TRUE,
                         ref = "GAPDH",
                         transform = TRUE)
```

If 2^{-ddCt} method is used, `transform` should be set to `FALSE` to avoid dCt values transformation that is performed later by the consecutive `RQ_ddCt()` function.

```
# For 2^-ddCt^ method:
data.dCt <- delta_Ct(data = data.CtF.ready,
                     normalise = TRUE,
                     ref = "GAPDH",
                     transform = FALSE)
```

NOTE: In the scenario where unnormalised data should be analysed, the `normalise` parameter should be set to `FALSE`.

Before further processing, non-transformed or transformed dCt data should be subjected to quality control using functions and methods described in [Quality control and filtering of transformed Ct data] section (see below).

Quality control and filtering of normalized Ct data

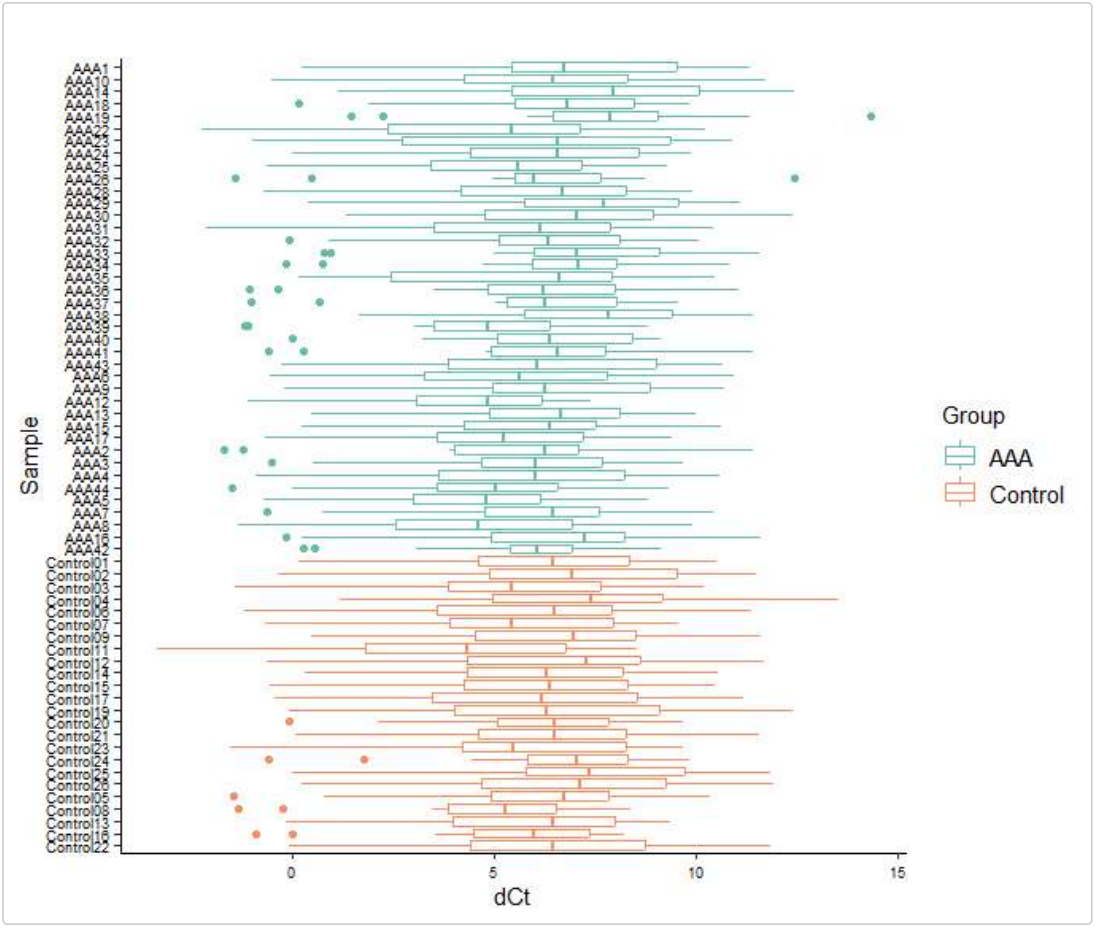
Normalized data intended for relative quantification should be subjected to quality assessment. The main purpose is to identify outlier samples that could introduce bias into the results. The `RQdeltaCT` package offers several functions which facilitate finding outlier samples in data by implementing distribution analysis (`control_boxplot_sample()` function), hierarchical clustering (`control_cluster_sample()` function) and principal component analysis PCA (`control_pca_sample()` function). Furthermore, corresponding functions are also provided for genes to evaluate similarities and differences between expression of analysed genes (`control_boxplot_gene()`, `control_cluster_gene()` and `control_pca_gene()` functions).

The abovementioned data quality control functions are designed to be directly applied to data objects returned from the `make_Ct_ready()` and `delta_Ct()` functions (see The summary of standard workflow).

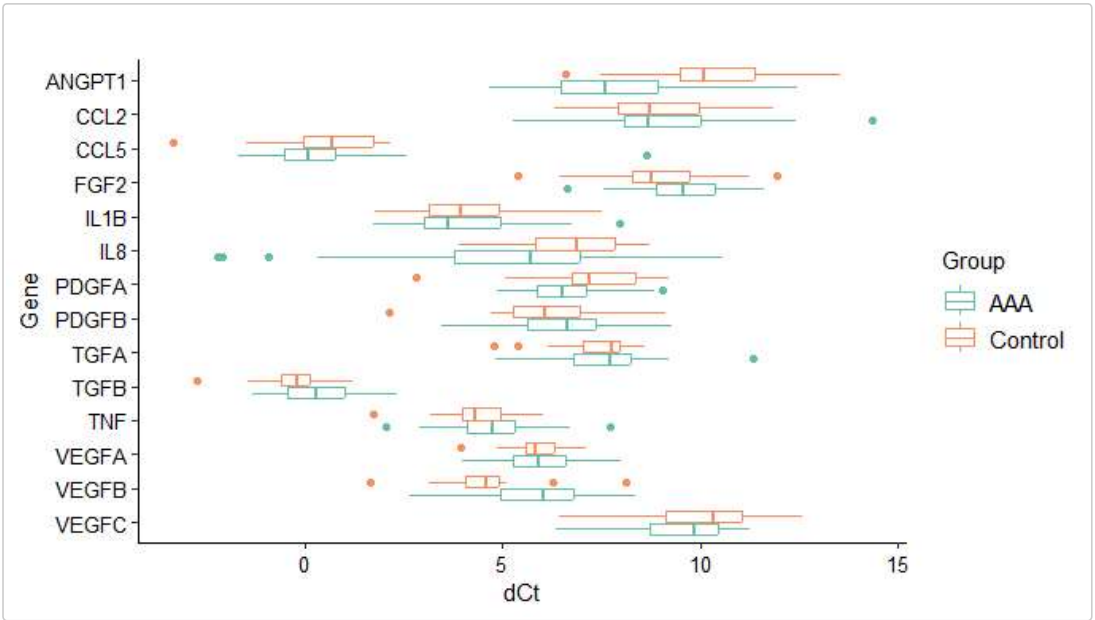
Analysis of data distribution

One of the ways to gain insight into data distribution is drawing boxplot that show several statistics, such as median (labelled inside box), first and third quartiles (ranged by box), extreme point in interquartile range (ranged by whiskers), and more distant data as separated points. In the `RQdeltaCT` package, the `control_boxplot_sample()` and `control_boxplot_gene()` functions are included to draw boxplots that present the distribution of samples and genes, respectively.

```
control_boxplot_sample <- control_boxplot_sample(data = data.dCt,
                                                 y.axis.title = "dCt",
                                                 axis.text.size = 7)
```




```
control_boxplot_gene <- control_boxplot_gene(data = data.dCt,
                                             by.group = TRUE,
                                             y.axis.title = "dCt",
                                             axis.text.size = 10)
```



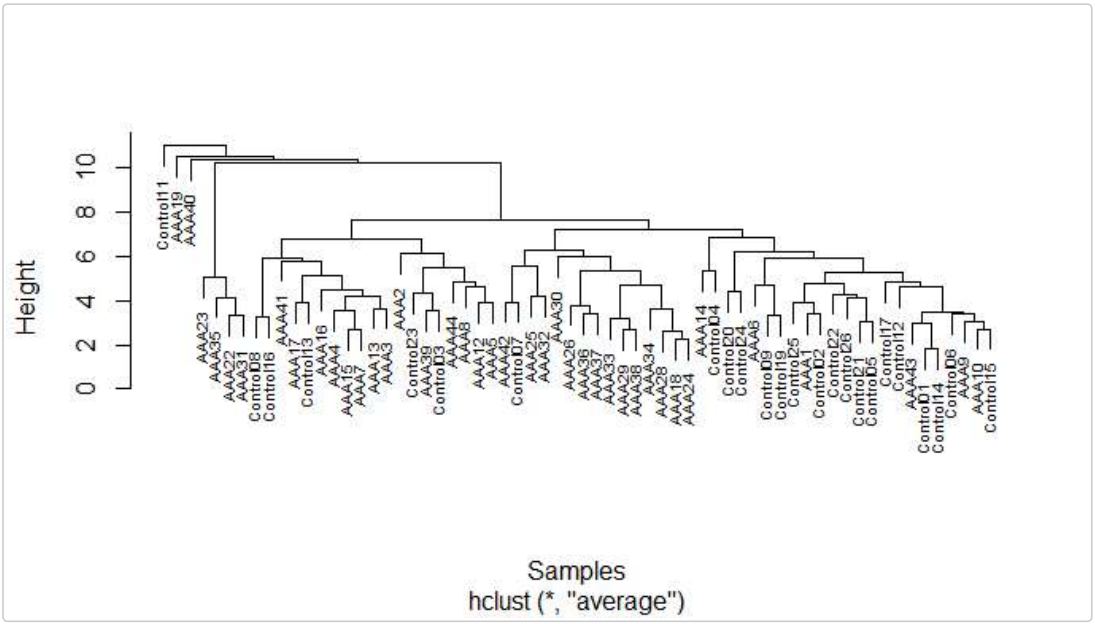
These functions return objects with a plot. The created plots are also displayed on the graphic device.

NOTE: If missing values are present in the data, they will be automatically removed with warning.

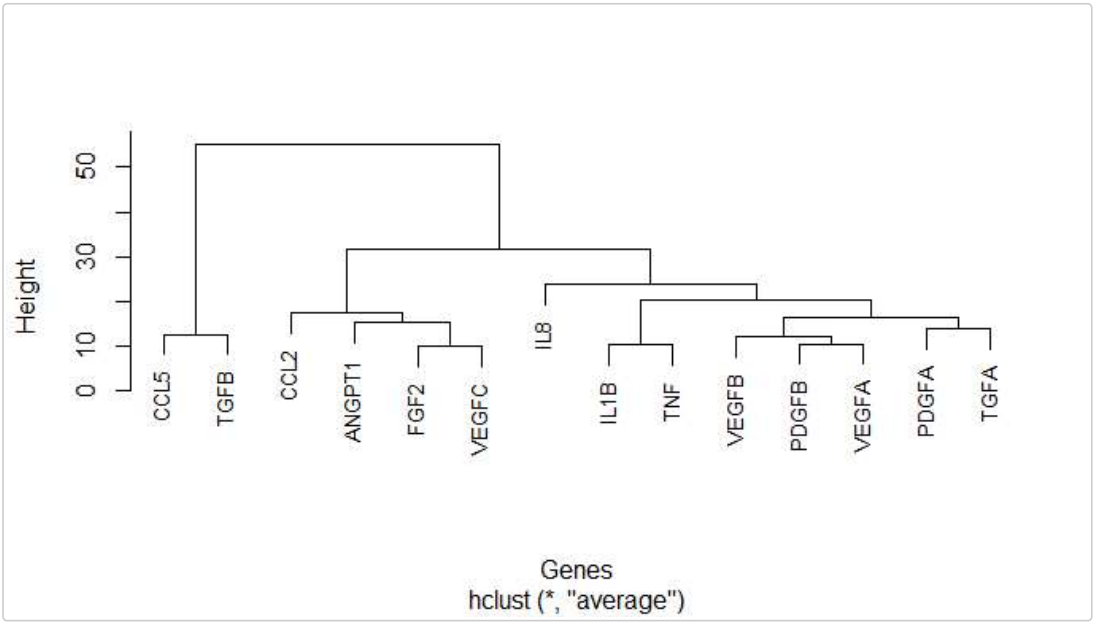
Hierarchical clustering

Hierarchical clustering is a convenient method to investigate similarities between variables. Hierarchical clustering of samples and genes can be done using the `control_cluster_sample()` and `control_cluster_gene()` functions, respectively. These functions allow drawing dendrograms using various methods of distance calculation (e.g. euclidean, canberra) and agglomeration (e.g. complete, average, single). For more details, refer to the functions documentation.

```
control_cluster_sample(data = data.dCt,
                       method.dist = "euclidean",
                       method.clust = "average",
                       label.size = 0.6)
```




```
control_cluster_gene(data = data.dCt,
                     method.dist = "euclidean",
                     method.clust = "average",
                     label.size = 0.8)
```



The created plots are displayed on the graphic device.

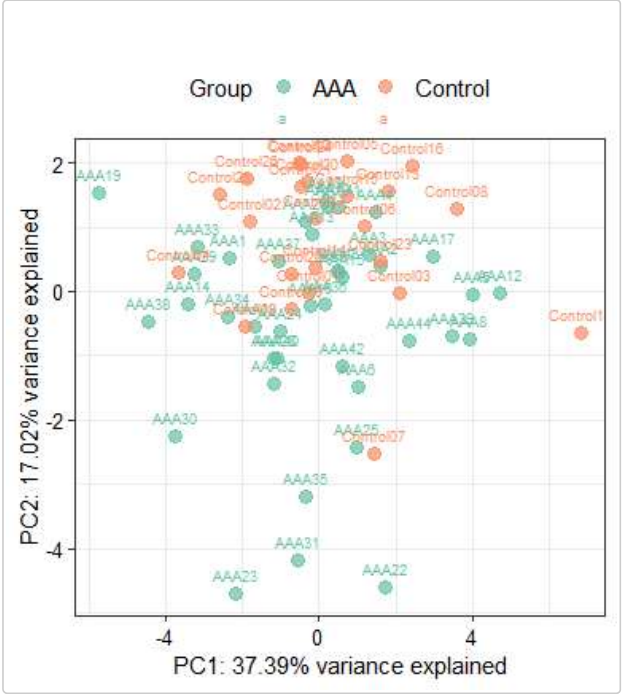
NOTE: Minimum three samples or genes in data is required for clustering analysis.

PCA analysis

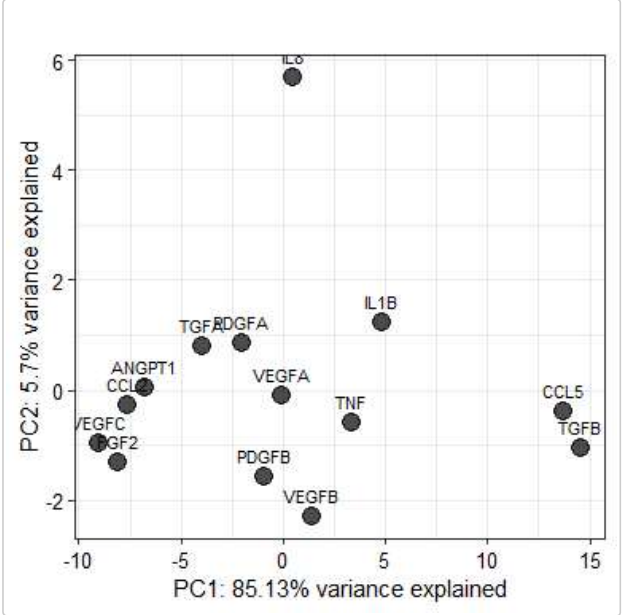
Principal component analysis (PCA) is a data exploratory method, that is commonly used to investigate similarities between variables based on the first principal components that contain the most information about variance. In the `RQdeltaCT` package, the `control_pca_sample()` and `control_pca_gene()` functions are developed to perform PCA analysis for samples and genes, respectively. These functions return objects with a plot. Created plots are also displayed on the graphic device.

```
control.pca.sample <- control_pca_sample(data = data.dCt,
                                         point.size = 3,
```

```
                label.size = 2.5,
                legend.position = "top")
```




```
control.pca.gene <- control_pca_gene(data = data.dCt)
```



NOTE: PCA algorithm can not deal with missing data (NAs); therefore, variables with NA values are removed before analysis. If at least one NA value occurs in all variables in at least one of the compared group, the analysis can not be done. Imputation of missing data will avoid this issue. Also, a minimum of three samples or genes in the data are required for analysis.

Data filtering after quality control

If any sample or gene was decided to be removed from the data, the `filter_transformed_data()` function can be used for filtering. Similarly to quality control functions, this function can be directly applied to data objects returned from the `make_Ct_ready()` and `delta_Ct()` functions (see The summary of standard workflow).

```
data.dCtF <- filter_transformed_data(data = data.dCt,
                                     remove.Sample = c("Control11"))
```

Relative quantification: 2^{-dCt} method

This method is used in studies where samples should be analysed as individual data points, e.g. in analysis of biological replicates. In this method, Ct values are normalised by the endogenous control gene by subtracting the Ct value of the endogenous control from the Ct value of the gene of interest, in the same sample. Obtained delta Ct (dCt) values are subsequently transformed using the 2^{-dCt} formula, summarised by means in the compared study groups, and a ratio of means (fold change) is calculated for the study group (see **Introduction** section).

The whole process can be done using `RQ_dCt()` function, which performs:

- calculation of means (returned in columns with the "_mean" pattern) and standard deviations (returned in columns with the "_sd" pattern) of transformed dCt values of genes analysed in the compared groups.
- normality testing (Shapiro_Wilk test) of transformed dCt values of analysed genes in compared groups and returned p values are stored in columns with the "_norm_p" pattern.
- calculation of fold change values for each gene by dividing the mean of transformed dCt values in the study group by the mean of transformed dCt values in the reference group. Fold change values are returned in the "FCh" column.
- statistical testing of differences in transformed Ct values between the study group and the reference group. Student's t test and Mann-Whitney U test are implemented and the resulting statistics (in column with the "_test_stat" pattern), p values (in column with the "_test_p" pattern), and adjusted p values (in column with the "_test_p_adj" pattern) are returned. The Benjamini-Hochberg method for adjustment of p values is used in default. If compared groups contain less than three samples, normality and statistical tests are not possible to perform (the `do.test` parameter should be set to `FALSE` to avoid error).

NOTE: For this method, delta Ct (dCt) values transformed by the 2^{-dCt} formula using `delta_Ct()` function (called with `transform = TRUE`) should be used.

```
data.dCt.exp <- delta_Ct(data = data.CtF.ready,
                         ref = "GAPDH",
                         transform = TRUE)
library(coin)
results.dCt <- RQ_dCt(data = data.dCt.exp,
                      do.tests = TRUE,
                      group.study = "AAA",
                      group.ref = "Control")

# Obtained table can be sorted by, e.g. p values from the Mann-Whitney U test:
head(as.data.frame(arrange(results.dCt, MW_test_p)))
#>     Gene    AAA_mean Control_mean      AAA_sd  Control_sd   AAA_norm_p
#> 1 ANGPT1 0.007572040  0.001779634 0.007962192 0.002442710 1.175080e-05
#> 2  VEGFB 0.024802219  0.058796934 0.027097216 0.060456391 4.020200e-08
#> 3  PDGFA 0.013343277  0.013651779 0.008790133 0.028406905 2.002333e-03
#> 4   TGFB 0.967490766  1.509777018 0.609112278 1.244618540 4.020298e-03
#> 5    IL8 0.340252219  0.016030664 1.013461602 0.018392782 6.309057e-12
#> 6   FGF2 0.001911581  0.003424749 0.001866461 0.004925526 3.737597e-07
#>   Control_norm_p        FCh     log10FCh     t_test_p t_test_stat    MW_test_p
#> 1   3.435812e-06  4.2548290  0.628882107 8.483229e-05  4.27774491 5.136855e-05
#> 2   3.400562e-07  0.4218284 -0.374864146 1.450713e-02 -2.60232741 5.449915e-05
#> 3   5.964156e-09  0.9774021 -0.009926733 9.591380e-01 -0.05173793 1.009811e-02
#> 4   1.086815e-06  0.6408170 -0.193265981 5.517804e-02 -1.99590948 1.255492e-02
#> 5   1.391742e-05 21.2250864  1.326849466 4.998438e-02  2.02276492 1.410617e-02
#> 6   1.322123e-07  0.5581668 -0.253236035 1.602119e-01 -1.44408971 6.717362e-02
#>   MW_test_stat t_test_p_adj MW_test_p_adj
#> 1     4.049311  0.001187652  0.0003814941
#> 2    -4.035444  0.101549928  0.0003814941
#> 3     2.572452  0.974195838  0.0394972722
#> 4    -2.496151  0.193123123  0.0394972722
#> 5     2.454548  0.193123123  0.0394972722
#> 6    -1.830511  0.448593384  0.1567384376
```

Relative quantification: 2^{-ddCt} method

Similarly to the 2^{-dCt} method, in the 2^{-ddCt} method Ct values are normalised by the endogenous control gene, obtaining dCt values. Subsequently, dCt values are summarised by means in the compared groups, and for each gene, the obtained mean in the control group is subtracted from the mean in the study group, giving the delta

delta Ct (ddCt) value. Finally, the ddCt values are transformed using the 2^{-ddCt} formula to obtain the fold change values. This method is recommended for analysis of technical replicates (see the **Introduction** section).

The whole process can be done using `RQ_ddCt()` function, which performs:

- calculation of means (returned in columns with the "_mean" pattern) and standard deviations (returned in columns with the "_sd" pattern) of delta Ct values of the analyzed genes in the compared groups.
- normality testing (Shapiro_Wilk test) of delta Ct values of the analyzed genes in the compared groups and returned p values are stored in columns with the "_norm_p" pattern.
- calculation of differences in the mean delta Ct values of genes between compared groups, returned in "ddCt" column,
- calculation of fold change values (returned in "FCh" column) for each gene by transforming the ddCt values using the 2^{-ddCt} formula.
- statistical testing of differences between the compared groups. Student's t test and Mann-Whitney U test are implemented and the resulted statistics (in column with the "_test_stat" pattern), p values (in column with the "_test_p" pattern) and adjusted p values (in column with the "_test_p_adj" pattern) are returned. The Benjamini-Hochberg method for adjustment of p values is used in default. If compared groups contain less than three samples, normality and statistical tests are not possible to perform and `do.test` parameter should be set to `FALSE` to avoid error.

NOTE: For this method, not transformed delta Ct (dCt) values (obtained from `delta_Ct()` function with `transform = FALSE`) should be used.

```
data.dCt <- delta_Ct(data = data.CtF.ready,
                     ref = "GAPDH",
                     transform = FALSE)
library(coin)
results.ddCt <- RQ_ddCt(data = data.dCt,
                        group.study = "AAA",
                        group.ref = "Control",
                        do.tests = TRUE)

# Obtained table can be sorted by, e.g. p values from the Mann-Whitney U test:
head(as.data.frame(arrange(results.ddCt, MW_test_p)))
#>     Gene AAA_mean Control_mean    AAA_sd Control_sd AAA_norm_p Control_norm_p
#> 1 ANGPT1 7.959426    10.117002 1.8871311  1.6881153 0.17446884    0.426642224
#> 2  VEGFB 5.938756     4.522833 1.3467583  1.1662848 0.56682258    0.007240964
#> 3  PDGFA 6.565200     7.216583 1.0627016  1.4577193 0.17567779    0.038235161
#> 4   TGFB 0.338025    -0.329125 0.9570775  0.8145069 0.57399623    0.213259767
#> 5    IL8 5.058075     6.675667 2.9513180  1.4070518 0.04414803    0.172997230
#> 6   FGF2 9.536064     8.932742 1.2052427  1.4167152 0.61294050    0.614472418
#>         ddCt       FCh   log10FCh     t_test_p t_test_stat    MW_test_p
#> 1 -2.1575763 4.4616467  0.6494952 1.690067e-05   -4.733411 5.136855e-05
#> 2  1.4159231 0.3747699 -0.4262353 4.581993e-05    4.432997 5.449915e-05
#> 3 -0.6513833 1.5706735  0.1960859 6.426170e-02   -1.906189 1.009811e-02
#> 4  0.6671500 0.6297495 -0.2008322 4.445141e-03    2.967530 1.255492e-02
#> 5 -1.6175917 3.0686235  0.4869436 4.508310e-03   -2.952083 1.410617e-02
#> 6  0.6033224 0.6582363 -0.1816182 8.872007e-02    1.742051 6.717362e-02
#>   MW_test_stat t_test_p_adj MW_test_p_adj
#> 1    -4.049311 0.0002366094  0.0003814941
#> 2     4.035444 0.0003207395  0.0003814941
#> 3    -2.572452 0.1799327522  0.0394972722
#> 4     2.496151 0.0157790854  0.0394972722
#> 5    -2.454548 0.0157790854  0.0394972722
#> 6     1.830511 0.2070134948  0.1567384376
```

Final visualisations

For visualisation of final results, the following functions can be used:

- `FCh_plot()` that allows to illustrate fold change values of genes,
- `results_volcano()` that allows to create volcano plot presenting the arrangement of fold change values and p values,
- `results_barplot()` that show mean and standard deviation values of genes across the compared groups,
- `results_boxplot()` that illustrate the data distribution of genes across the compared groups,
- `results_heatmap()` that allows to create heatmap with hierarchical clustering,

All these functions can be run on all data or on selected genes (see `sel.Gene` parameter). These functions have a large number of parameters, and the user should familiarise with all of them to properly adjust created plots to the user needs.

NOTE: The functions `FCh_plot()`, `results_barplot()`, and `results_boxplot()` also allow to add customised statistical significance labels to plots based on the incorporated `ggsignif` package functionalities. If statistical significance labels should be added to the plot, a vector with labels (e.g., "ns", "*","p = 0.03") should be provided in the `signif.labels` parameter. There are two important points that must be taking into account when preparing this vector:

- The order of labels should correspond to the order of genes presented on the plot, not the order of genes in the data, which can be different.
- In this vector, due to restrictions of the `ggsignif` package, all values must be different (the same values are not allowed). Thus, if the same labels are needed, they should be distinguishable by adding symmetrically a different number of white spaces (see the examples below).

The `FCh_plot()` function

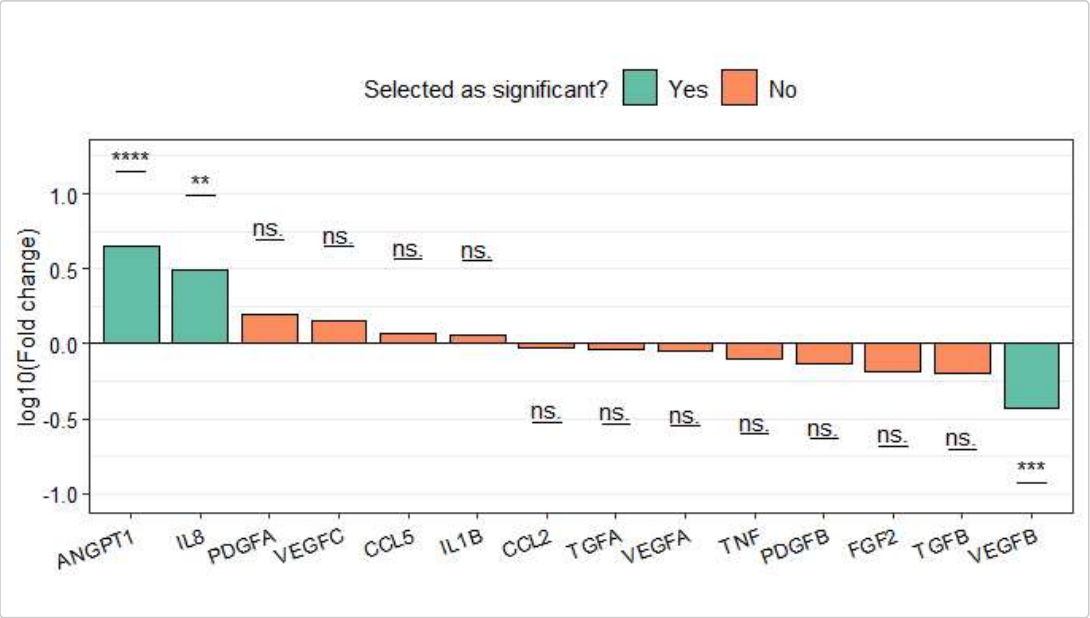
This function creates a barplot that illustrates fold change values obtained from the analysis, together with an indication of statistical significance. Data returned from the `RQ_dCt()` and `RQ_ddCt()` functions can be directly applied to this function (see The summary of standard workflow).

On the barplot, bars of significant genes are distinguished by colors and/or significance labels. The significance of genes can be established by two criteria: p values and (optionally) fold change values. Thresholds for both criteria can be specified. The `FCh_plot()` function offers various options of which p values are used on the plot:

- p values from the Student's t test (if `mode = "t"`).
- p values from the Mann-Whitney U test (if `mode = "mw"`).
- p values depend on the normality of data (if `mode = "depends"`). If the data in both compared groups were considered derived from normal distribution (p value of Shapiro_Wilk test > 0.05) - p values of Student's t test will be used, otherwise p values of Mann-Whitney U test will be used.
- external p values provided by the user (if `mode = "user"`). If the user intends to use the p values obtained from the other statistical test, the `mode` parameter should be set to "user". In this scenario, before running the `FCh_plot()` function, the user should prepare a data.frame object named "user" containing two columns: the first column with gene names and the second column with p values (see example below).

The created plot is displayed on the graphic device. The returned object is a lists that contain two elements: an object with plot and a table with results.

```
# Variant with p values depending on the normality of the data:
library(ggsignif)
# Specifying vector with significance labels:
signif.labels <- c("****",
                   "**",
                   "ns.",
                   " ns. ",
                   "  ns.  ",
                   "   ns.   ",
                   "    ns.    ",
                   "     ns.     ",
                   "      ns.      ",
                   "       ns.       ",
                   "        ns.        ",
                   "         ns.         ",
                   "          ns.          ",
                   "***")
# Genes with p < 0.05 and 2-fold changed expression between compared groups are considered significant:
FCh.plot <- FCh_plot(data = results.ddCt,
                     use.p = TRUE,
                     mode = "depends",
                     p.threshold = 0.05,
                     use.FCh = TRUE,
                     FCh.threshold = 2,
                     signif.show = TRUE,
                     signif.labels = signif.labels,
                     angle = 20)
```

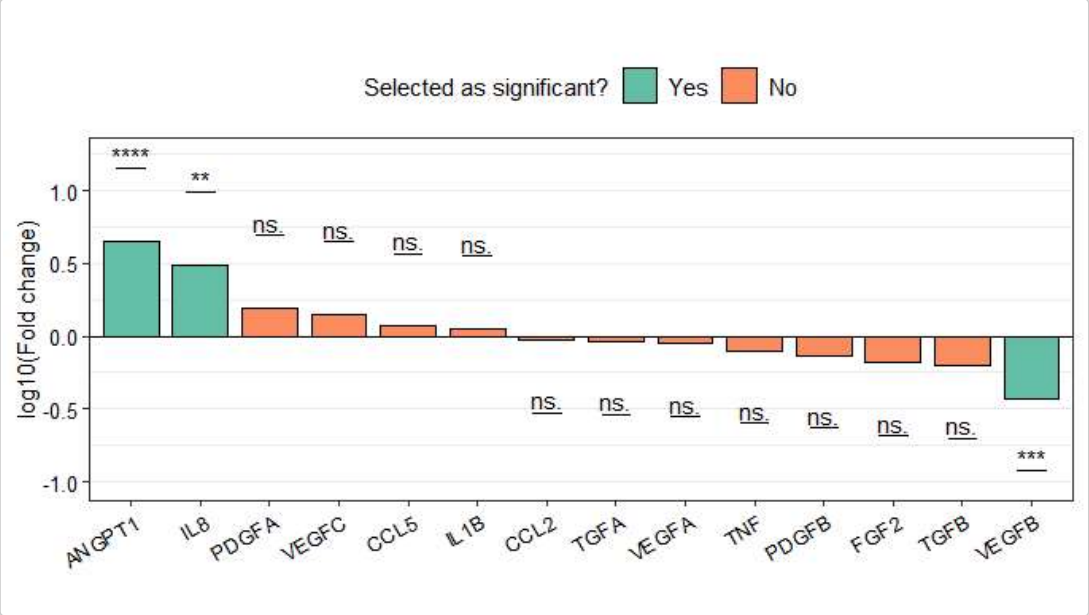



```
# Access the table with results:
head(as.data.frame(FCh.plot[[2]]))
#>     Gene AAA_mean Control_mean   AAA_sd Control_sd   AAA_norm_p Control_norm_p
#> 1 ANGPT1 7.959426    10.117002 1.887131   1.688115 1.744688e-01     0.42664222
#> 2    IL8 5.058075     6.675667 2.951318   1.407052 4.414803e-02     0.17299723
#> 3  PDGFA 6.565200     7.216583 1.062702   1.457719 1.756778e-01     0.03823516
#> 4  VEGFC 9.481083     9.983942 1.143782   1.571200 2.041532e-02     0.68169170
#> 5   CCL5 0.360150     0.584125 1.670593   1.358816 2.060155e-07     0.03149060
#> 6   IL1B 3.922625     4.101125 1.381818   1.336964 8.555529e-02     0.47231242
#>         ddCt      FCh   log10FCh     t_test_p t_test_stat    MW_test_p
#> 1 -2.1575763 4.461647 0.64949517 1.690067e-05  -4.7334112 5.136855e-05
#> 2 -1.6175917 3.068624 0.48694361 4.508310e-03  -2.9520830 1.410617e-02
#> 3 -0.6513833 1.570674 0.19608592 6.426170e-02  -1.9061886 1.009811e-02
#> 4 -0.5028583 1.417018 0.15137544 1.801127e-01  -1.3657411 1.271530e-01
#> 5 -0.2239750 1.167947 0.06742319 5.610445e-01  -0.5847601 8.551052e-02
#> 6 -0.1785000 1.131707 0.05373385 6.118866e-01  -0.5105974 6.572160e-01
#>   MW_test_stat t_test_p_adj MW_test_p_adj test.for.comparison       p.used
#> 1   -4.0493114 0.0002366094  0.0003814941    t.student's.test 1.690067e-05
#> 2   -2.4545484 0.0157790854  0.0394972722   Mann-Whitney.test 1.410617e-02
#> 3   -2.5724516 0.1799327522  0.0394972722   Mann-Whitney.test 1.009811e-02
#> 4   -1.5254255 0.3527853502  0.2225177371   Mann-Whitney.test 1.271530e-01
#> 5   -1.7195706 0.7138676822  0.1710210453   Mann-Whitney.test 8.551052e-02
#> 6   -0.4437602 0.7138676822  0.7902902022    t.student's.test 6.118866e-01
#>   Selected as significant?
#> 1                      Yes
#> 2                      Yes
#> 3                       No
#> 4                       No
#> 5                       No
#> 6                       No
```

A variant with user p values - the used p values are calculated using the `stats::wilcox.test()` function:

```
user <- data.dCt %>%
          pivot_longer(cols = -c(Group, Sample),
                       names_to = "Gene",
                       values_to = "dCt") %>%
          group_by(Gene) %>%
          summarise(MW_test_p = wilcox.test(dCt ~ Group)$p.value,
                    .groups = "keep")
# The stats::wilcox.test() functions is limited to cases without ties;
# therefore, a warning "cannot compute exact p-value with ties" will appear when ties occur.

FCh.plot <- FCh_plot(data = results.ddCt,
                     use.p = TRUE,
                     mode = "user",
                     p.threshold = 0.05,
```

```
                use.FCh = TRUE,
                FCh.threshold = 2,
                signif.show = TRUE,
                signif.labels = signif.labels,
                angle = 30)
```




```
# Access the table with results (p.used column was changed):
head(as.data.frame(FCh.plot[[2]]))
#>     Gene AAA_mean Control_mean   AAA_sd Control_sd   AAA_norm_p Control_norm_p
#> 1 ANGPT1 7.959426    10.117002 1.887131   1.688115 1.744688e-01     0.42664222
#> 2    IL8 5.058075     6.675667 2.951318   1.407052 4.414803e-02     0.17299723
#> 3  PDGFA 6.565200     7.216583 1.062702   1.457719 1.756778e-01     0.03823516
#> 4  VEGFC 9.481083     9.983942 1.143782   1.571200 2.041532e-02     0.68169170
#> 5   CCL5 0.360150     0.584125 1.670593   1.358816 2.060155e-07     0.03149060
#> 6   IL1B 3.922625     4.101125 1.381818   1.336964 8.555529e-02     0.47231242
#>         ddCt      FCh   log10FCh     t_test_p t_test_stat    MW_test_p
#> 1 -2.1575763 4.461647 0.64949517 1.690067e-05  -4.7334112 5.136855e-05
#> 2 -1.6175917 3.068624 0.48694361 4.508310e-03  -2.9520830 1.410617e-02
#> 3 -0.6513833 1.570674 0.19608592 6.426170e-02  -1.9061886 1.009811e-02
#> 4 -0.5028583 1.417018 0.15137544 1.801127e-01  -1.3657411 1.271530e-01
#> 5 -0.2239750 1.167947 0.06742319 5.610445e-01  -0.5847601 8.551052e-02
#> 6 -0.1785000 1.131707 0.05373385 6.118866e-01  -0.5105974 6.572160e-01
#>   MW_test_stat t_test_p_adj MW_test_p_adj       p.used Selected as significant?
#> 1   -4.0493114 0.0002366094  0.0003814941 2.570256e-05                      Yes
#> 2   -2.4545484 0.0157790854  0.0394972722 1.360267e-02                      Yes
#> 3   -2.5724516 0.1799327522  0.0394972722 1.030219e-02                       No
#> 4   -1.5254255 0.3527853502  0.2225177371 1.295905e-01                       No
#> 5   -1.7195706 0.7138676822  0.1710210453 8.683564e-02                       No
#> 6   -0.4437602 0.7138676822  0.7902902022 6.645315e-01                       No
```

Three genes (ANGPT1, IL8, and VEGFB) are shown to meet the used significance criteria.

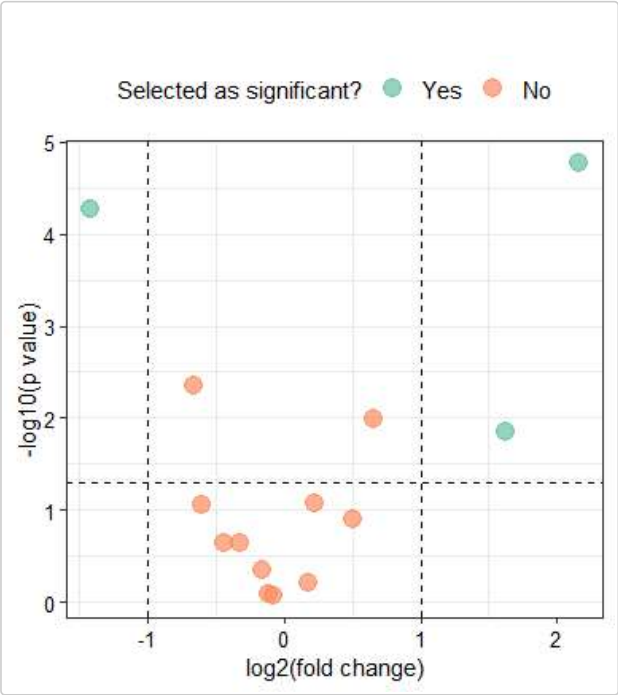
NOTE: If p values were not calculated due to the low number of samples, or they are not intended to be used to create a plot, the `use.p` parameter should be set to `FALSE`.

The `results_volcano()` function

This function creates a volcano plot that illustrates the arrangement of genes based on fold change values and p values. Significant genes can be pointed out using specified p value and fold change thresholds, and highlighted on the plot by color and (optionally) isolated by thresholds lines. Similarly to `FCh_plot()` function, data returned from the `RQ_dCt()` and `RQ_ddCt()` functions can be directly applied (see The summary of standard workflow) and various sources of used p values are available (from the Student's t test, the Mann-Whitney U test, depended on the normality of data, or provided by the user).

The created plot is displayed on the graphic device. The returned object is a lists that contain two elements: an object with plot and a table with results.

```
# Genes with p < 0.05 and 2-fold changed expression between compared groups are considered significant:
volcano <- results_volcano(data = results.ddCt,
                           mode = "depends",
                           p.threshold = 0.05,
                           FCh.threshold = 2)
```



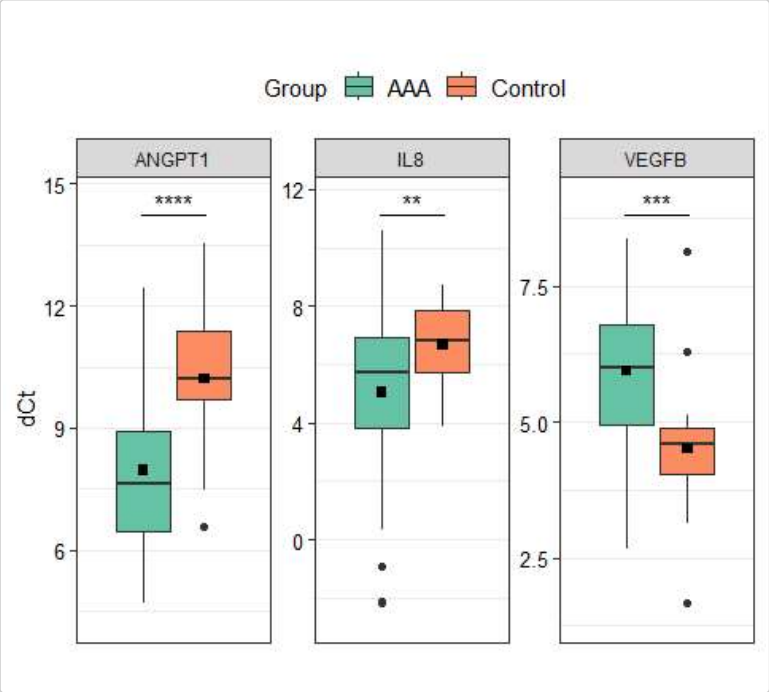

```
# Access the table with results:
head(as.data.frame(volcano[[2]]))
#>     Gene AAA_mean Control_mean   AAA_sd Control_sd   AAA_norm_p Control_norm_p
#> 1 ANGPT1 7.959426    10.117002 1.887131   1.688115 1.744688e-01      0.4266422
#> 2   CCL2 9.071111     8.992292 1.857556   1.659579 4.155848e-01      0.2597440
#> 3   CCL5 0.360150     0.584125 1.670593   1.358816 2.060155e-07      0.0314906
#> 4   FGF2 9.536064     8.932742 1.205243   1.416715 6.129405e-01      0.6144724
#> 5   IL1B 3.922625     4.101125 1.381818   1.336964 8.555529e-02      0.4723124
#> 6    IL8 5.058075     6.675667 2.951318   1.407052 4.414803e-02      0.1729972
#>          ddCt       FCh    log10FCh     t_test_p t_test_stat    MW_test_p
#> 1 -2.15757627 4.4616467  0.64949517 1.690067e-05  -4.7334112 5.136855e-05
#> 2  0.07881944 0.9468321 -0.02372702 8.611239e-01   0.1757974 9.889357e-01
#> 3 -0.22397500 1.1679472  0.06742319 5.610445e-01  -0.5847601 8.551052e-02
#> 4  0.60332244 0.6582363 -0.18161815 8.872007e-02   1.7420508 6.717362e-02
#> 5 -0.17850000 1.1317066  0.05373385 6.118866e-01  -0.5105974 6.572160e-01
#> 6 -1.61759167 3.0686235  0.48694361 4.508310e-03  -2.9520830 1.410617e-02
#>   MW_test_stat t_test_p_adj MW_test_p_adj test.for.comparison       p.used
#> 1   -4.0493114 0.0002366094  0.0003814941    t.student's.test 1.690067e-05
#> 2    0.0138675 0.8611239429  0.9889356866    t.student's.test 8.611239e-01
#> 3   -1.7195706 0.7138676822  0.1710210453   Mann-Whitney.test 8.551052e-02
#> 4    1.8305106 0.2070134948  0.1567384376    t.student's.test 8.872007e-02
#> 5   -0.4437602 0.7138676822  0.7902902022    t.student's.test 6.118866e-01
#> 6   -2.4545484 0.0157790854  0.0394972722   Mann-Whitney.test 1.410617e-02
#>   Selected as significant?
#> 1                      Yes
#> 2                       No
#> 3                       No
#> 4                       No
#> 5                       No
#> 6                      Yes
```

The `results_boxplot()` function

This function creates a boxplot that illustrates the distribution of the data for the genes. It is similar to `control_boxplot_gene()` function; however, some new options are added, including gene selection, faceting, addition of mean points to boxes, and statistical significance labels.

Data objects returned from the `make_Ct_ready()` and `delta_Ct()` functions can be directly applied to this function (see The summary of standard workflow).

```
final_boxplot <- results_boxplot(data = data.dCtF,
                                 sel.Gene = c("ANGPT1","IL8", "VEGFB"),
                                 by.group = TRUE,
                                 signif.show = TRUE,
                                 signif.labels = c("****","**","***"),
                                 signif.dist = 1.05,
                                 faceting = TRUE,
                                 facet.row = 1,
                                 facet.col = 4,
                                 y.exp.up = 0.1,
                                 angle = 20,
                                 y.axis.title = "dCt")
```

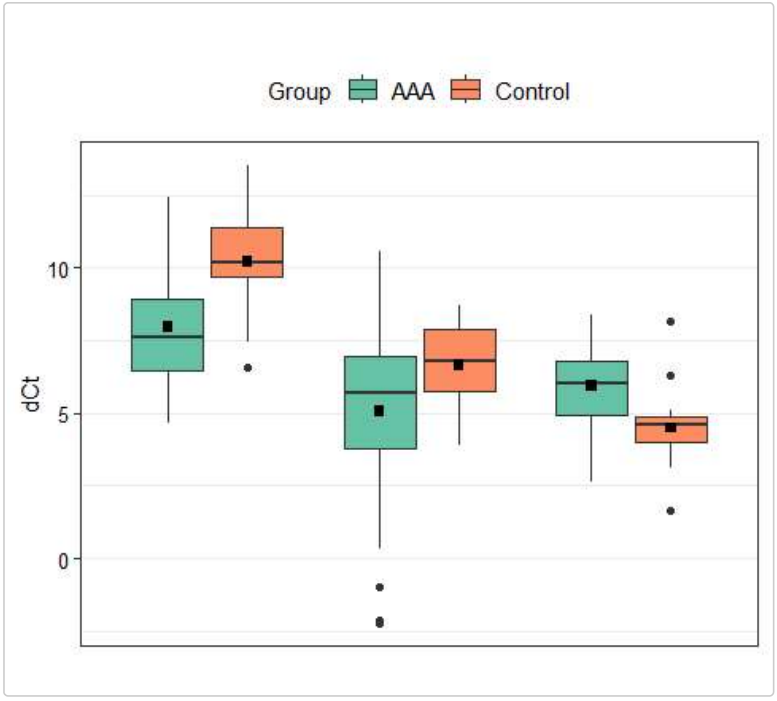


NOTE: If missing values are present in the data, they will be automatically removed, and a warning will appear.

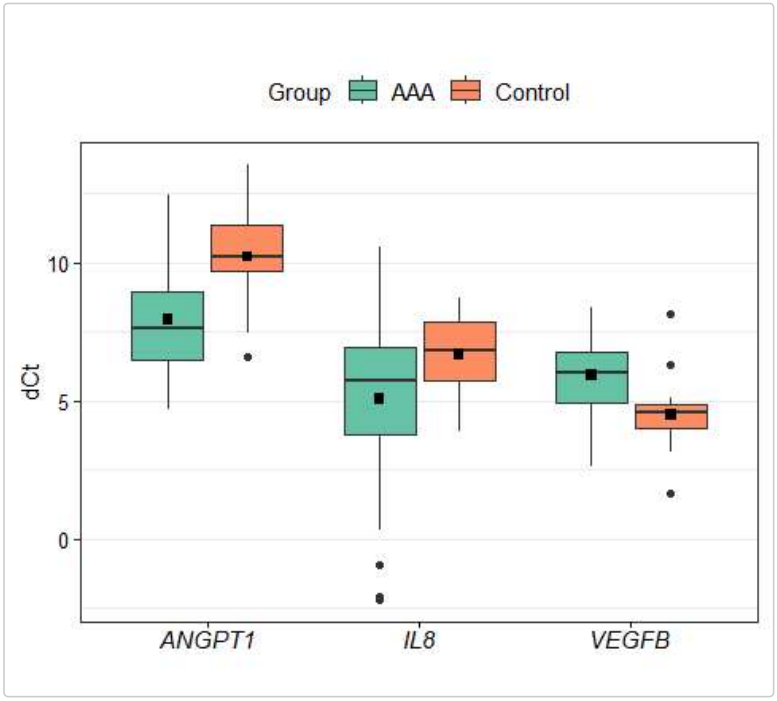
IMPORTANT NOTE: Significance labels are properly located on the plot only if faceting is used (`faceting = TRUE`). For plots without faceting, coordinates of significance labels depends on the data and need to be individually provided (can not be added automatically). In such a situations, the following solution can be used.

Step 1: Prepare object that contain plot without faceting and significance labels:

```
final_boxplot_no_fac <- results_boxplot(data = data.dCtF,
                                        sel.Gene = c("ANGPT1","IL8", "VEGFB"),
                                        by.group = TRUE,
                                        signif.show = FALSE,  # Disable significance labels
                                        faceting = FALSE, # Disable faceting
                                        y.axis.title = "dCt") +
                                        theme(axis.text.x = element_text(size = 5, colour = "black"))
```




```
# Add x axis annotations and ticks:
final_boxplot_no_fac <- final_boxplot_no_fac +
                        theme(axis.text.x = element_text(size = 11, colour = "black", face="italic"),
         # Use italic font for human gene symbols
                              axis.ticks.x = element_line(colour = "black"))
final_boxplot_no_fac
```

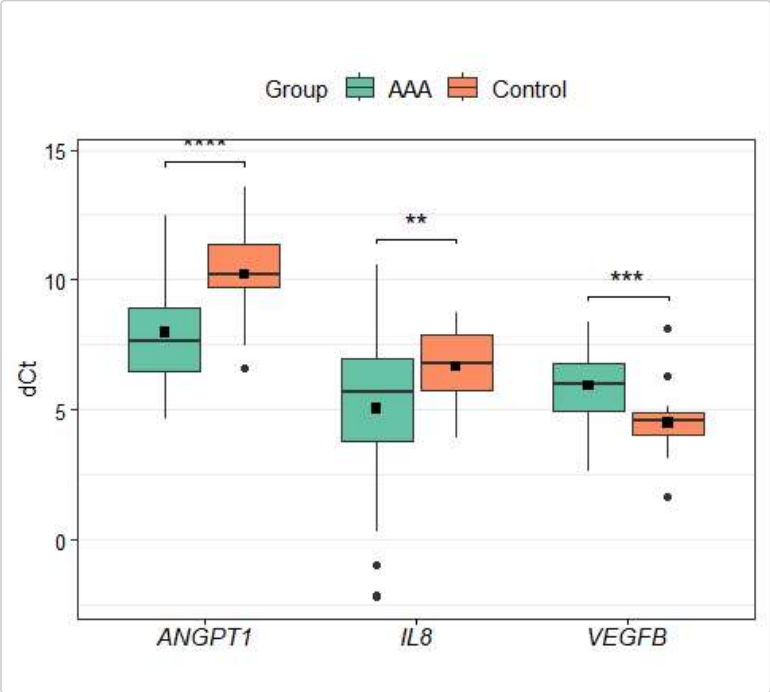


Step 2: Prepare tables with coordinates for significance labels:

```
data.label <- data.frame(matrix(nrow = 3, ncol = 4)) # Number of rows should be equal to number of
         genes
rownames(data.label) <- c("ANGPT1","IL8","VEGFB")
colnames(data.label) <- c("x", "xend", "y", "annotation")
data.label$Gene <- rownames(data.label)

data.label$y <- 1 + c(max(data.dCtF$ANGPT1), max(data.dCtF$IL8), max(data.dCtF$VEGFB))
data.label$x <- c(0.81,1.81,2.81)
data.label$xend <- c(1.19,2.19,3.19)
data.label$annotation <- c("****","**","***")
```

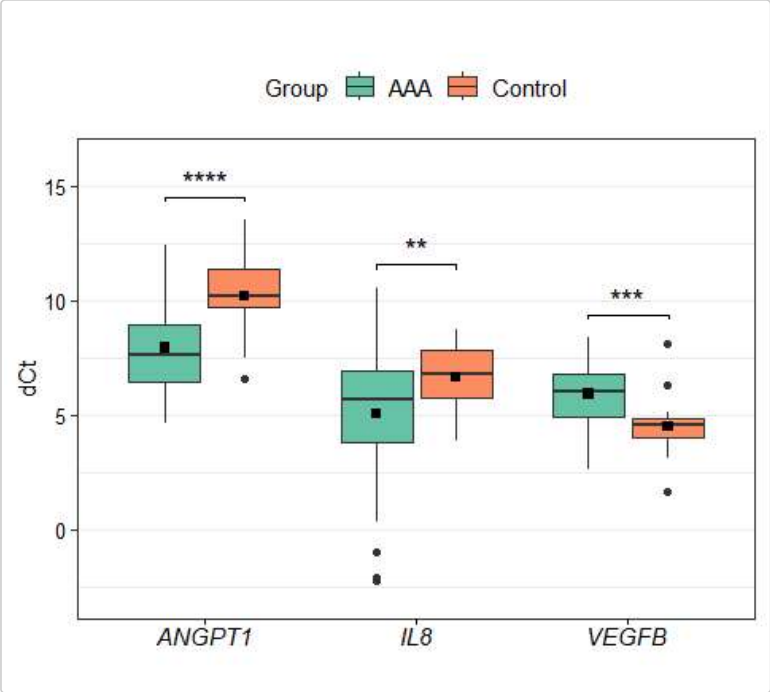
Step 3: Add significance labels to the plot:

```
final_boxplot_no_fac_ok <- final_boxplot_no_fac +
                           geom_signif(annotation = data.label$annotation,
                                       y_position = data.label$y,
                                       xmin = data.label$x,
                                       xmax = data.label$xend,
                                       tip_length = 0.01,
                                       textsize = 5)
final_boxplot_no_fac_ok
```



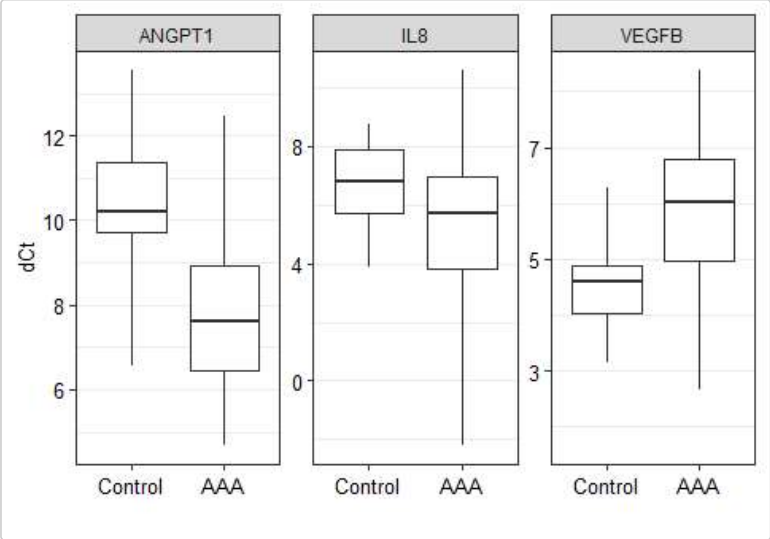
Step 4: Minor improvement and the plot is ready!

```
final_boxplot_no_fac_ok <- final_boxplot_no_fac_ok +
                           scale_y_continuous(expand = expansion(mult = c(0.1, 0.15))) # Make room for
         the first label

final_boxplot_no_fac_ok
```



A situation may arise in which the user prefers to generate a faceted plot without a color legend, displaying group names as annotations on the x axis. This can be the solution for this situation:

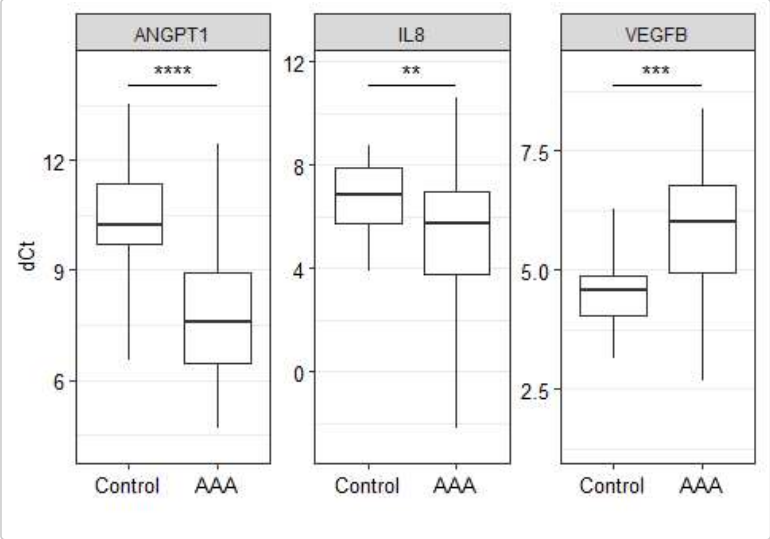
```
library(tidyverse)
colnames(data.dCtF)
#>  [1] "Group"  "Sample" "ANGPT1" "CCL2"   "CCL5"   "FGF2"   "IL1B"   "IL8"
#>  [9] "PDGFA"  "PDGFB"  "TGFA"   "TGFB"   "TNF"    "VEGFA"  "VEGFB"  "VEGFC"
data.dCtF.slim <- pivot_longer(data.dCtF, cols = ANGPT1:VEGFC, names_to = "gene", values_to = "exp")

# Select genes
data.dCtF.slim_sel <- data.dCtF.slim[data.dCtF.slim$gene %in% c("ANGPT1","IL8","VEGFB"), ]

# Change order of groups if needed
data.dCtF.slim_sel$Group <- factor(data.dCtF.slim_sel$Group, levels = c("Control","AAA"))

# Create plot
final_boxplot_no_colors <- ggplot(data.dCtF.slim_sel, aes(x = Group, y = exp)) +
                           geom_boxplot(outlier.shape = NA, coef = 2) +
                           theme_bw() +
                           ylab("dCt") +
                           xlab("") +
                           theme(axis.text = element_text(size = 10, color = "black")) +
                           theme(axis.title = element_text(size = 10, color = "black")) +
                           theme(panel.grid.major.x = element_blank()) +
                           facet_wrap(vars(gene), nrow = 1, dir = "h", scales = "free")

final_boxplot_no_colors
```



To add significance labels, the following code can be used:

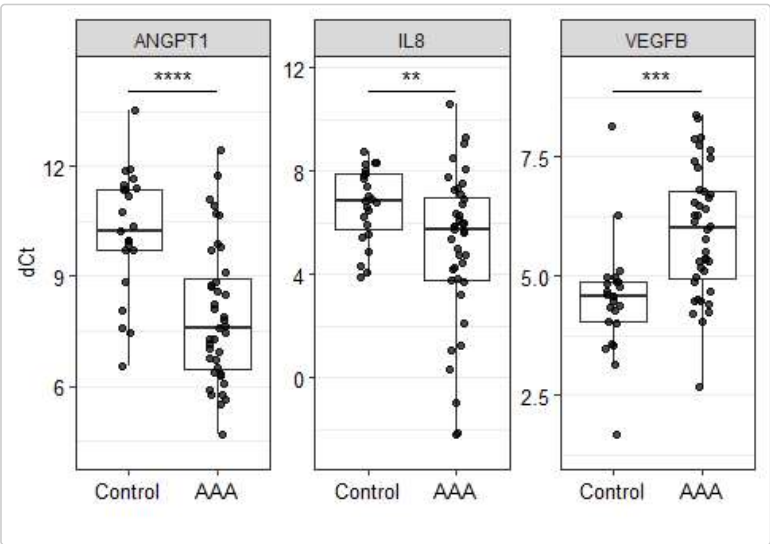
```
data.label <- data.frame(matrix(nrow = 3, ncol = 4)) # Number of rows is equal to number of genes
rownames(data.label) <- c("ANGPT1","IL8","VEGFB")
colnames(data.label) <- c("x", "xend", "y", "annotation")
data.label$gene <- rownames(data.label) # Name of column with gene symbols in this table must be
# the same as name of the column with gene symbols in data used for create the plot.

data.label$y <- 0.5 + c(max(data.dCtF$ANGPT1), max(data.dCtF$IL8), max(data.dCtF$VEGFB))
data.label$x <- c(1,1,1)
data.label$xend <- c(1.98,1.98,1.98)
data.label$annotation <- c("****","**","***")

final_boxplot_no_colors_labels <- final_boxplot_no_colors +
 geom_signif(
    stat = "identity",
    data = data.label,
    aes(x = x,
        xend = xend,
        y = y,
        yend = y,
        annotation = annotation),
```

```
    color = "black",
    manual = TRUE) +
  scale_y_continuous(expand = expansion(mult = c(0.1, 0.1)))

final_boxplot_no_colors_labels
```



If points of individual samples need to be added, it can be simply run:

```
final_boxplot_no_colors_labels_points <- final_boxplot_no_colors_labels +
                                         geom_point(position=position_jitter(w=0.1,h=0),
                                                    alpha = 0.7,
                                                    size = 1.5)

final_boxplot_no_colors_labels_points
```



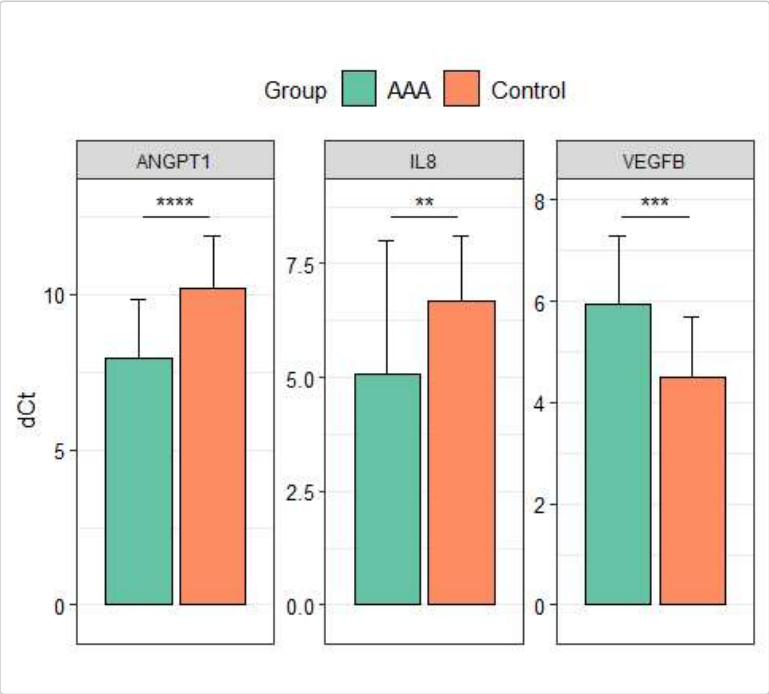
The `results_barplot()` function

This function creates a barplot that illustrates mean and standard deviation values of the data for all or selected genes.

Data objects returned from the `make_Ct_ready()` and `delta_Ct() functions` can be directly applied to this function (see The summary of standard workflow).

```
final_barplot <- results_barplot(data = data.dCtF,
                                 sel.Gene = c("ANGPT1","IL8", "VEGFB"),
                                 signif.show = TRUE,
                                 signif.labels = c("****","**","***"),
                                 angle = 30,
                                 signif.dist = 1.05,
                                 faceting = TRUE,
                                 facet.row = 1,
```

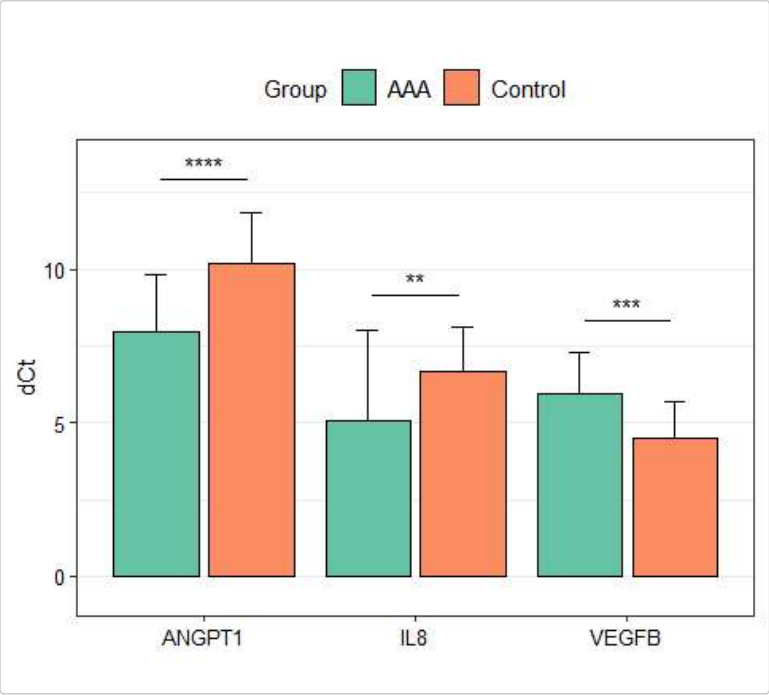
```
                    facet.col = 4,
                    y.exp.up = 0.1,
                    y.axis.title = "dCt")
```



NOTE: At least two samples in each group are required to calculate the standard deviation and properly generate the plot.

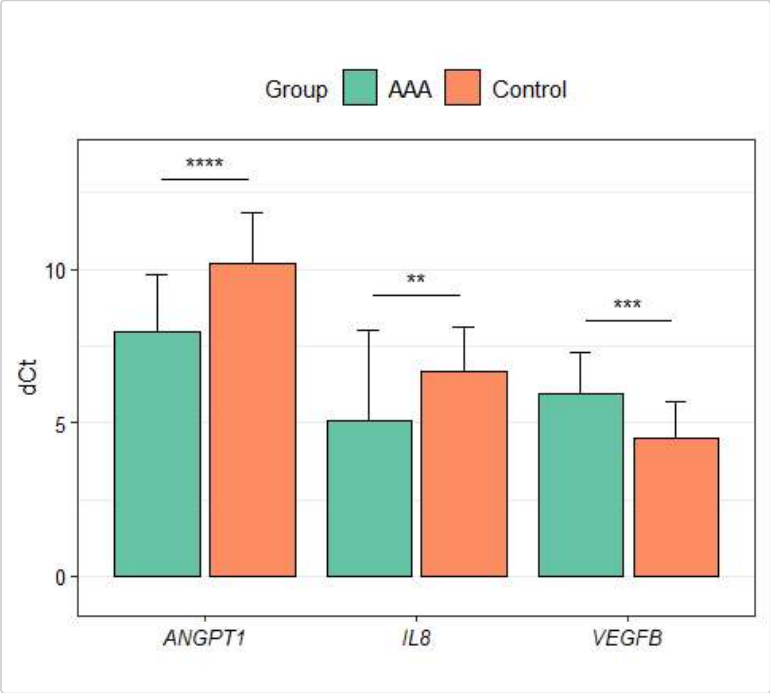
If faceting is not needed, simply run this function with `faceting = FALSE`:

```
final_barplot <- results_barplot(data = data.dCtF,
                                 sel.Gene = c("ANGPT1","IL8", "VEGFB"),
                                 signif.show = TRUE,
                                 signif.labels = c("****","**","***"),
                                 angle = 0,
                                 signif.dist = 1.05,
                                 faceting = FALSE,
                                 y.exp.up = 0.1,
                                 y.axis.title = "dCt")
```




```
# Add italic font to the x axis:
```

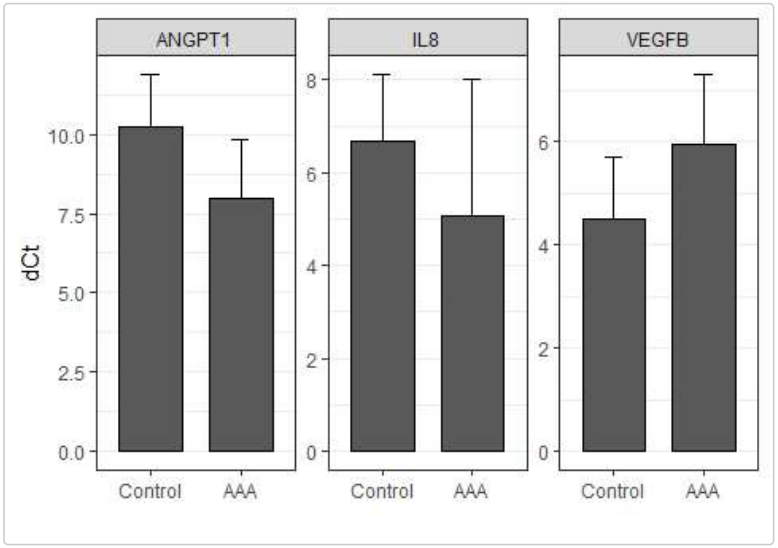
```
final_barplot <- final_barplot +
                 theme(axis.text.x = element_text(face="italic"))

final_barplot
```



A situation may arise in which the user prefers to generate a faceted plot without a color legend, displaying group names as annotations on the x-axis. This is the solution for this situation:

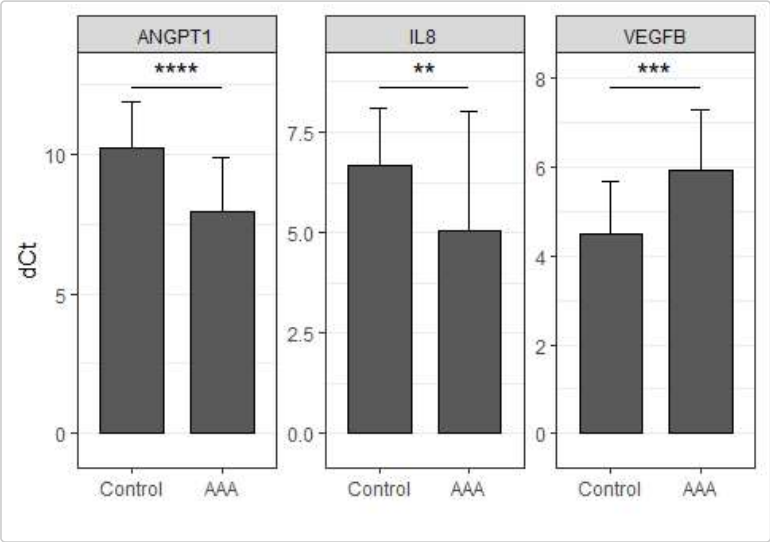
```
library(tidyverse)
colnames(data.dCtF)
#>  [1] "Group"  "Sample" "ANGPT1" "CCL2"   "CCL5"   "FGF2"   "IL1B"   "IL8"
#>  [9] "PDGFA"  "PDGFB"  "TGFA"   "TGFB"   "TNF"    "VEGFA"  "VEGFB"  "VEGFC"
data.dCtF.slim <- pivot_longer(data.dCtF, cols = ANGPT1:VEGFC, names_to = "gene", values_to = "exp")

# Select genes
data.dCtF.slim_sel <- data.dCtF.slim[data.dCtF.slim$gene %in% c("ANGPT1","IL8","VEGFB"), ]

# Change order of groups if needed
data.dCtF.slim_sel$Group <- factor(data.dCtF.slim_sel$Group, levels = c("Control","AAA"))

data.mean <- data.dCtF.slim_sel %>%
              group_by(Group, gene) %>%
              summarise(mean = mean(exp, na.rm = TRUE), .groups = "keep")

data.sd <- data.dCtF.slim_sel %>%
            group_by(Group, gene) %>%
            summarise(sd = sd(exp, na.rm = TRUE), .groups = "keep")

data.mean$sd <- data.sd$sd

final_barplot_no_colors <- ggplot(data.mean, aes(x = Group, y = mean)) +
                               geom_errorbar(aes(group = Group,
                                                 y = mean,
                                                 ymin = ifelse(mean < 0, mean - abs(sd), mean),
                                                 ymax = ifelse(mean > 0, mean + abs(sd), mean)),
                                             width = .2,
                                             position = position_dodge(0.9)) +

                               geom_col(aes(group = Group),
                                        position = position_dodge(0.9),
                                        width = 0.7,
                                        color = "black") +
                  xlab("") +
```

```
                ylab("dCt") +
                theme_bw() +
                #theme(axis.text = element_text(size = axis.text.size, colour = "black")) +
                #theme(axis.title = element_text(size = axis.title.size, colour = "black")) +
                #theme(legend.text = element_text(size = legend.text.size, colour ="black")) +
                #theme(legend.title = element_text(size = legend.title.size, colour =  "black")) +
                theme(panel.grid.major.x = element_blank()) +
  facet_wrap(vars(gene), scales = "free", nrow = 1)

final_barplot_no_colors
```



Significance labels can be added similarly as regarding `results_boxplot()` function:

```
data.label <- data.frame(matrix(nrow = 3, ncol = 4)) # Number of rows is equal to number of genes
rownames(data.label) <- c("ANGPT1","IL8","VEGFB")
colnames(data.label) <- c("x", "xend", "y", "annotation")
data.label$gene <- rownames(data.label) # Name of column with gene symbols in this table
# must be the same as name of the column with gene symbols in data used for create the plot.

data.mean <- data.mean %>%
               mutate(max = mean + sd) %>%
               group_by(gene) %>%
               summarise(height = max(max, na.rm = TRUE), .groups = "keep")
data.label$y <- 0.5 + data.mean$height
data.label$x <- c(1,1,1)
data.label$xend <- c(1.98,1.98,1.98)
data.label$annotation <- c("****","**","***")

final_barplot_no_colors_labels <- final_barplot_no_colors +
 geom_signif(
    stat = "identity",
    data = data.label,
    aes(x = x,
        xend = xend,
        y = y,
        yend = y,
        annotation = annotation,
        textsize = 5),
    color = "black",
    manual = TRUE) +
   scale_y_continuous(expand = expansion(mult = c(0.1, 0.1)))

final_barplot_no_colors_labels
```



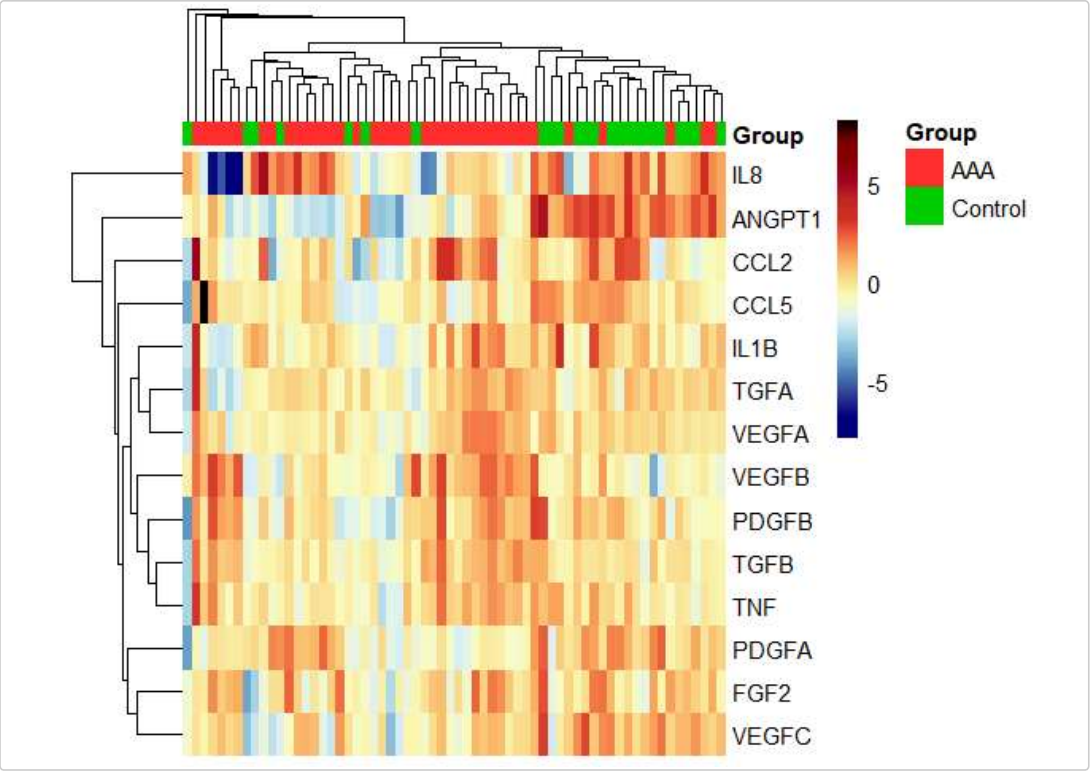
The `results_heatmap()` function

This function allows to draw heatmap with hierarchical clustering. Various methods of distance calculation (e.g. euclidean, canberra) and agglomeration (e.g. complete, average, single) can be used.

Data objects returned from the `make_Ct_ready()` and `delta_Ct() functions` can be directly applied to this function (see The summary of standard workflow).

NOTE: Remember to create named list with colors for groups annotation and pass it in `col.groups` parameter.

```
# Create named list with colors for groups annotation:
colors.for.groups = list("Group" = c("AAA"="firebrick1","Control"="green3"))
# Vector of colors for heatmap can be also specified to fit the user needings:
colors <- c("navy","navy","#313695","#4575B4","#74ADD1","#ABD9E9",
            "#E0F3F8","#FFFFBF","#FEE090","#FDAE61","#F46D43",
            "#D73027","#C32B23","#A50026","#8B0000",
            "#7E0202","#000000")
results_heatmap(data.dCt,
                sel.Gene = "all",
                col.groups = colors.for.groups,
                colors = colors,
                show.colnames = FALSE,
                show.rownames = TRUE,
                fontsize = 11,
                fontsize.row = 11,
                cellwidth = 4) # It avoids cropping the image on the right side.
```



The created plot is displayed on the graphic device (if `save.to.tiff = FALSE`) or saved to .tiff file (if `save.to.tiff = TRUE`).

Further analyses

Gene expression levels and differences between groups can be further analysed using the following methods and corresponding functions of the `RQdeltaCT` package:

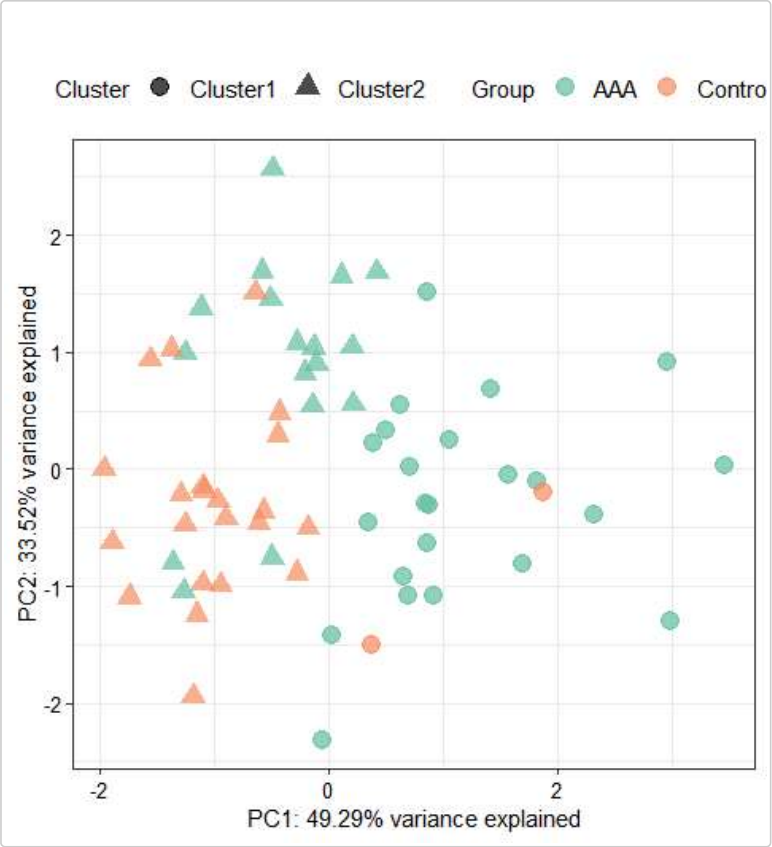
- principal component analysis (PCA) and k means clustering - used to assess samples clustering based on the gene expression data (`pca_kmeans()` function)
- correlation analysis - used to generate and visualise the correlation matrix of samples (`corr_sample()` function) and genes (`corr_gene()` function).
- simple linear regression - used to analysis and visualisation of relationships between pair of samples (`single_pair_sample()` function) or genes (`single_pair_gene()` function).
- Receiver Operating Characteristic (ROC) analysis - used to evaluate performance of sample classification by gene expression data (`ROCh()` function).
- simple logistic regression analysis - used to calculate odds ratio values for genes (`log_reg()` function).

Data objects returned from the `make_Ct_ready()` and `delta_Ct() functions` can be directly applied to all of these functions (see The summary of standard workflow). Moreover, all functions can be run on the entire data or only on selected samples/genes.

PCA and k means clustering

This function allows to simultaneously perform principal component analysis (PCA) and, if `do.k.means = TRUE`, samples classification using k means method. Number of clusters can be set using `k.clust` parameter. Results obtained from both methods are presented on one plot. Obtained clusters are distinguishable by point shapes. Confusion matrix of sample classification is returned as the second element of the returned object and can be used for calculation further parameters of classification performance like precision, accuracy, recall, and others.

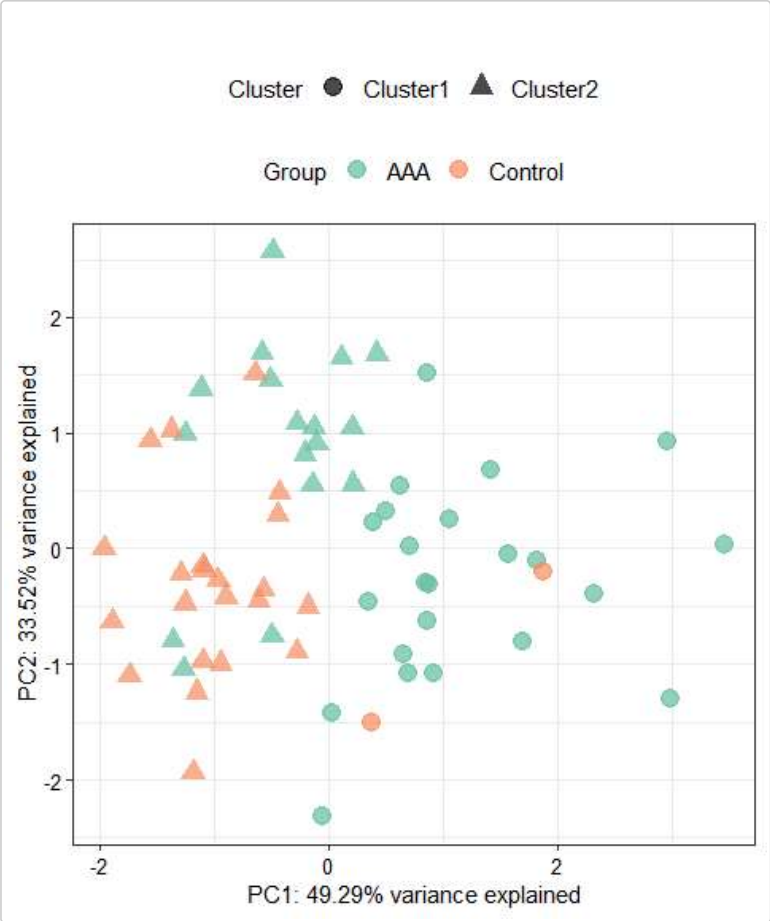
```
pca.kmeans <- pca_kmeans(data.dCt,
                         sel.Gene = c("ANGPT1","IL8", "VEGFB"),
                         legend.position = "top")
```




```
# Access to the confusion matrix:
pca.kmeans[[2]]
#>
#>           Cluster1 Cluster2
#>   AAA           23       17
#>   Control        2       22
```

If the legend is to wide and is cropped, setting a vertical position of legend should solve this problem:

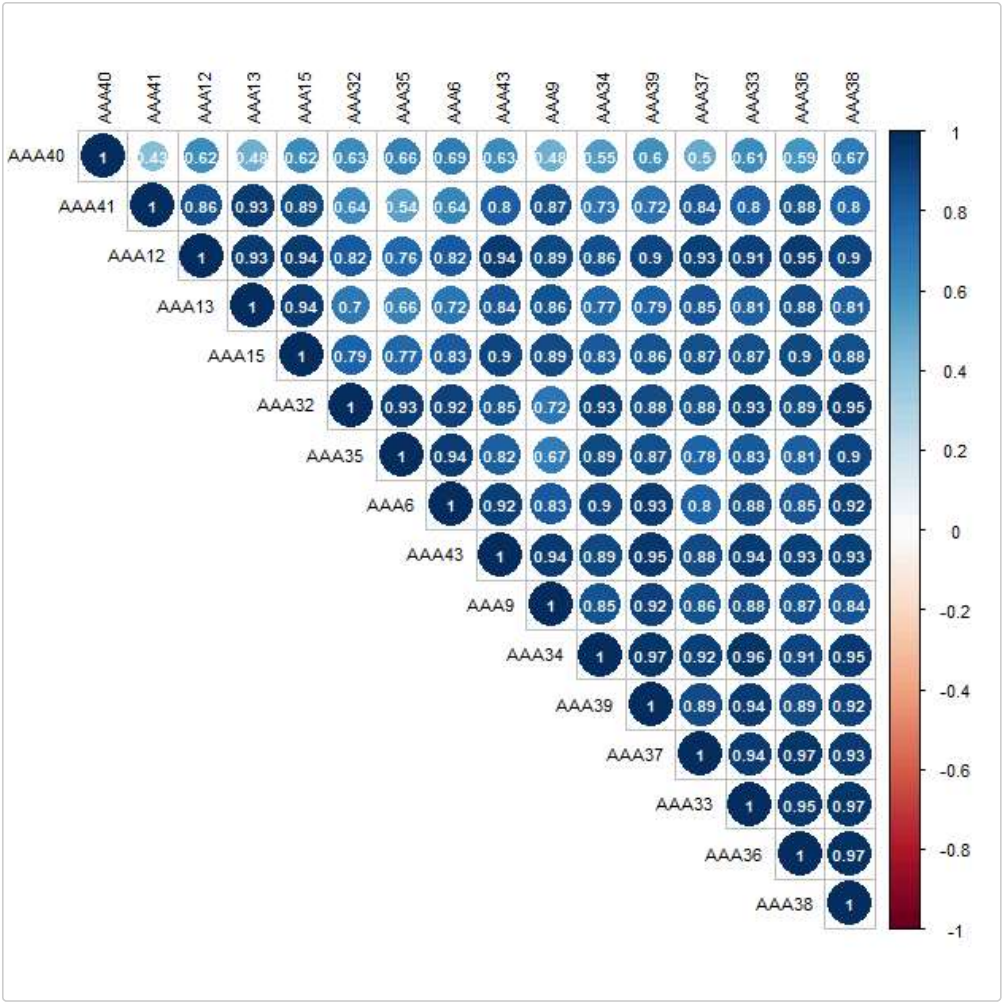
```
pca.kmeans[[1]] + theme(legend.box = "vertical")
```



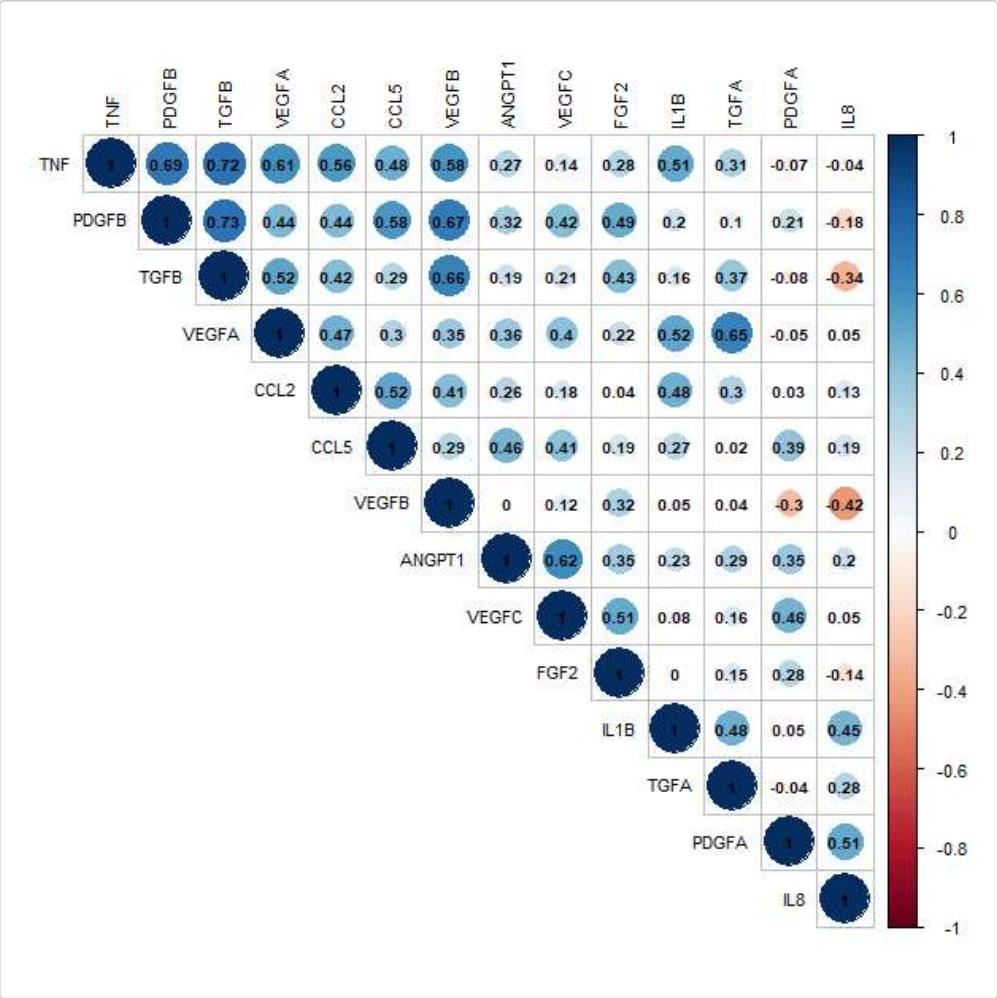
Correlation analysis

Correlation analysis is a very useful method to explore linear relationships between variables. The `RQdeltaCT` package offers `corr_sample()` and `corr_gene()` functions to generate and plot correlation matrices of samples and genes, respectively. The correlation coefficients can be calculated using either the Pearson or Spearman algorithm. To facilitate plot interpretation, these functions also have possibilities to order samples or genes according to several methods, e.g. hierarchical clustering or PCA first component, by using `order` parameter.

```
library(Hmisc)
library(corrplot)
# To make the plot more readable, only part of the data was used:
corr.samples <- corr_sample(data = data.dCt[15:30, ],
                            method = "pearson",
                            order = "hclust",
                            size = 0.7,
                            p.adjust.method = "BH",
                            add.coef = "white")
```




```
library(Hmisc)
library(corrplot)
corr.genes <- corr_gene(data = data.dCt,
                        method = "spearman",
                        order = "FPC",
                        size = 0.7,
                        p.adjust.method = "BH")
```



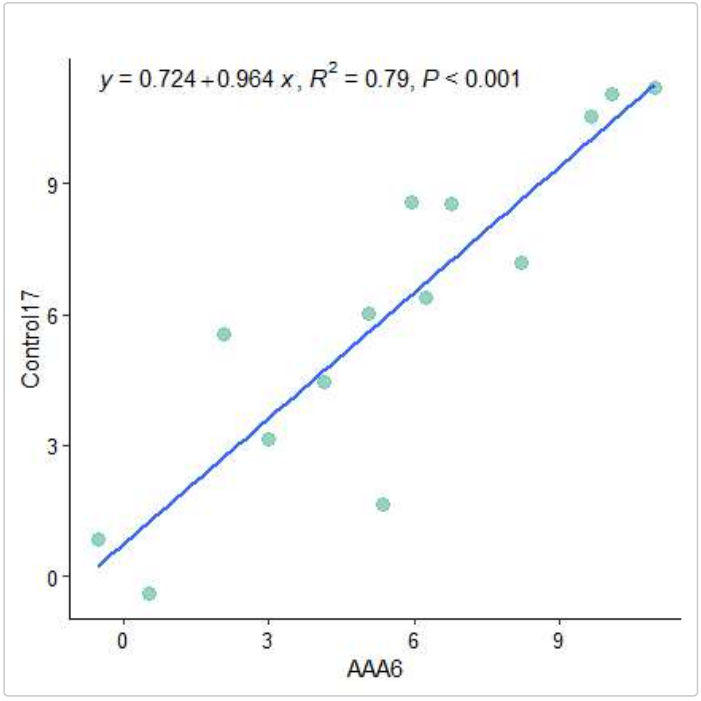
The created plots are displayed on the graphic device. The returned objects are tables with computed correlation coefficients, p values, and p values adjusted by Benjamini-Hochberg correction (by default). Tables are sorted by the absolute values of correlation coefficients in descending order.

NOTE: A minimum of 5 samples/target are required for correlation analysis.

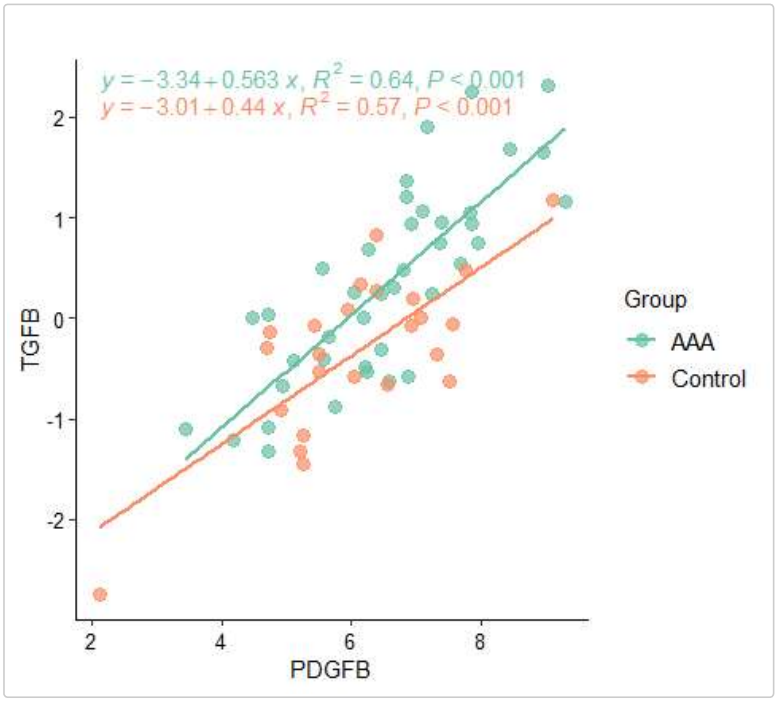
Simple linear regression analysis

Linear relationships between pairs of samples or genes can be further analysed using simple linear regression models using the `single_pair_sample()` function (for analysis of samples) and the `single_pair_gene()` function for analysis of genes. These functions draw a scatter plot with a simple linear regression line. Regression results such as regression equation, coefficient of determination, F value, or p value can be optionally added to the plot.

```
library(ggpmisc)
AAA6_Control17 <- single_pair_sample(data = data.dCt,
                                     x = "AAA6",
                                     y = "Control17",
                                     point.size = 3,
                                     labels = TRUE,
                                     label = c("eq", "R2", "p"),
                                     label.position.x = 0.05)
```




```
library(ggpmisc)
PDGFB_TGFB <- single_pair_gene(data.dCt,
                               x = "PDGFB",
                               y = "TGFB",
                               by.group = TRUE,
                               point.size = 3,
                               labels = TRUE,
                               label = c("eq", "R2", "p"),
                               label.position.x = c(0.05),
                               label.position.y = c(1,0.95))
```

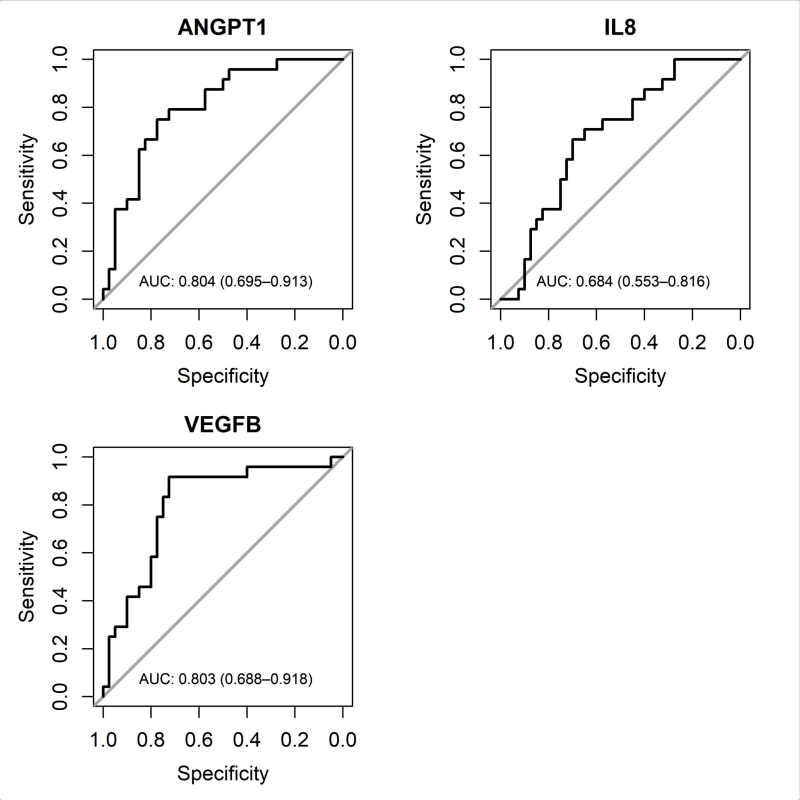


Receiver Operating Characteristic (ROC) analysis

The Receiver Operating Characteristic (ROC) analysis is useful for assessing the performance of sample classification to the particular group, based on gene expression data. In this analysis, ROC curves together with parameters such as the area under curve (AUC), specificity, sensitivity, accuracy, positive and negative predictive value are received. The `ROCh()` function was designed to perform all of these tasks. This function returns a table with calculated parameters and a plot with multiple panels, each with ROC curve for one gene.

NOTE: The created plot is not displayed on the graphic device, but should be saved as .tiff image (`save.to.txt = TRUE`) and can be opened directly from the file in the working directory.

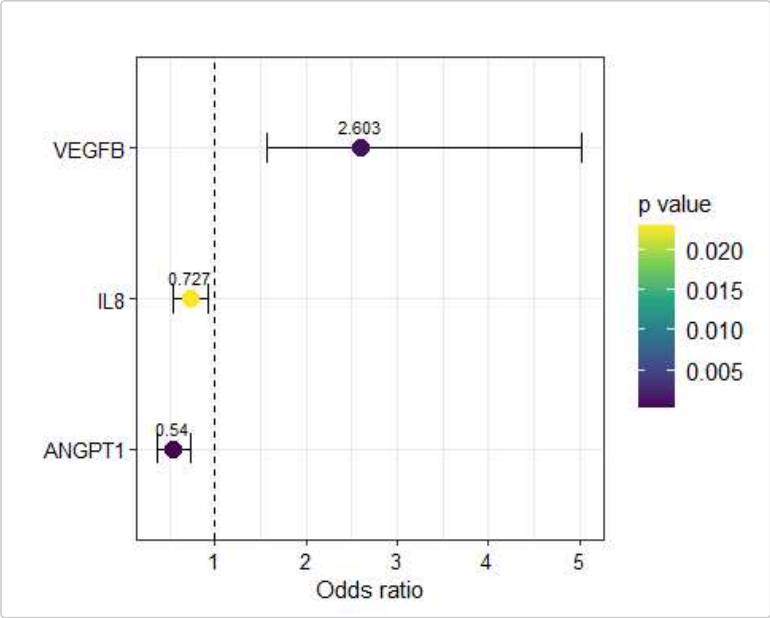
```
library(pROC)
# Remember to specify the numbers of rows (panels.row parameter) and columns (panels.col parameter)
# to be sufficient to arrange panels:
roc_parameters <- ROCh(data = data.dCt,
                       sel.Gene = c("ANGPT1","IL8", "VEGFB"),
                       groups = c("AAA","Control"),
                       panels.row = 2,
                       panels.col = 2)
roc_parameters
#>     Gene Threshold Specificity Sensitivity Accuracy       ppv       npv
#> 1 ANGPT1   9.40925       0.775   0.7500000 0.765625 0.6666667 0.8378378
#> 2    IL8   6.40600       0.700   0.6666667 0.687500 0.5714286 0.7777778
#> 3  VEGFB   5.11675       0.725   0.9166667 0.796875 0.6666667 0.9354839
#>     youden       AUC
#> 1 1.525000 0.8041667
#> 2 1.366667 0.6843750
#> 3 1.641667 0.8031250
```



Simple logistic regression

Logistic regression is a useful method to investigate the impact of the analysed variable on the odds of the occurrence of the studied experimental condition. In the `RQdeltaCT` package, `log_reg()` function allows to calculate for each gene a chances (odds ratio, OR) of being included in the study group when gene expression level increases by one unit (suitable for non-transformed data) or by mean of expression levels (more suitable for transformed data). This function returns a plot and table with the calculated parameters (OR, confidence interval, intercept, coefficient, and p values).

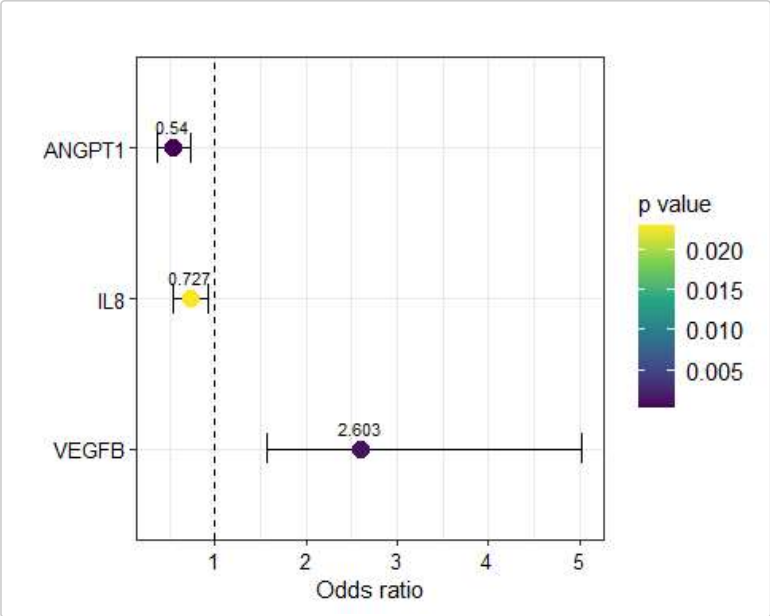
```
library(oddsratio)
# Remember to set the increment parameter.
log.reg.results <- log_reg(data = data.dCt,
                           increment = 1,
                           sel.Gene = c("ANGPT1","IL8", "VEGFB"),
                           group.study = "AAA",
                           group.ref = "Control")
```




```
log.reg.results[[2]]
#>     Gene oddsratio CI_low CI_high Increment Intercept coeficient  p_intercept
#> 1 ANGPT1     0.540  0.374   0.734         1  6.082084 -0.6159817 0.0001466894
#> 2    IL8     0.727  0.534   0.930         1  2.416073 -0.3189009 0.0085244809
#> 3  VEGFB     2.603  1.572   5.031         1 -4.444207  0.9567034 0.0027620282
#>         p_coef   p_coef_adj
#> 1 0.0002879457 0.0008638371
#> 2 0.0229318870 0.0229318870
#> 3 0.0010298471 0.0015447706
```

If genes should be displayed in alphabetical order, simply sort them:

```
log.reg.results.sorted <- log.reg.results[[1]] +
                          scale_y_discrete(limits = rev(sort(log.reg.results[[2]]$Gene)))
log.reg.results.sorted
```



Part B: A pairwise analysis

A pairwise analysis regards situations when samples are grouped in pairs. In general, a pairwise analysis is quite similar to the workflow for comparison of independent groups; however, some crucial points are different. Therefore, for the users convenience, in this part of vignette, a pairwise variant of analysis using the `RQdeltaCT` workflow was demonstrated in detail.

For a demonstration purposes, the data object named `data.Ct.pairwise` from `RQdeltaCT` package is used:

```
data(data.Ct.pairwise)
str(data.Ct.pairwise)
#> tibble [756 × 4] (S3: tbl_df/tbl/data.frame)
#>  $ Sample: chr [1:756] "Sample01" "Sample01" "Sample01" "Sample01" ...
#>  $ Group : chr [1:756] "After" "After" "After" "After" ...
#>  $ Gene  : chr [1:756] "Gene1" "Gene2" "Gene3" "Gene4" ...
#>  $ Ct    : chr [1:756] "32.563" "36.554" "31.334" "23.415" ...
```

This dataset contains 21 paired samples analyzed before (“Before” group) and after (“After” group) the exposure on the experimental factor. Total 18 genes were analyzed in these samples.

Quality control of raw Ct data (a pairwise approach)

The assessment of the quality and usefulness of the data can be conducted using `control_Ct_barplot_sample()` (for quality control of samples) and `control_Ct_barplot_gene()` (for quality control of genes) functions. Both functions require specifying quality control criteria to be applied to Ct values, in order to label each Ct value as reliable or not. These functions return numbers of reliable and unreliable Ct values in each sample or each gene, as well as total number of Ct values obtained from each sample and each gene. These results are presented graphically on barplots. The obtained results are useful for inspecting the analysed data in order to identify samples and genes that should be considered to be removed from the data (based on applied reliability criteria).

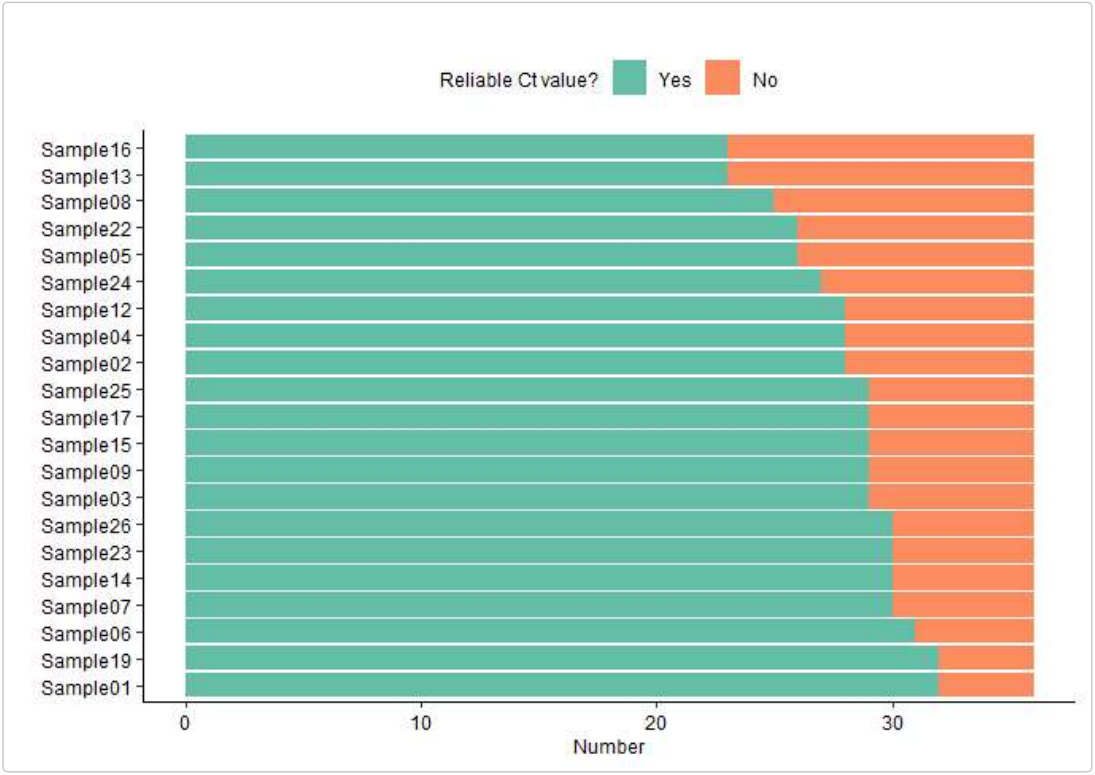
Three selection criteria can be set for these functions:

- a flag used for undetermined Ct values. Default to “Undetermined”.
- a maximum of Ct value allowed. Default to 35.
- a flag used in the Flag column for values which are unreliable. Default to “Undetermined”.

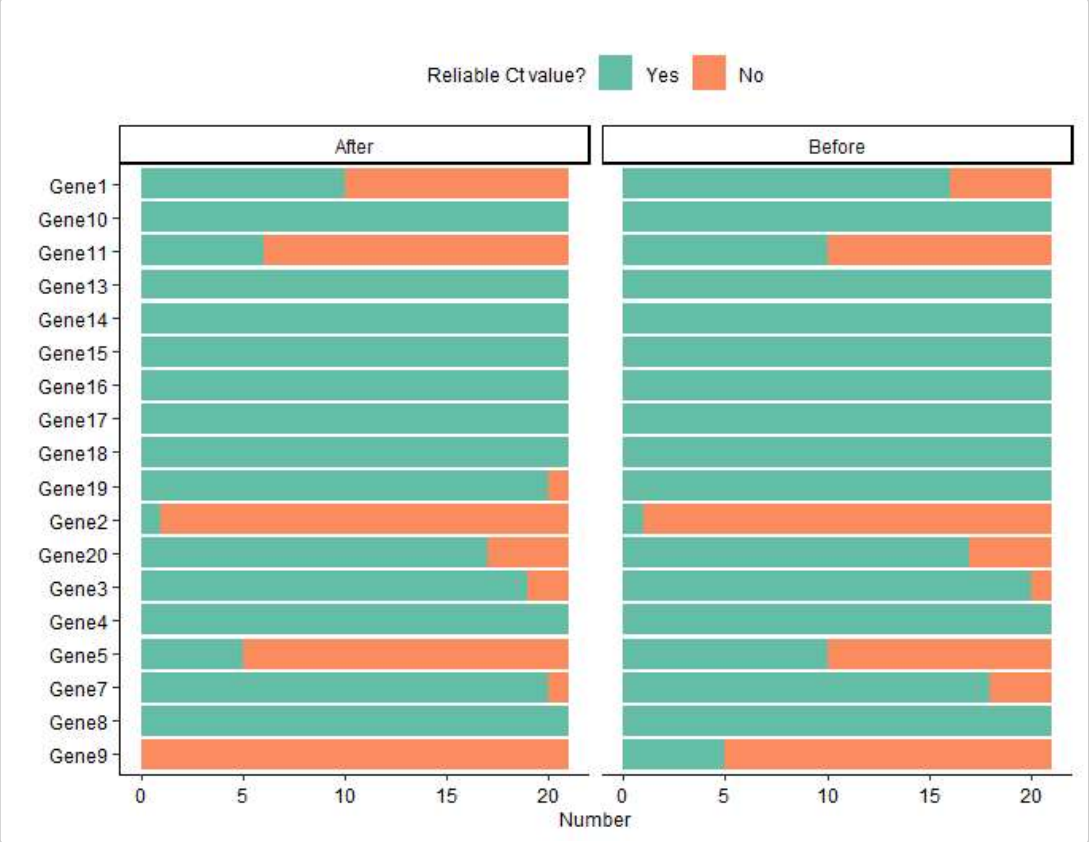
NOTE: This function does not perform data filtering, but only report numbers of Ct values labelled as reliable or not and presents them graphically.

An example of using these functions is provided below:

```
library(tidyverse)
sample.Ct.control.pairwise <- control_Ct_barplot_sample(data = data.Ct.pairwise,
                                                        flag.Ct = "Undetermined",
                                                        maxCt = 35,
                                                        flag = c("Undetermined"),
                                                        axis.title.size = 9,
                                                        axis.text.size = 9,
                                                        plot.title.size = 9,
                                                        legend.title.size = 9,
                                                        legend.text.size = 9)
```



```
gene.Ct.control.pairwise <- control_Ct_barplot_gene(data = data.Ct.pairwise,
                                                    flag.Ct = "Undetermined",
                                                    maxCt = 35,
                                                    flag = c("Undetermined"),
                                                    axis.title.size = 9,
                                                    axis.text.size = 9,
                                                    plot.title.size = 9,
                                                    legend.title.size = 9,
                                                    legend.text.size = 9)
```



Created plots are displayed on the graphic device, and short information about the returned tables appears. Returned objects are lists that contain two elements: an object with plot and a table with numbers of Ct values labelled as reliable (in column “Reliable”) and unreliable (in column “Not.reliable”), as well as fraction of unreliable Ct values in each gene. To easily identify samples or genes with high number of unreliable values, tables are sorted to show them at the top. To access returned tables, the second element of returned objects should be called:

```
head(sample.Ct.control.pairwise[[2]])
#> # A tibble: 6 × 4
#>   Sample   Not.reliable Reliable Not.reliable.fraction
#>   <fct>           <int>    <int>                <dbl>
#> 1 Sample13           13       23                0.361
#> 2 Sample16           13       23                0.361
#> 3 Sample08           11       25                0.306
#> 4 Sample05           10       26                0.278
#> 5 Sample22           10       26                0.278
#> 6 Sample24            9       27                0.25
```

```
head(gene.Ct.control.pairwise[[2]], 10)
#> # A tibble: 10 × 5
#>    Gene   Group  Not.reliable Reliable Not.reliable.fraction
#>    <fct>  <fct>         <int>    <int>                <dbl>
#>  1 Gene9  After            21        0                1
#>  2 Gene2  After            20        1                0.952
#>  3 Gene2  Before           20        1                0.952
#>  4 Gene5  After            16        5                0.762
#>  5 Gene9  Before           16        5                0.762
#>  6 Gene11 After            15        6                0.714
#>  7 Gene1  After            11       10                0.524
```

```
#>  8 Gene11 Before          11       10               0.524
#>  9 Gene5  Before          11       10               0.524
#> 10 Gene1  Before           5       16               0.238
```

Visual inspection of returned plots and obtained tables gives a clear, preliminary image of data quality. The results obtained in the examples show that the Before group contains the same number of samples as the After group. In all samples, the majority of Ct values are reliable.

Regarding genes, Gene9 has all unreliable Ct values in the After group. Other genes that can be considered to remove from the data are Gene2, Gene5, Gene11, and Gene1.

In some situations, a unified fraction of unreliable data need to be established and used to make decision which samples or genes should be excluded from the analysis. It can be done using the following code, in which the table returned by the `control_Ct_barplot_sample()` and `control_Ct_barplot_gene()` functions can be used to identify samples or genes for which the fraction of unreliable Ct values is higher than a specified threshold:

```
# Finding samples with more than half of the unreliable Ct values.
low.quality.samples.pairwise <- filter(sample.Ct.control.pairwise[[2]],
                                       Not.reliable.fraction > 0.5)$Sample
low.quality.samples.pairwise <- as.vector(low.quality.samples.pairwise)
low.quality.samples.pairwise
#> character(0)
```

In the above example, there is no sample with more than half of the unreliable data.

```
# Finding genes with more than half of the unreliable Ct values in at least one group.
low.quality.genes.pairwise <- filter(gene.Ct.control.pairwise[[2]],
                                     Not.reliable.fraction > 0.5)$Gene
low.quality.genes.pairwise <- unique(as.vector(low.quality.genes.pairwise))
low.quality.genes.pairwise
#> [1] "Gene9"  "Gene2"  "Gene5"  "Gene11" "Gene1"
```

Five genes (Gene1, Gene2, Gene5, Gene9, and Gene11) have more than half of the unreliable Ct values in at least one group. In this example, these genes will be removed from the data in the next step of analysis.

In the `data.Ct.pairwise` data numbers of replicates in all samples are equal; therefore, the `control_heatmap()` function will not work here.

Filtering of raw Ct data (a pairwise approach)

When reliability criteria are finally established for Ct values, and some samples or genes are decided to be excluded from the analysis after quality control of the data, the data with raw Ct values can be filtered using the `filter_Ct()` function.

As a filtering criteria, a flag used for undetermined Ct values, a maximum of Ct threshold, and a flag used in Flag column can be applied. Furthermore, vectors with samples, genes, and groups to be removed can also be specified:

```
# Objects returned from the `low_quality_samples()` and
# `low_quality_genes()`functions can be used directly:
data.Ct.pairwiseF <- filter_Ct(data = data.Ct.pairwise,
                               flag.Ct = "Undetermined",
                               maxCt = 35,
                               flag = c("Undetermined"),
                               remove.Gene = low.quality.genes.pairwise,
                               remove.Sample = low.quality.samples.pairwise)

# Check dimensions of data before and after filtering:
dim(data.Ct.pairwise)
#> [1] 756   4
dim(data.Ct.pairwiseF)
#> [1] 530   4
```

NOTE: If data contain more than two groups, it is good practice to remove groups that are out of comparison; however, a majority of other functions can deal with more groups unless only two groups are indicated to be

given in function's parameters.

Collapsing technical replicates and imputation of missing data - `make_Ct_ready()` function (a pairwise approach)

In the next step, filtered Ct data can be subjected to collapsing of technical replicates and (optional) data imputation by means within groups using the `make_Ct_ready()` function. The term 'technical replicates' means observations with the same group name, gene name, and sample name. In the scenario when data contain technical replicates but they should not be collapsed, these technical replicates must be distinguished by different sample names, e.g. SampleA_1, SampleA_2, SampleA_3, etc.

The parameter `imput.by.mean.within.groups` can be used to control data imputation. If it is set to `TRUE`, imputation will be done, otherwise missing values will be left in the data. For a better view of the amount of missing values in the data, the information about the number and percentage of missing values is displayed automatically:

```
# Without imputation:
data.Ct.pairwiseF.ready <- make_Ct_ready(data = data.Ct.pairwiseF,
                                         imput.by.mean.within.groups = FALSE)
# A part of the data with missing values:
as.data.frame(data.Ct.pairwiseF.ready)[9:19,10:15]
#>    Gene19 Gene20  Gene3  Gene4  Gene7  Gene8
#> 9  28.246 33.375 31.086 22.900 32.242 23.070
#> 10 28.867 33.011 32.077 24.332 32.318 24.677
#> 11 29.951     NA 33.982 24.895     NA 23.973
#> 12 29.459 32.032 31.916 24.139 31.937 22.586
#> 13 29.018 34.362 33.497 24.937 32.363 23.714
#> 14 28.959     NA     NA 23.072 32.809 23.197
#> 15 28.763     NA 32.078 24.051 34.122 24.710
#> 16 27.325 29.509 32.185 21.328 28.986 19.841
#> 17 29.077 32.482 31.011 23.016 32.927 22.667
#> 18 30.415 31.895     NA 23.886 32.974 22.041
#> 19 31.039 34.386 33.739 24.505 33.868 24.492

# With imputation:
data.Ct.pairwiseF.ready <- make_Ct_ready(data = data.Ct.pairwiseF,
                                         imput.by.mean.within.groups = TRUE)
# Missing values were imputed:
as.data.frame(data.Ct.pairwiseF.ready)[9:19,10:15]
#>    Gene19   Gene20    Gene3  Gene4    Gene7  Gene8
#> 9  28.246 33.37500 31.08600 22.900 32.24200 23.070
#> 10 28.867 33.01100 32.07700 24.332 32.31800 24.677
#> 11 29.951 32.91194 33.98200 24.895 32.71465 23.973
#> 12 29.459 32.03200 31.91600 24.139 31.93700 22.586
#> 13 29.018 34.36200 33.49700 24.937 32.36300 23.714
#> 14 28.959 32.91194 32.19953 23.072 32.80900 23.197
#> 15 28.763 32.91194 32.07800 24.051 34.12200 24.710
#> 16 27.325 29.50900 32.18500 21.328 28.98600 19.841
#> 17 29.077 32.48200 31.01100 23.016 32.92700 22.667
#> 18 30.415 31.89500 32.19953 23.886 32.97400 22.041
#> 19 31.039 34.38600 33.73900 24.505 33.86800 24.492
```

NOTE: The data imputation process can significantly influence data; therefore, no default value was set to the `imput.by.mean.within.groups` parameter to force the specification by the user. If there are missing data for a certain gene in the entire group, they will not be imputed and will remain NA.

In general, a majority of functions in `RQdeltaCT` package can deal with missing data; however, some methods (e.g. PCA) are sensitive to missing data (see PCA analysis section).

NOTE: The `make_Ct_ready()` function should be used even if the collapsing of technical replicates and data imputation is not required, because this function also prepares the data structure to fit to further functions.

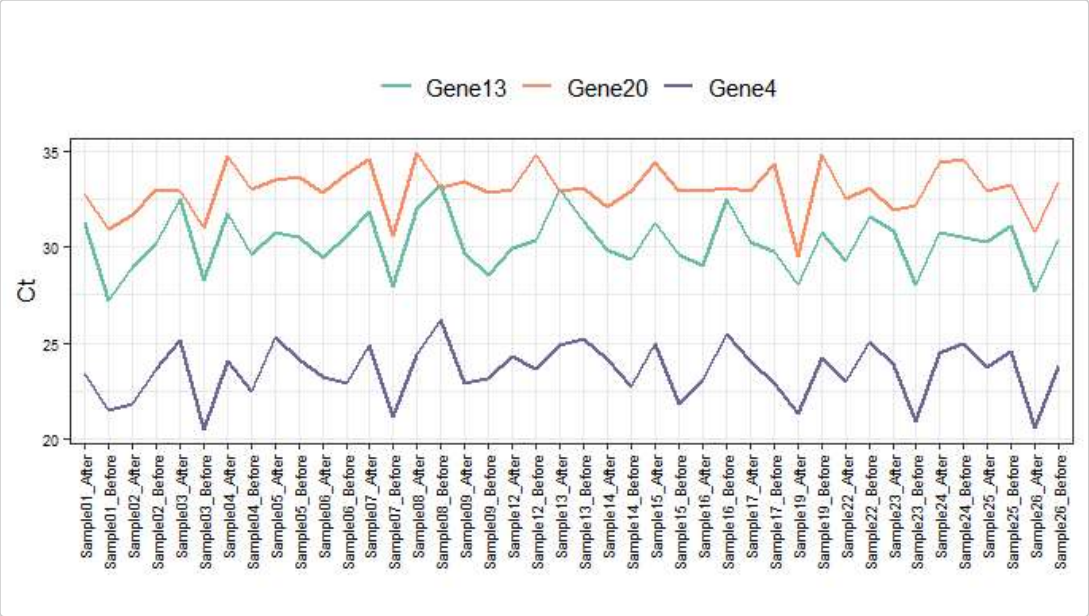
Reference gene selection (a pairwise approach)

The `RQdeltaCT` package includes `find_ref_gene()` function that can be used to select the best reference gene for normalisation. This function calculates descriptive statistics such as minimum, maximum, standard deviation, and variance, as well as stability scores calculated using the NormFinder (Article) and geNorm (Article) algorithms. Ct values are also presented on a line plot.

NormFinder scores are computed using internal `RQdeltaCT::norm_finder()` function working on the code adapted from the original NormFinder code. To calculate NormFinder scores, at least two samples must be present in each group. For the geNorm score, the `geNorm()` function of the `ctrlGene` package is used. The `find_ref_gene()` function allows one to choose which of these algorithms should be done by setting the logical parameters: `norm.finder.score` and `genorm.score`.

The created plot is displayed on the graphic device. The returned object is a list that contains two elements: an object with plot and a table with results. In the example below, three genes are tested for suitability to be a reference gene:

```
library(ctrlGene)
# Remember that the number of colors in col parameter should be equal to the number of tested genes:
ref.pairwise <- find_ref_gene(data = data.Ct.pairwiseF.ready,
                      groups = c("After","Before"),
                      candidates = c("Gene4","Gene13","Gene20"),
                      col = c("#66c2a5", "#fc8d62","#6A6599"),
                      angle = 90,
                      axis.text.size = 7,
                      norm.finder.score = TRUE,
                      genorm.score = TRUE)
```




```
ref.pairwise[[2]]
#>           Gene    min    max       sd      var NormFinder_score geNorm_score
#> 1       Gene13 27.184 33.224 1.449397 2.100751             0.17           NA
#> 2       Gene20 29.509 34.871 1.223547 1.497068             0.26    0.9961593
#> 3        Gene4 20.499 26.228 1.418726 2.012784             0.15           NA
#> 4 Gene4-Gene13     NA     NA       NA       NA               NA    0.6912531
```

NA values are caused because geNorm method returns a pair of genes with the highest stability.
Among tested genes, Gene4 seems to have the best characteristics to be a reference gene (it has Ct values below 30, relatively low standard deviation, variance, NormFinder score, and geNorm score).

Data normalization using reference gene (a pairwise approach)

Data normalization can be performed using `delta_Ct()` function that calculates delta Ct (dCt) values by subtracting Ct values of reference gene (or mean of the Ct values of reference genes, if more than one reference gene is used) from Ct values of gene of interest across all samples.

Normalization without transformation (for 2^{-ddCt} method purposes):

```
data.dCt.pairwise <- delta_Ct(data = data.Ct.pairwiseF.ready,
                              ref = "Gene4",
```

```
                                transform = FALSE)
```

If 2^{-dCt} transformation of normalized data is needed (2^{-dCt} method), the `transform` parameter should be set to `TRUE`:

```
data.dCt.exp.pairwise <- delta_Ct(data = data.Ct.pairwiseF.ready,
                                  ref = "Gene4",
                                  transform = TRUE)
```

Before further processing, non-transformed or transformed dCt data should be subjected to quality control using functions and methods described in Quality control and filtering of normalized Ct data (a pairwise approach).

Relative quantification: 2^{-dCt} method (a pairwise approach)

This method is used in studies where samples should be analysed as individual data points, e.g. in analysis of biological replicates (see [Introduction] section). In the pairwise variant of this method, Ct values are normalised by the endogenous control gene by subtracting the Ct value of the endogenous control from the Ct value of the gene of interest, in the same sample. Obtained delta Ct (dCt) values are subsequently transformed using the 2^{-dCt} formula, and a ratio (fold change) of transformed Ct values is calculated individually for each pair of samples, and summarised by mean for each gene.

The whole process can be done using `RQ_dCt()` function, which performs:
+ calculation of means (returned in columns with the "_mean" pattern) and standard deviations (returned in columns with the "_sd" pattern) of transformed dCt values of genes analysed in the compared groups. + normality testing (Shapiro_Wilk test) of transformed dCt values of analysed genes in compared groups and returned p values are stored in columns with the "_norm_p" pattern. + calculation of the fold change values. In this pairwise approach (when `pairwise = TRUE`), individual fold change values are firstly calculated for each sample by dividing transformed Ct value obtained for particular gene in the study group by transformed Ct value obtained for the same gene in the reference group. Subsequently, mean and standard deviation values of individual fold changes are calculated within each group (returned in FCh_mean and FCh_sd columns, respectively). + statistical testing of differences in transformed Ct values between the study group and the reference group. Student's t test and Mann-Whitney U test are implemented and the resulting statistics (in column with the "_test_stat" pattern), p values (in column with the "_test_p" pattern), and adjusted p values (in column with the "_test_p_adj" pattern) are returned. The Benjamini-Hochberg method for adjustment of p values is used in default. If compared groups contain less than three samples, normality and statistical tests are not possible to perform (the `do.test` parameter should be set to `FALSE` to avoid error).

NOTE: For this method, delta Ct (dCt) values transformed by the 2^{-dCt} formula using `delta_Ct()` function (called with `transform = TRUE`) should be used.

```
data.dCt.pairwise <- delta_Ct(data = data.Ct.pairwiseF.ready,
                              ref = "Gene4",
                              transform = FALSE)
library(coin)
results.dCt.pairwise <- RQ_dCt(data = data.dCt.pairwise,
                               do.tests = TRUE,
                               pairwise = TRUE,
                               group.study = "After",
                               group.ref = "Before")

# Obtained table can be sorted by, e.g. p values from the Mann-Whitney U test:
results <- as.data.frame(arrange(results.dCt.pairwise[[1]], MW_test_p))
head(results)
#>     Gene After_mean Before_mean  After_sd Before_sd After_norm_p Before_norm_p
#> 1 Gene19  5.2109333   3.7824762 0.8428066 0.9893666   0.43685704    0.03510469
#> 2  Gene8 -0.3372381  -1.0184286 0.8749796 1.0425276   0.04748179    0.17942875
#> 3 Gene16 -0.2142381  -0.7020476 0.7958921 0.6781031   0.19024540    0.07832005
#> 4 Gene14  6.0329048   5.7692381 0.5984831 0.6675626   0.58378047    0.03738612
#> 5  Gene7  9.0139833   8.4662381 0.8176409 1.3667726   0.01484502    0.69000384
#> 6 Gene20  9.2112745   9.6542381 0.9144846 1.2836473   0.17140590    0.21430743
#>         FCh    log10FCh    FCh_sd     t_test_p t_test_stat  MW_test_p
#> 1 1.4700335  0.16732722 0.4499545 0.0001845345    4.573104 0.00102128
#> 2 0.2043090 -0.68971252 2.7242366 0.0537860813    2.049238 0.04565545
#> 3 1.9663253  0.29365536 6.4052867 0.0715023737    1.903242 0.05372471
#> 4 1.0595285  0.02511265 0.1643101 0.2140118947    1.283435 0.24426953
```

```
#> 5 1.0967073  0.04009074 0.2375549 0.1444388848    1.518898 0.24426953
#> 6 0.9765855 -0.01028974 0.2018320 0.2733067651   -1.126458 0.41404000
#>   MW_test_stat t_test_p_adj MW_test_p_adj
#> 1    3.2845979  0.002214414    0.01225536
#> 2    1.9985649  0.286009495    0.21489883
#> 3    1.9290496  0.286009495    0.21489883
#> 4    1.1643813  0.513628547    0.58624688
#> 5    1.1643813  0.433316654    0.58624688
#> 6   -0.8168048  0.546613530    0.82808001
# Access to the table with fold change values calculated individually for each pair of sampleS:
FCh <- results.dCt.pairwise[[2]]
head(FCh)
#> # A tibble: 6 × 5
#>   Sample   Gene    After Before   FCh
#>   <chr>    <chr>   <dbl>  <dbl> <dbl>
#> 1 Sample01 Gene10  3.14   2.78  1.13
#> 2 Sample01 Gene13  7.81   5.67  1.38
#> 3 Sample01 Gene14  6.49   5.86  1.11
#> 4 Sample01 Gene15  6.63   6.18  1.07
#> 5 Sample01 Gene16 -0.724 -0.882 0.821
#> 6 Sample01 Gene17  2.86   3.47  0.824
```

NOTE: In a pairwise approach (if `pairwise = TRUE`), the abovementioned results can be found in the first element of a list object returned by `RQ_dCt()` function. For the results transparency, fold change values calculated individually for each sample pair are also returned as the second element of this object, and can be used to assess the homogeneity of individual fold change values within each analysed gene using methods described in [Quality control and filtering of normalized Ct data - a pairwise approach] section.

Relative quantification: 2^{-ddCt} method (a pairwise approach)

Similarly to the 2^{-dCt} method, in the 2^{-ddCt} method Ct values are normalised by the endogenous control gene, obtaining dCt values. Subsequently, individually for each pair of samples, dCt values obtained in the control group are subtracted from the dCt values in the study group, giving individual delta delta Ct (ddCt) values. Finally, the ddCt values are transformed using the 2^{-ddCt} formula to obtain the fold change values, which then are summarised by mean across analysed genes.

The whole process can be done using `RQ_ddCt()` function, which performs:
+ calculation of means (returned in columns with the "_mean" pattern) and standard deviations (returned in columns with the "_sd" pattern) of delta Ct values of the analyzed genes in the compared groups. + normality testing (Shapiro_Wilk test) of delta Ct values of the analyzed genes in the compared groups and returned p values are stored in columns with the "_norm_p" pattern. + calculation of differences (delta delta Ct, ddCt values) in the delta Ct values between paired samples by subtracting dCt values obtained in the control group from the dCt values in the study group. + calculation of fold change values using 2^{-ddCt} formula. Fold change values are returned in a separated table (see NOTE below). Subsequently, means (returned in "FCh" column) and standard deviations (returned in "FCh_sd" column) of fold change values are calculated for each gene. + a pairwise statistical testing of differences between the compared groups. A pairwise Student's t test and Mann-Whitney U test are implemented and the resulted statistics (in column with the "_test_stat" pattern), p values (in column with the "_test_p" pattern) and adjusted p values (in column with the "_test_p_adj" pattern) are returned. The Benjamini-Hochberg method for adjustment of p values is used in default. If compared groups contain less than three samples, normality and statistical tests are not possible to perform and `do.test` parameter should be set to FALSE to avoid error.

NOTE: For this method, not transformed delta Ct (dCt) values (obtained from `delta_Ct()` function with `transform = FALSE`) should be used.

```
data.dCt.pairwise <- delta_Ct(data = data.Ct.pairwiseF.ready,
                              ref = "Gene4",
                              transform = FALSE)
library(coin)
# Remember to set pairwise = TRUE:
results.ddCt.pairwise <- RQ_ddCt(data = data.dCt.pairwise,
                                 group.study = "After",
                                 group.ref = "Before",
                                 pairwise = TRUE,
                                 do.tests = TRUE)

# Obtained table can be sorted by, e.g. p values from the Mann-Whitney U test:
```

```
results <- as.data.frame(arrange(results.ddCt.pairwise[[1]], MW_test_p))
head(results)
#>     Gene After_mean Before_mean  After_sd Before_sd After_norm_p Before_norm_p
#> 1 Gene19  5.2109333   3.7824762 0.8428066 0.9893666   0.43685704    0.03510469
#> 2  Gene8 -0.3372381  -1.0184286 0.8749796 1.0425276   0.04748179    0.17942875
#> 3 Gene16 -0.2142381  -0.7020476 0.7958921 0.6781031   0.19024540    0.07832005
#> 4 Gene14  6.0329048   5.7692381 0.5984831 0.6675626   0.58378047    0.03738612
#> 5  Gene7  9.0139833   8.4662381 0.8176409 1.3667726   0.01484502    0.69000384
#> 6 Gene20  9.2112745   9.6542381 0.9144846 1.2836473   0.17140590    0.21430743
#>         FCh     log10FCh    FCh_sd     t_test_p t_test_stat  MW_test_p
#> 1 0.6485663 -0.188045654 0.9753027 0.0001845345    4.573104 0.00102128
#> 2 1.1687779  0.067731984 1.7391643 0.0537860813    2.049238 0.04565545
#> 3 0.9811133 -0.008280834 0.9145980 0.0715023737    1.903242 0.05372471
#> 4 1.0206707  0.008885637 0.6804825 0.2140118947    1.283435 0.24426953
#> 5 1.1322906  0.053957891 1.0765954 0.1444388848    1.518898 0.24426953
#> 6 2.5222676  0.401791166 2.6165869 0.2733067651   -1.126458 0.41404000
#>   MW_test_stat t_test_p_adj MW_test_p_adj
#> 1    3.2845979  0.002214414    0.01225536
#> 2    1.9985649  0.286009495    0.21489883
#> 3    1.9290496  0.286009495    0.21489883
#> 4    1.1643813  0.513628547    0.58624688
#> 5    1.1643813  0.433316654    0.58624688
#> 6   -0.8168048  0.546613530    0.82808001
# Access to the table with fold change values calculated individually for each pair of samples:
FCh <- results.ddCt.pairwise[[2]]
head(FCh)
#> # A tibble: 6 × 5
#>   Sample   Gene    After Before   FCh
#>   <chr>    <chr>   <dbl>  <dbl> <dbl>
#> 1 Sample01 Gene10  3.14   2.78  0.783
#> 2 Sample01 Gene13  7.81   5.67  0.227
#> 3 Sample01 Gene14  6.49   5.86  0.647
#> 4 Sample01 Gene15  6.63   6.18  0.732
#> 5 Sample01 Gene16 -0.724 -0.882 0.896
#> 6 Sample01 Gene17  2.86   3.47  1.53
```

NOTE: In a pairwise approach (if `pairwise = TRUE`), the results can be found in the first element of a list object returned by `RQ_ddCt()` function. For the results transparency, fold change values calculated individually for each sample pair can be found in the second element of this object, and can be assessed using methods described in the [Quality control and filtering of normalized Ct data - a pairwise approach] section.

Quality control and filtering of normalized Ct data (a pairwise approach)

The `RQdeltaCT` package offers several functions which facilitate finding outlier samples in data by implementing distribution analysis (`control_boxplot_sample()` function), hierarchical clustering (`control_cluster_sample()` function) and principal component analysis PCA (`control_pca_sample()` function). However, corresponding functions for genes are also provided to evaluate similarities and differences between expression of analysed genes (`control_boxplot_gene()`, `control_cluster_gene()` and `control_pca_gene()` functions).

The abovementioned data quality control functions are designed to be directly applied to data objects returned from the `make_Ct_ready()` and `delta_Ct()` functions (see The summary of standard workflow). Moreover, in a pairwise approach, a tables with fold change values obtained from individual pairs of samples (the second element of list returned from `RQ_dCt()` and `RQ_ddCt()` functions) can be also passed directly, but the `pairwise.FCh` parameter of control functions must be set to `TRUE`.

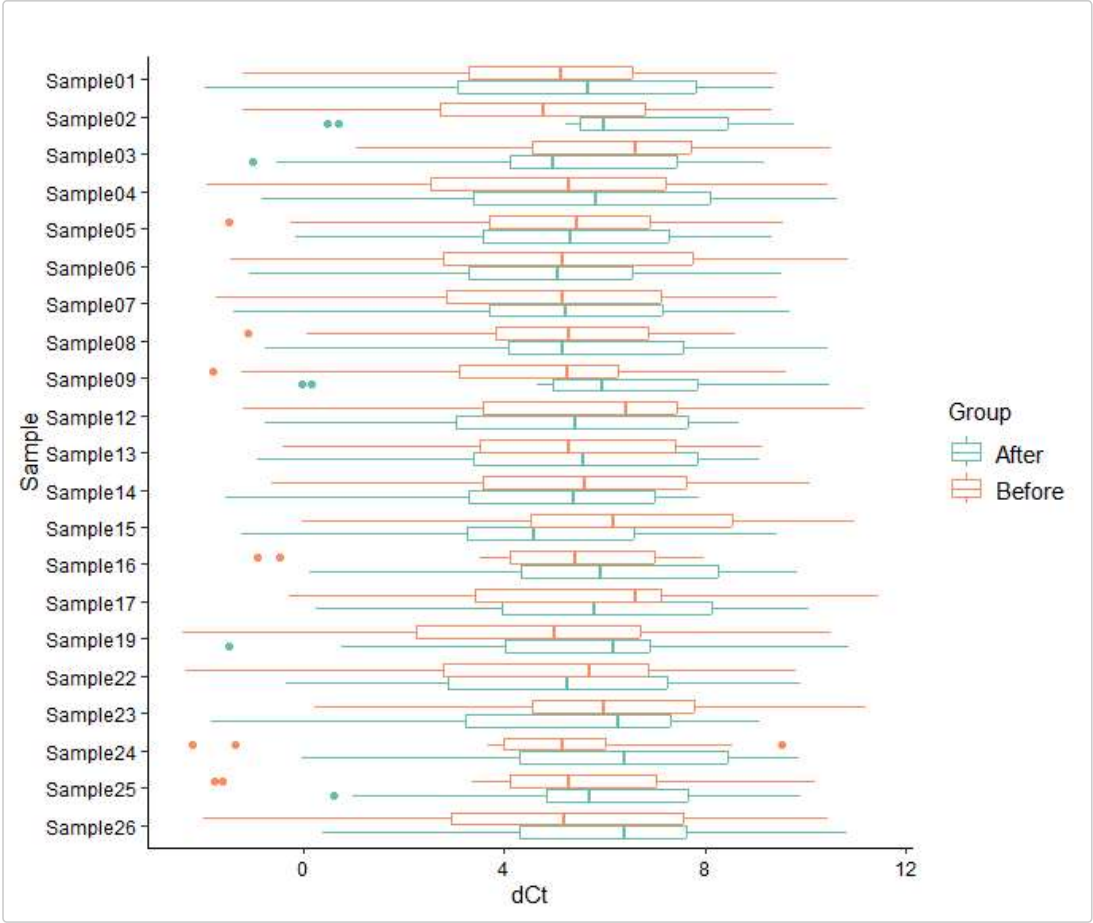
Analysis of data distribution (a pairwise approach)

In the `RQdeltaCT` package, the `control_boxplot_sample()` and `control_boxplot_gene()` functions are included to draw boxplots that present the distribution of samples and genes, respectively.

These functions return objects with a plot. The created plots are also displayed on the graphic device.

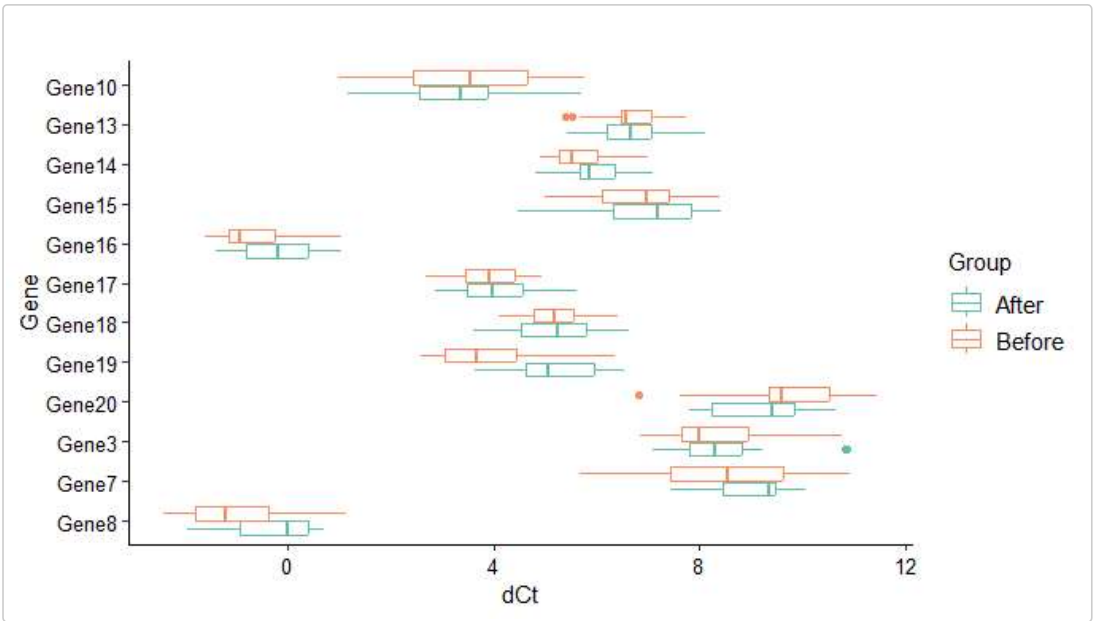
Boxplots for samples:

```
control_boxplot_sample <- control_boxplot_sample(data = data.dCt.pairwise,
                                                 axis.text.size = 9,
                                                 y.axis.title = "dCt")
```



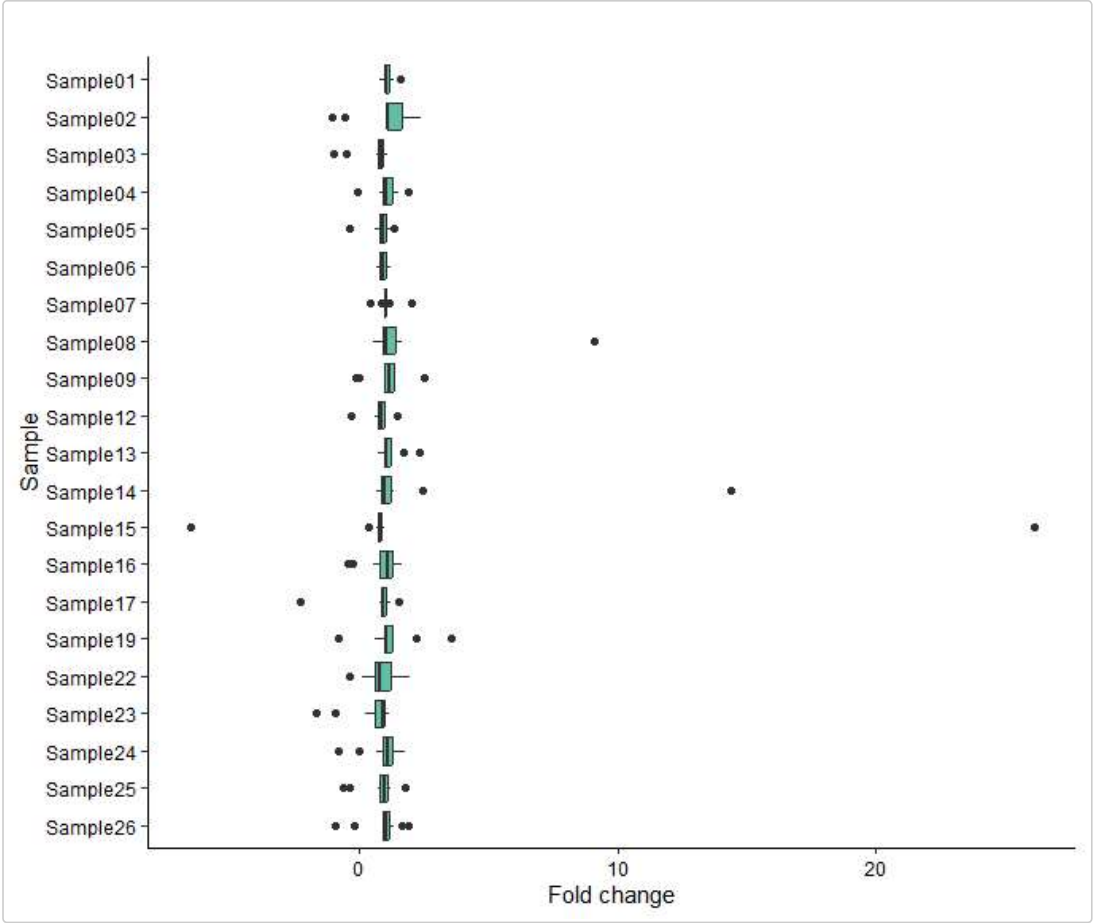
And boxplots for genes:

```
control_boxplot_gene <- control_boxplot_gene(data = data.dCt.pairwise,
                                             by.group = TRUE,
                                             axis.text.size = 10,
                                             y.axis.title = "dCt")
```



Similar plots can be created for fold change values data returned from `RQ_dCt()` and `RQ_ddCt()` functions:

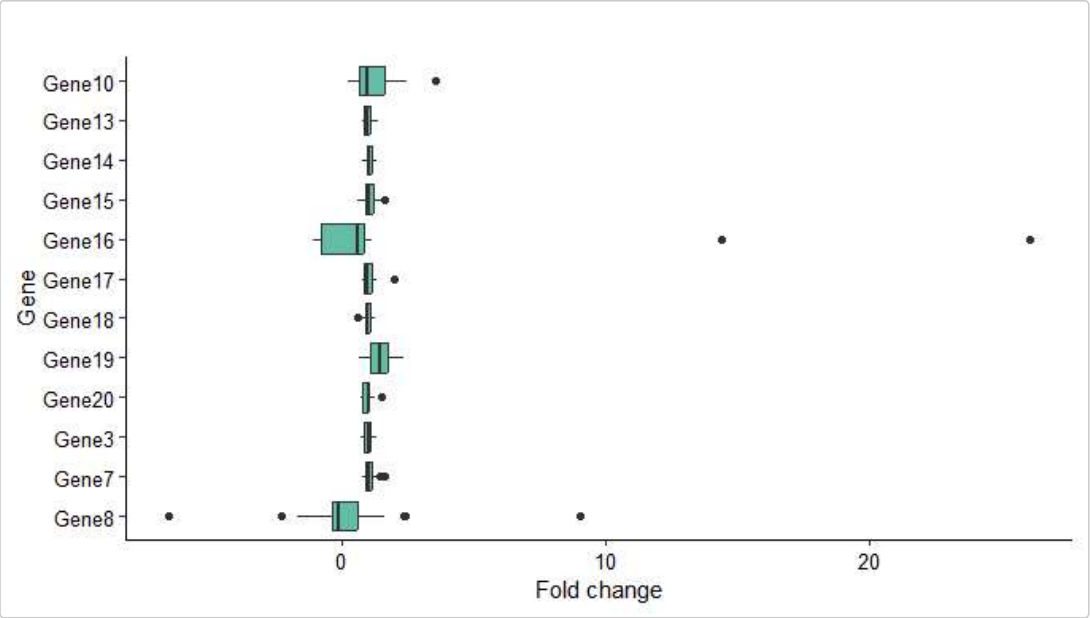
```
# Remember to set pairwise.FCh to TRUE:
FCh <- results.dCt.pairwise[[2]]
control.boxplot.sample.pairwise <- control_boxplot_sample(data = FCh,
                                                          pairwise.FCh = TRUE,
                                                          axis.text.size = 9,
                                                          y.axis.title = "Fold change")
```




```
# There are some very high values, we can identify them using:
head(arrange(FCh, -FCh))
#> # A tibble: 6 × 5
#>   Sample   Gene    After  Before   FCh
#>   <chr>    <chr>   <dbl>   <dbl> <dbl>
#> 1 Sample15 Gene16 -0.966 -0.0370 26.1
#> 2 Sample14 Gene16 -1.40  -0.0970 14.4
#> 3 Sample08 Gene8   0.554  0.0610  9.08
#> 4 Sample19 Gene10  3.46   0.969   3.57
#> 5 Sample09 Gene10  5.08   2.03    2.50
#> 6 Sample14 Gene8  -1.55  -0.634   2.45
```

Similar visualisation can be done for genes:

```
control.boxplot.gene.pairwise <- control_boxplot_gene(data = FCh,
                                                      by.group = FALSE,
                                                      pairwise.FCh = TRUE,
                                                      axis.text.size = 10,
                                                      y.axis.title = "Fold change")
```

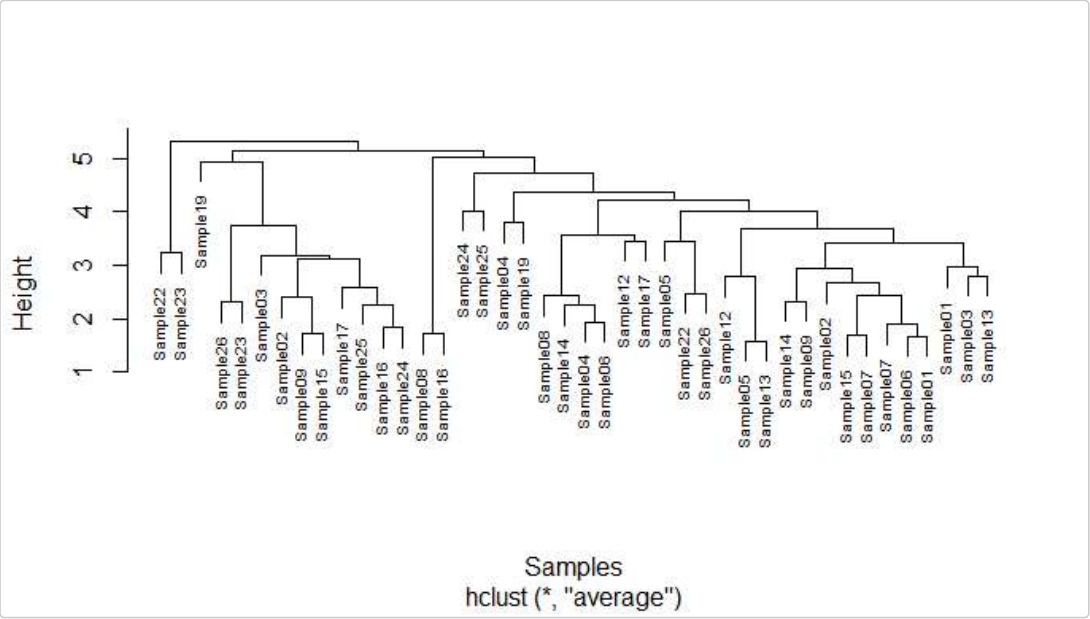


NOTE: If missing values are present in the data, they will be automatically removed with warning.

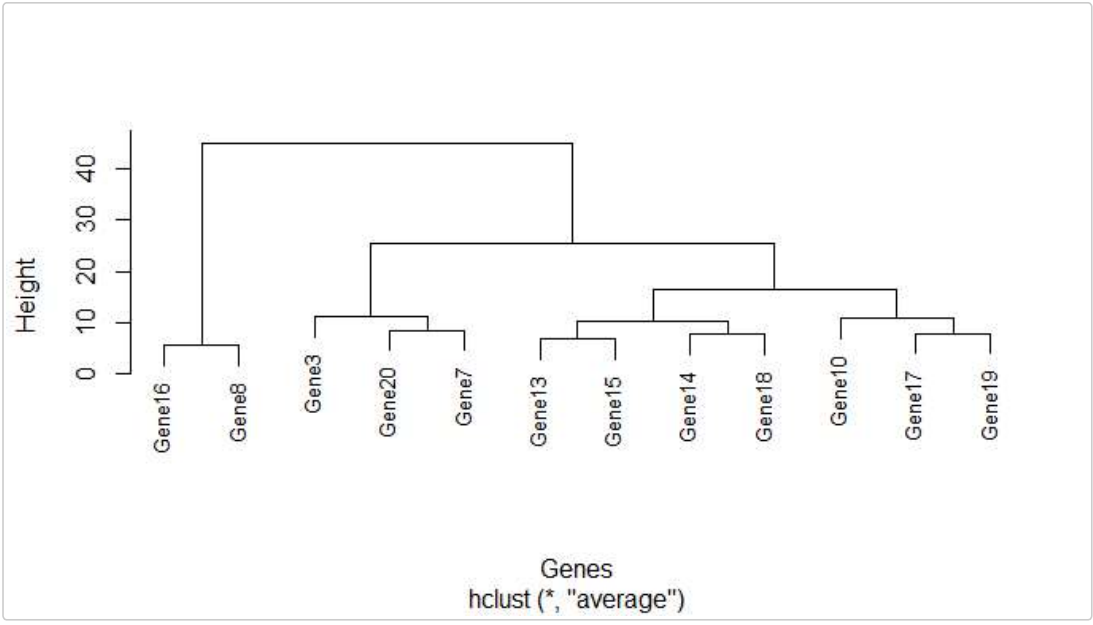
Hierarchical clustering (a pairwise approach)

Hierarchical clustering of samples and genes can be done using the `control_cluster_sample()` and `control_cluster_gene()` functions, respectively. These functions allow to draw dendrograms using various methods of distance calculation (e.g. euclidean, canberra) and agglomeration (e.g. complete, average, single). For more details, refer to the functions documentation.

```
# Hierarchical clustering of samples:
control_cluster_sample(data = data.dCt.pairwise,
                       method.dist = "euclidean",
                       method.clust = "average",
                       label.size = 0.6)
```

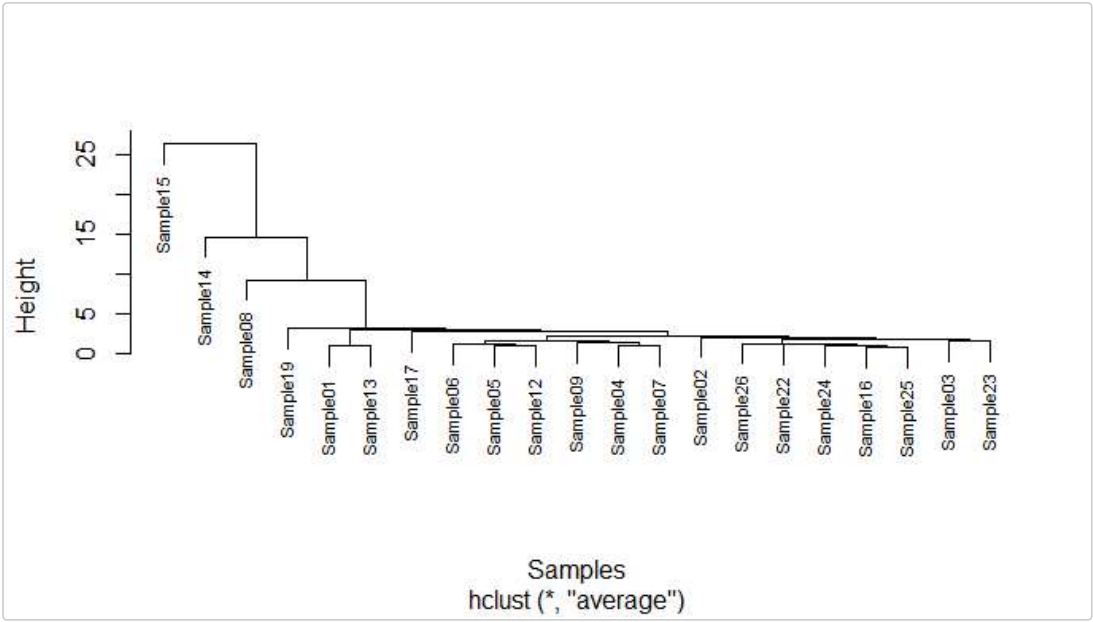



```
# Hierarchical clustering of genes:
control_cluster_gene(data = data.dCt.pairwise,
                     method.dist = "euclidean",
                     method.clust = "average",
                     label.size = 0.8)
```

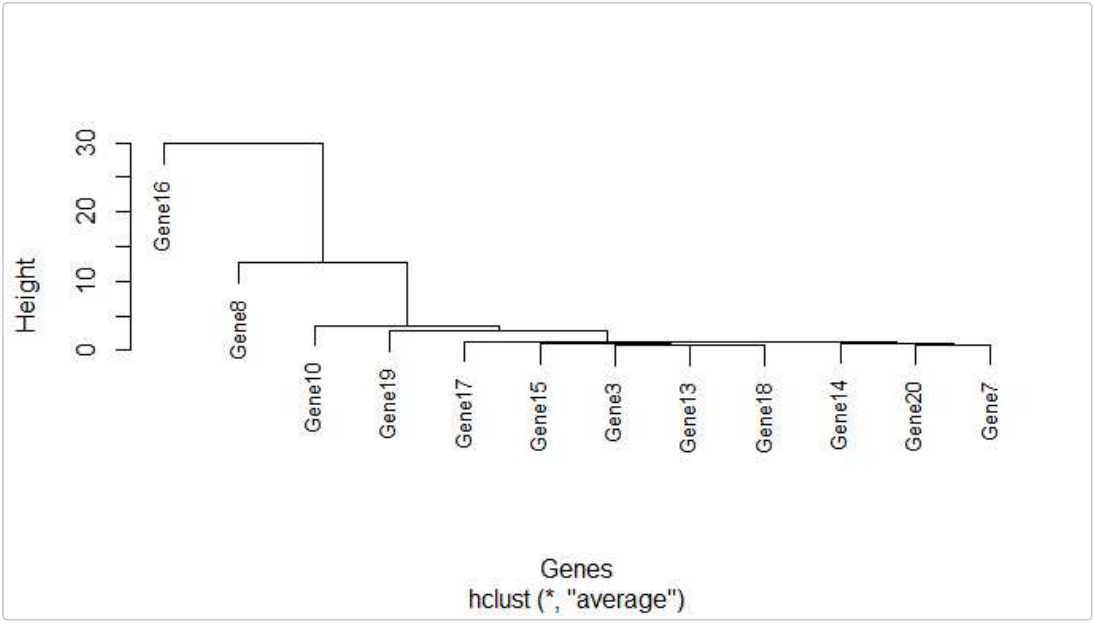


Similar plots can be created for fold change values data returned from `RQ_dCt()` and `RQ_ddCt()` functions:

```
# Remember to set pairwise.FCh = TRUE:
control_cluster_sample(data = FCh,
                       pairwise.FCh = TRUE,
                       method.dist = "euclidean",
                       method.clust = "average",
                       label.size = 0.7)
```




```
control_cluster_gene(data = FCh,
                     pairwise.FCh = TRUE,
                     method.dist = "euclidean",
                     method.clust = "average",
                     label.size = 0.8)
```



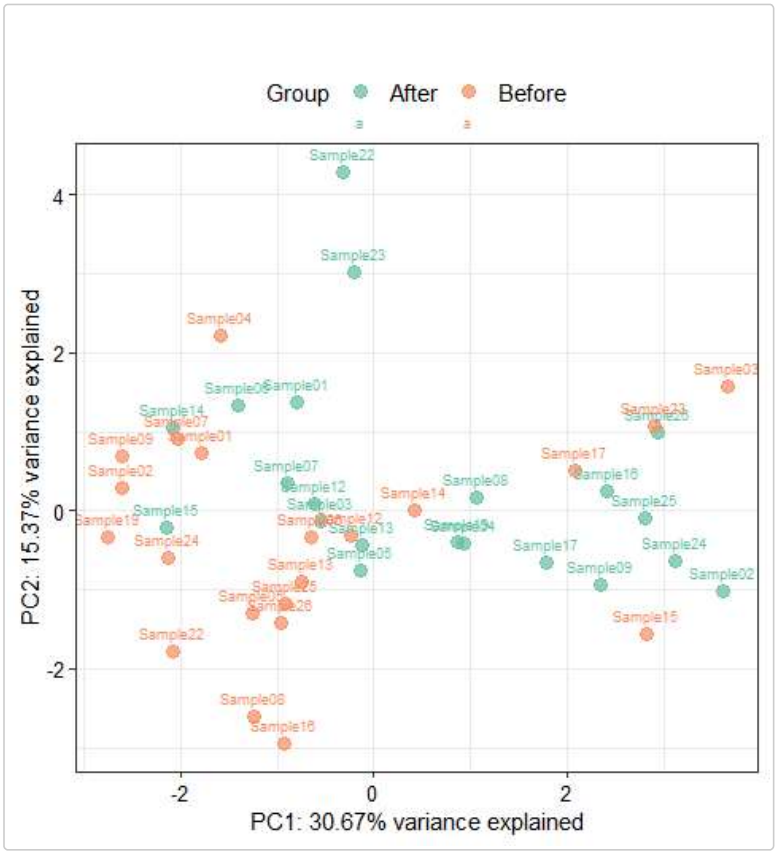
The created plots are displayed on the graphic device.

NOTE: Minimum three samples or genes in data is required for clustering analysis.

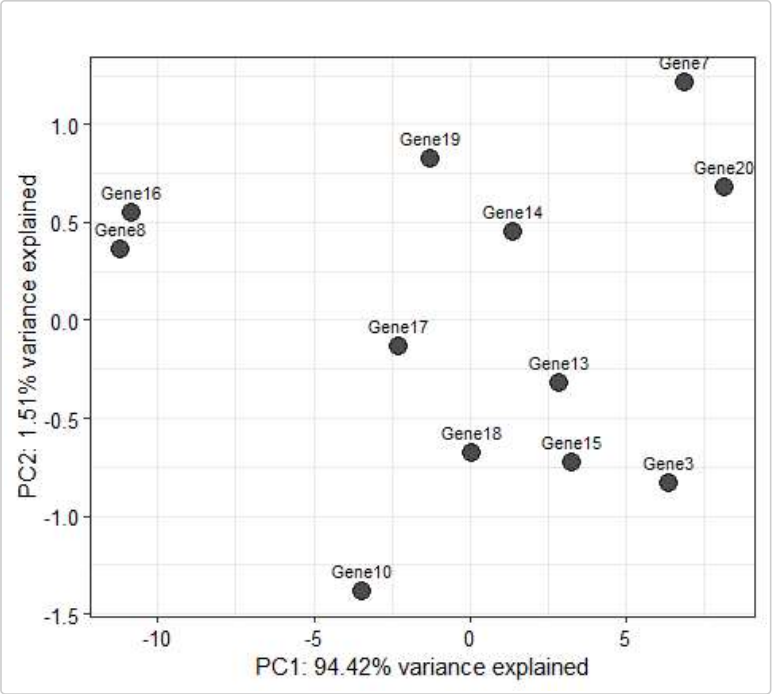
PCA analysis (a pairwise approach)

In the `RQdeltaCT` package, the `control_pca_sample()` and `control_pca_gene()` functions are developed to perform PCA analysis for samples and genes, respectively. These functions return objects with a plot. Created plots are also displayed on the graphic device.

```
control.pca.sample.pairwise <- control_pca_sample(data = data.dCt.pairwise,
                                                  point.size = 3,
                                                  label.size = 2.5,
                                                  hjust = 0.5,
                                                  legend.position = "top")
```

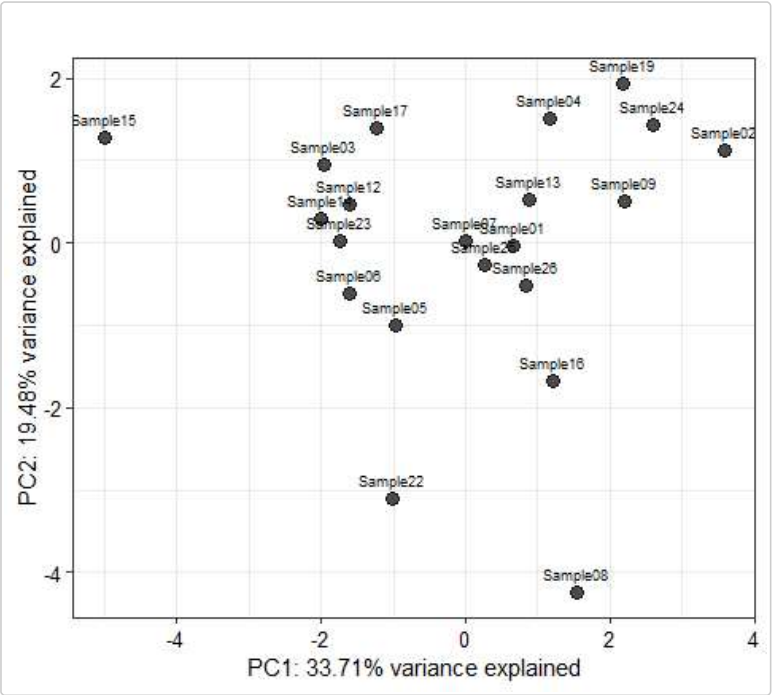



```
control.pca.gene.pairwise <- control_pca_gene(data = data.dCt.pairwise,
                                              hjust = 0.5)
```

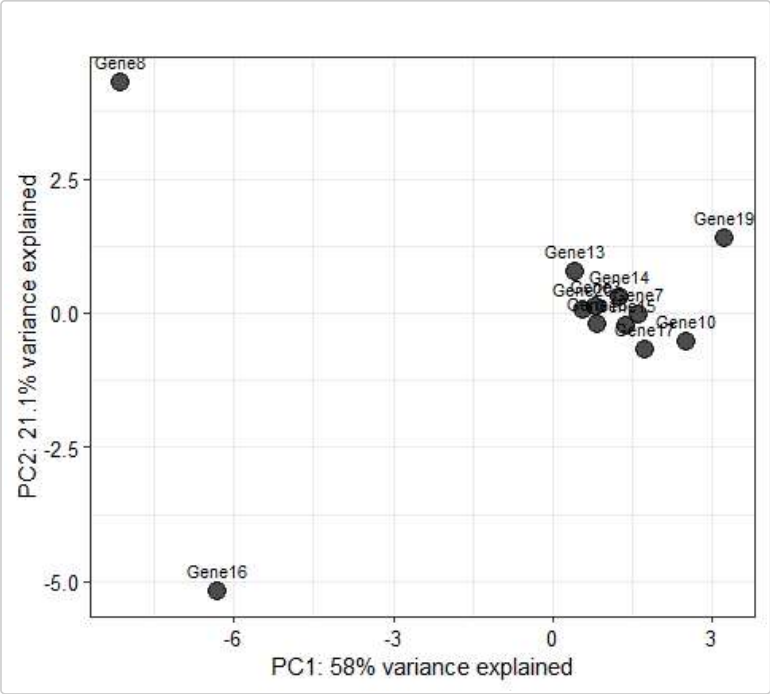


Similar plots can be created for fold change values data returned from `RQ_dCt()` and `RQ_ddCt()` functions:

```
control.pca.sample.pairwise <- control_pca_sample(data = FCh,
                                                  pairwise.FCh = TRUE,
                                                  colors = "black",
                                                  point.size = 3,
                                                  label.size = 2.5,
                                                  hjust = 0.5)
```




```
control.pca.gene.pairwise <- control_pca_gene(data = FCh,
                                              pairwise.FCh = TRUE,
                                              color = "black",
                                              hjust = 0.5)
```



NOTE: PCA algorithm can not deal with missing data (NAs); therefore, variables with NA values are removed before analysis. If at least one NA value occurs in all variables in at least one of the compared group, the analysis can not be done. Imputation of missing data will avoid this issue. Also, a minimum of three samples or genes in the data are required for analysis.

Data filtering after quality control (a pairwise approach)

If any sample or gene was decided to be removed from the data, the `filter_transformed_data()` function can be used for filtering. Similarly to quality control functions, this function can be directly applied to data objects returned from the `make_Ct_ready()` and `delta_Ct()` functions (see The summary of standard workflow).

```
data.dCt.pairwise.F <- filter_transformed_data(data = data.dCt.pairwise,
                                               remove.Sample = c("Sample22",
                                                                 "Sample23",
                                                                 "Sample15",
                                                                 "Sample03"))
dim(data.dCt.pairwise)
#> [1] 42 14
dim(data.dCt.pairwise.F)
#> [1] 34 14
```

NOTE: Remember, there must be a good objective reason to remove specific samples.

Final visualisations (a pairwise approach)

For visualisation of final results, these five functions can be used:

- `FCh_plot()` that allows to illustrate fold change values of genes,
- `results_volcano()` that allows to create volcano plot presenting the arrangement of fold change values and p values,
- `results_barplot()` that show mean and standard deviation values of genes across the compared groups,
- `results_boxplot()` that illustrate the data distribution of genes across the compared groups,
- `results_heatmap()` that allows to create heatmap with hierarchical clustering,
- `parallel_plot()` that allows to present a pairwise changes in genes expression.

All these functions can be run on all data or on finally selected genes (see `sel.Gene` parameter). These functions have a large number of parameters, and the user should familiarise with all of them to properly adjust created plots to the user needs.

NOTE: The functions `FCh_plot()`, `results_barplot()`, and `results_boxplot()` also allow to add customised statistical significance labels to plots using `ggsignif` package. If statistical significance labels should be added to the plot, a vector with labels (e.g., “ns”, “*“,”p = 0.03”) should be provided in the `signif.labels` parameter. There are two important points that must be taking into account when preparing this vector:

- The order of labels should correspond to the order of genes presented on the plot, not the order of genes in the data, which can be different.
- In this vector, due to restrictions of the `ggsignif` package, all values must be different (the same values are not allowed). Thus, if the same labels are needed, they should be distinguishable by adding symmetrically a different number of white spaces (see the examples below).

```
# Remember to set pairwise = TRUE:
results.ddCt.pairwise <- RQ_ddCt(data = data.dCt.pairwise.F,
                  group.study = "After",
                  group.ref = "Before",
                  pairwise = TRUE,
                  do.tests = TRUE)

# Obtained table can be sorted by, e.g. p values from the Mann-Whitney U test:
results <- as.data.frame(arrange(results.ddCt.pairwise[[1]], MW_test_p))
head(results)
#>     Gene After_mean Before_mean  After_sd Before_sd After_norm_p Before_norm_p
#> 1 Gene19  5.2123294   3.5095882 0.7641196 0.6450896   0.76631561    0.37520243
#> 2  Gene8 -0.1840000  -1.2615882 0.8593104 0.7335902   0.01848096    0.57804549
#> 3 Gene16 -0.1601765  -0.8746471 0.8306935 0.5154298   0.23806535    0.28735006
#> 4 Gene15  7.2110588   6.5512941 0.9833445 0.9034758   0.10927151    0.05594945
#> 5  Gene7  9.0409206   8.3574706 0.7906209 1.3700506   0.02375954    0.98526475
#> 6 Gene17  4.0798824   3.7351176 0.7764462 0.6373618   0.72685827    0.90708293
#>         FCh     log10FCh    FCh_sd     t_test_p t_test_stat    MW_test_p
#> 1 0.4078506 -0.389498890 0.3354031 1.134508e-05    6.261821 0.0005026301
#> 2 0.6323613 -0.199034707 0.5379766 9.009319e-04    4.064665 0.0035993566
#> 3 0.7577723 -0.120461285 0.5356882 1.041055e-02    2.901408 0.0056180461
#> 4 1.1067408  0.044045931 1.4292426 9.330078e-02    1.784581 0.0928595314
#> 5 0.9901632 -0.004293203 0.9777963 8.966683e-02    1.806604 0.1127786546
#> 6 0.9001501 -0.045685066 0.4176108 1.174977e-01    1.654544 0.1239292250
#>   MW_test_stat t_test_p_adj MW_test_p_adj
#> 1     3.479351  0.000136141   0.006031561
#> 2     2.911294  0.005405591   0.021596140
#> 3     2.769279  0.041642189   0.022472184
#> 4     1.680503  0.223921872   0.247858450
#> 5     1.585827  0.223921872   0.247858450
#> 6     1.538488  0.224668939   0.247858450
```

Three genes (Gene8, Gene16, and Gene19) seem to have statistically significant differences in expression between Before and After groups

The `FCh_plot()` function (a pairwise approach)

This function creates a barplot that illustrates fold change values obtained from the analysis, together with an indication of statistical significance. The first element of the object returned from the `RQ_dCt()` and `RQ_ddCt()` functions worked in a pairwise mode (when `pairwise = TRUE`) can be directly applied to this function (see The summary of standard workflow).

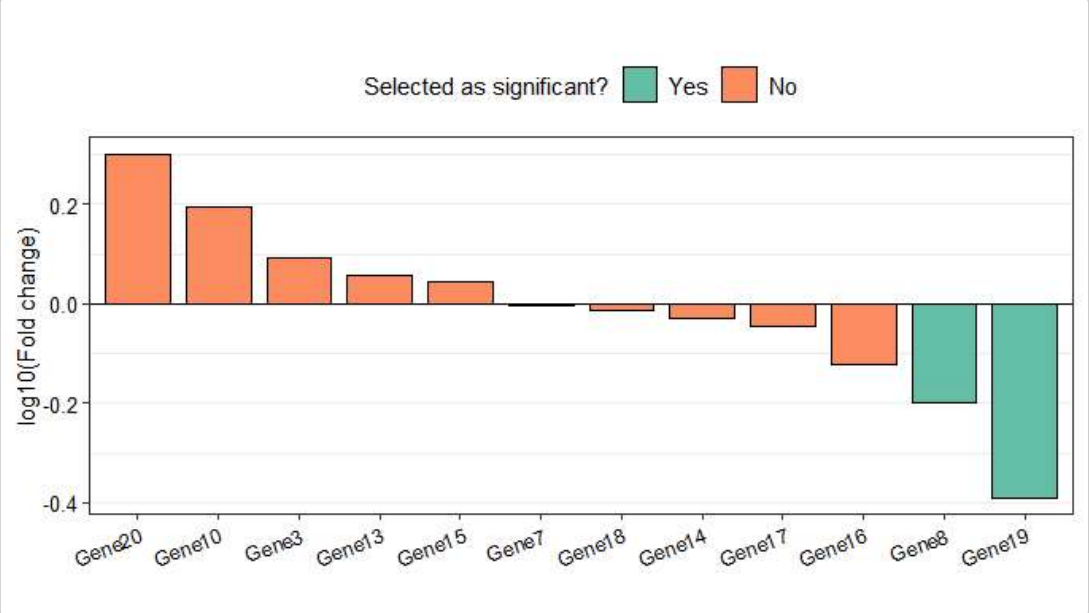
On the barplot, bars of significant genes are distinguished by colors and/or significance labels. The significance of genes can be established by two criteria: p values and (optionally) fold change values. Thresholds for both criteria can be specified. The `FCh_plot()` function offers various options of which p values are used on the plot:

- p values from the Student's t test (if `mode = "t"`) or adjusted p values from the Student's t test (if `mode = "t.adj"`).
- p values from the Mann-Whitney U test (if `mode = "mw"`) or adjusted p values from the Mann-Whitney U test (if `mode = "mw.adj"`)
- p values depend on the normality of data (if `mode = "depends"`). If the data in both compared groups were considered derived from normal distribution (p value of Shapiro_Wilk test > 0.05) - p values of Student's t test will be used, otherwise p values of Mann-Whitney U test will be used. For adjusted p values, use `mode = "depends.adj"`.
- external p values provided by the user (if `mode = "user"`). If the user intends to use the p values obtained from the other statistical test, the `mode` parameter should be set to "user". In this scenario, before running the `FCh_plot()` function, the user should prepare a data.frame object named "user" containing two columns: the first column with gene names and the second column with p values (see example below).

The created plot is displayed on the graphic device. The returned object is a lists that contain two elements: an object with plot and a table with results.

Below, a variant with p values depending on the normality of the data is presented. Furthermore, genes with p < 0.05 and with at least 1.5-fold changed expression between groups are considered significant.

```
library(ggsignif)
#  Remember to use the first element of list object returned by `RQ_dCt()` or RQ_ddCt() function:
FCh.plot.pairwise <- FCh_plot(data = results.ddCt.pairwise[[1]],
                              use.p = TRUE,
                              mode = "depends.adj",
                              p.threshold = 0.05,
                              use.FCh = TRUE,
                              FCh.threshold = 1.5,
                              angle = 20)
```

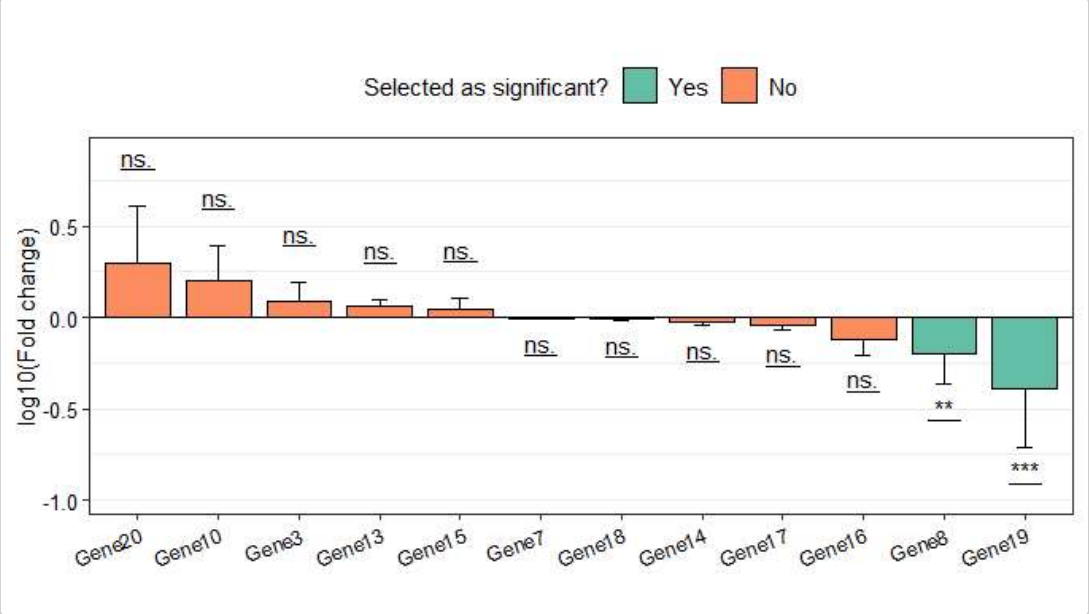



```
# Access the table with results:
head(as.data.frame(FCh.plot.pairwise[[2]]))
#>     Gene After_mean Before_mean  After_sd Before_sd After_norm_p Before_norm_p
#> 1 Gene10  3.4842353   3.2941765 0.9051358 1.3854897    0.1317249    0.79709435
#> 2 Gene13  6.6754706   6.5255882 0.8060583 0.5893336    0.7451515    0.41911404
#> 3 Gene14  6.0054118   5.6788824 0.4614858 0.6721121    0.5598084    0.01920284
#> 4 Gene15  7.2110588   6.5512941 0.9833445 0.9034758    0.1092715    0.05594945
#> 5 Gene16 -0.1601765  -0.8746471 0.8306935 0.5154298    0.2380653    0.28735006
#> 6 Gene17  4.0798824   3.7351176 0.7764462 0.6373618    0.7268583    0.90708293
#>         FCh    log10FCh    FCh_sd   t_test_p t_test_stat   MW_test_p
#> 1 1.5731857  0.19678000 1.5533045 0.66262724   0.4445108 0.554033858
#> 2 1.1437228  0.05832077 0.7683948 0.57420244   0.5736167 0.758312374
#> 3 0.9376122 -0.02797677 0.5691316 0.13105688   1.5915084 0.162571759
#> 4 1.1067408  0.04404593 1.4292426 0.09330078   1.7845808 0.092859531
#> 5 0.7577723 -0.12046129 0.5356882 0.01041055   2.9014075 0.005618046
#> 6 0.9001501 -0.04568507 0.4176108 0.11749771   1.6545444 0.123929225
#>   MW_test_stat t_test_p_adj MW_test_p_adj test.for.comparison     p.used
#> 1    0.5917263   0.66262724    0.66484063    t.student's.test 0.66262724
#> 2    0.3076977   0.66262724    0.82724986    t.student's.test 0.66262724
#> 3    1.3964742   0.22466894    0.27869444   Mann-Whitney.test 0.27869444
#> 4    1.6805028   0.22392187    0.24785845    t.student's.test 0.22392187
#> 5    2.7692792   0.04164219    0.02247218    t.student's.test 0.04164219
#> 6    1.5384885   0.22466894    0.24785845    t.student's.test 0.22466894
#>   Selected as significant?
#> 1                       No
#> 2                       No
#> 3                       No
#> 4                       No
#> 5                       No
#> 6                       No
```

Two genes (Gene8 and Gene19) met the significance criteria. Standard deviation bars and statistical significance annotations can be added to the plot:

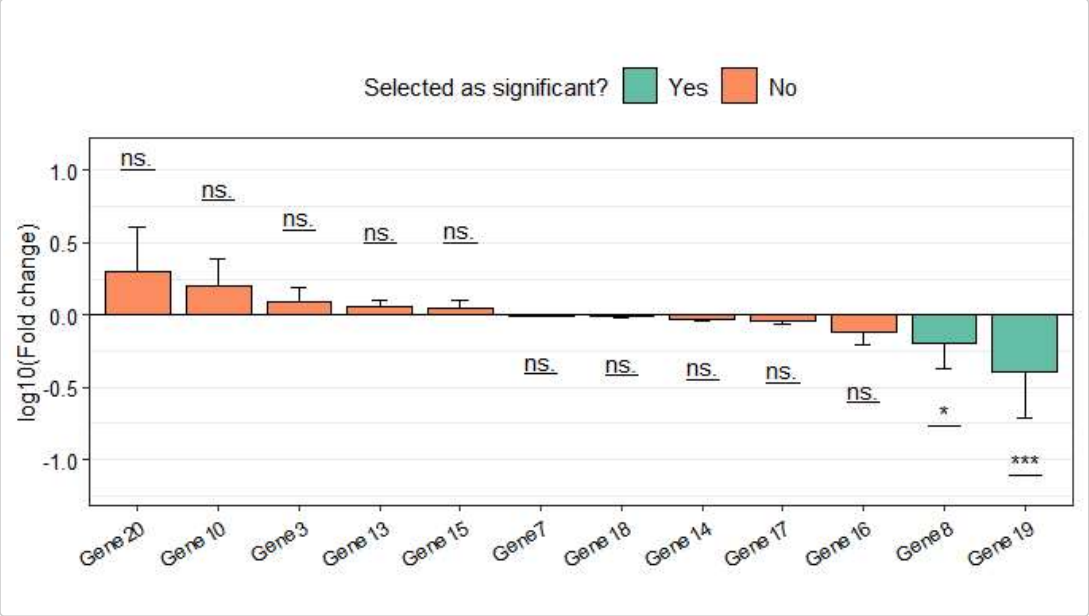
```
# Firstly prepare a vector with significance labels specified according to the user needings:
```

```
signif.labels <- c("ns.",
                   " ns. ",
                   "  ns.  ",
                   "   ns.   ",
                   "    ns.    ",
                   "     ns.     ",
                   "      ns.      ",
                   "       ns.       ",
                   "        ns.        ",
                   "         ns.         ",
                   "          ns.          ",
                   "**",
                   "***")
# Remember to set signif.show = TRUE:
FCh.plot.pairwise <- FCh_plot(data = results.ddCt.pairwise[[1]],
                              use.p = TRUE,
                              mode = "depends.adj",
                              p.threshold = 0.05,
                              use.FCh = TRUE,
                              FCh.threshold = 1.5,
                              use.sd = TRUE,
                              signif.show = TRUE,
                              signif.labels = signif.labels,
                              signif.dist = 0.2,
                              angle = 20)
```




```
# Variant with user p values - the used p values are calculated using the stats::wilcox.test()
        function:
user <- data.dCt.pairwise %>%
          pivot_longer(cols = -c(Group, Sample),
                       names_to = "Gene",
                       values_to = "dCt") %>%
          group_by(Gene) %>%
          summarise(MW_test_p = wilcox.test(dCt ~ Group)$p.value,
                    .groups = "keep")
# The stats::wilcox.test() functions is limited to cases without ties; therefore,
# a warning "cannot compute exact p-value with ties" will appear when ties occur.
signif.labels <- c("ns.",
                   " ns. ",
                   "  ns.  ",
                   "   ns.   ",
                   "    ns.    ",
                   "     ns.     ",
                   "      ns.      ",
                   "       ns.       ",
                   "        ns.        ",
                   "         ns.         ",
                   "          ns.          ",
                   " * ",
```

```
                    "***")
FCh.plot <- FCh_plot(data = results.ddCt.pairwise[[1]],
                     use.p = TRUE,
                     mode = "user",
                     p.threshold = 0.05,
                     use.FCh = TRUE,
                     FCh.threshold = 1.5,
                     use.sd = TRUE,
                     signif.show = TRUE,
                     signif.labels = signif.labels,
                     signif.dist = 0.4,
                     angle = 30)
```



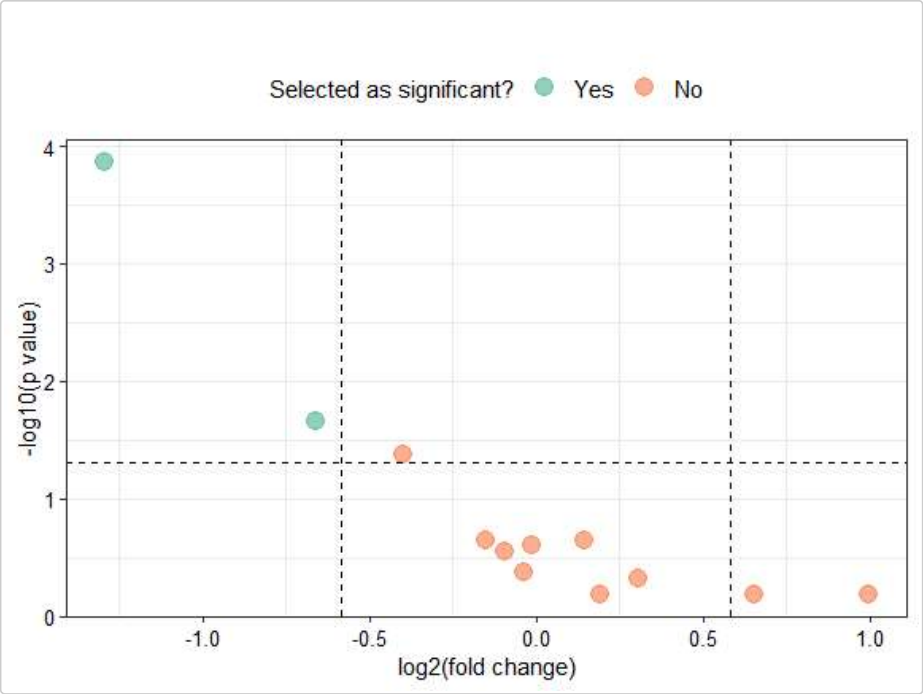
NOTE: If p values were not calculated due to the low number of samples, or they are not intended to be used to create a plot, the `use.p` parameter should be set to `FALSE`.

The `results_volcano()` function (a pairwise approach)

This function creates a volcano plot that illustrates the arrangement of genes based on fold change values and p values. Significant genes can be pointed out using specified p value and fold change thresholds, and highlighted on the plot by color and (optionally) isolated by thresholds lines. Similarly to `FCh_plot()` function, data returned from the `RQ_dCt()` and `RQ_ddCt()` functions can be directly applied (see The summary of standard workflow) and various sources of used p values are available (from the Student's t test, the Mann-Whitney U test, depended on the normality of data, or provided by the user).

The created plot is displayed on the graphic device. The returned object is a lists that contain two elements: an object with plot and a table with results.

```
# Genes with p < 0.05 and 2-fold changed expression between compared groups
# are considered significant. Remember to use the first element of list object
# returned by RQ_ddCt() function:
RQ.volcano.pairwise <- results_volcano(data = results.ddCt.pairwise[[1]],
                                       mode = "depends.adj",
                                       p.threshold = 0.05,
                                       FCh.threshold = 1.5)
```




```
# Access the table with results:
head(as.data.frame(RQ.volcano.pairwise[[2]]))
#>     Gene After_mean Before_mean  After_sd Before_sd After_norm_p Before_norm_p
#> 1 Gene10  3.4842353   3.2941765 0.9051358 1.3854897    0.1317249    0.79709435
#> 2 Gene13  6.6754706   6.5255882 0.8060583 0.5893336    0.7451515    0.41911404
#> 3 Gene14  6.0054118   5.6788824 0.4614858 0.6721121    0.5598084    0.01920284
#> 4 Gene15  7.2110588   6.5512941 0.9833445 0.9034758    0.1092715    0.05594945
#> 5 Gene16 -0.1601765  -0.8746471 0.8306935 0.5154298    0.2380653    0.28735006
#> 6 Gene17  4.0798824   3.7351176 0.7764462 0.6373618    0.7268583    0.90708293
#>         FCh    log10FCh    FCh_sd   t_test_p t_test_stat   MW_test_p
#> 1 1.5731857  0.19678000 1.5533045 0.66262724   0.4445108 0.554033858
#> 2 1.1437228  0.05832077 0.7683948 0.57420244   0.5736167 0.758312374
#> 3 0.9376122 -0.02797677 0.5691316 0.13105688   1.5915084 0.162571759
#> 4 1.1067408  0.04404593 1.4292426 0.09330078   1.7845808 0.092859531
#> 5 0.7577723 -0.12046129 0.5356882 0.01041055   2.9014075 0.005618046
#> 6 0.9001501 -0.04568507 0.4176108 0.11749771   1.6545444 0.123929225
#>   MW_test_stat t_test_p_adj MW_test_p_adj test.for.comparison     p.used
#> 1    0.5917263   0.66262724    0.66484063    t.student's.test 0.66262724
#> 2    0.3076977   0.66262724    0.82724986    t.student's.test 0.66262724
#> 3    1.3964742   0.22466894    0.27869444   Mann-Whitney.test 0.27869444
#> 4    1.6805028   0.22392187    0.24785845    t.student's.test 0.22392187
#> 5    2.7692792   0.04164219    0.02247218    t.student's.test 0.04164219
#> 6    1.5384885   0.22466894    0.24785845    t.student's.test 0.22466894
#>   Selected as significant?
#> 1                       No
#> 2                       No
#> 3                       No
#> 4                       No
#> 5                       No
#> 6                       No
```

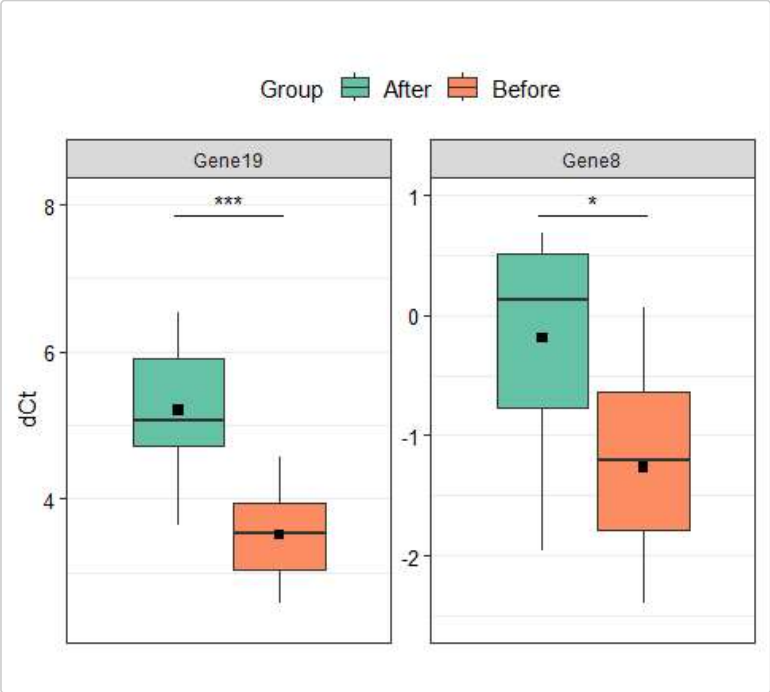
The `results_boxplot()` function (a pairwise approach)

This function creates a boxplot that illustrates the distribution of the data for the genes. It is similar to `control_boxplot_gene()` function; however, some new options are added, including gene selection, faceting, addition of mean points to boxes, and statistical significance labels.

Data objects returned from the `make_Ct_ready()` and `delta_Ct() functions` can be directly applied to this function (see The summary of standard workflow).

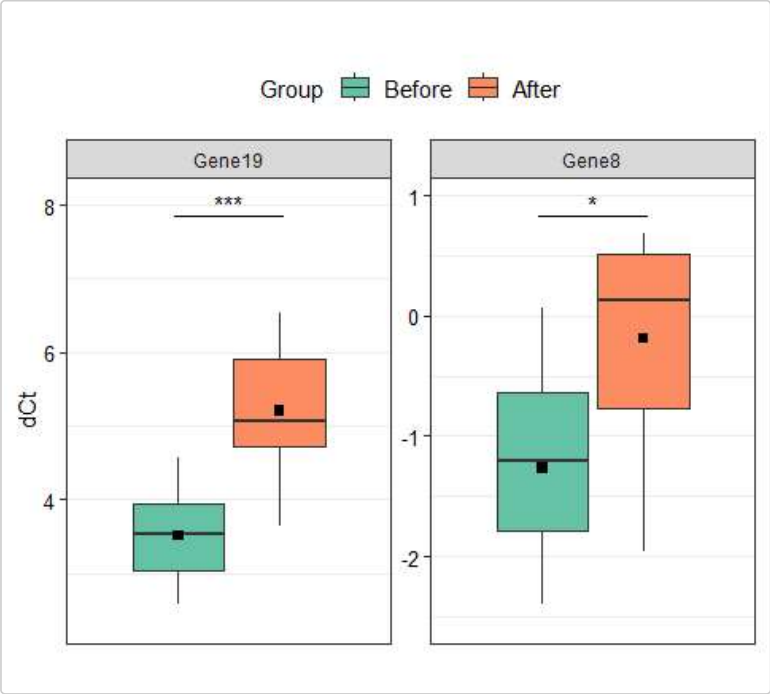
```
final.boxplot.pairwise <- results_boxplot(data = data.dCt.pairwise.F,
                                          sel.Gene = c("Gene8","Gene19"),
                                          by.group = TRUE,
                                          signif.show = TRUE,
                                          signif.labels = c("***","*"),
```

```
                                        signif.dist = 1.2,
                                        faceting = TRUE,
                                        facet.row = 1,
                                        facet.col = 2,
                                        y.exp.up = 0.1,
                                        y.axis.title = "dCt")
```



If the order of groups in the legend should be changed (to put Before group as first), the levels of the variable with group names should be reordered:

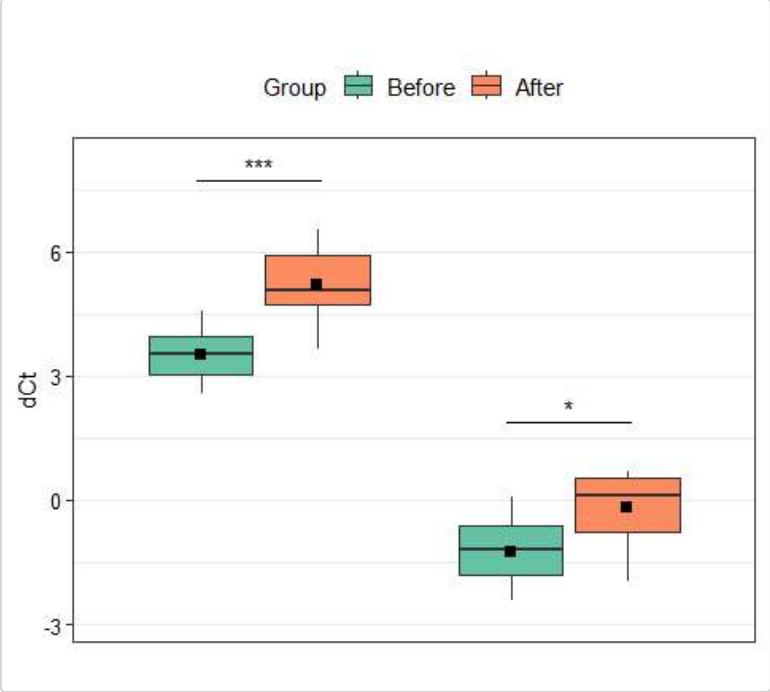
```
data.dCt.pairwise.F$Group <- factor(data.dCt.pairwise.F$Group, levels = c("Before", "After"))

final.boxplot.pairwise <- results_boxplot(data = data.dCt.pairwise.F,
                                          sel.Gene = c("Gene8","Gene19"),
                                          by.group = TRUE,
                                          signif.show = TRUE,
                                          signif.labels = c("***","*"),
                                          signif.dist = 1.2,
                                          faceting = TRUE,
                                          facet.row = 1,
                                          facet.col = 2,
                                          y.exp.up = 0.1,
                                          y.axis.title = "dCt")
```



NOTE: If missing values are present in the data, they will be automatically removed, and a warning will appear.

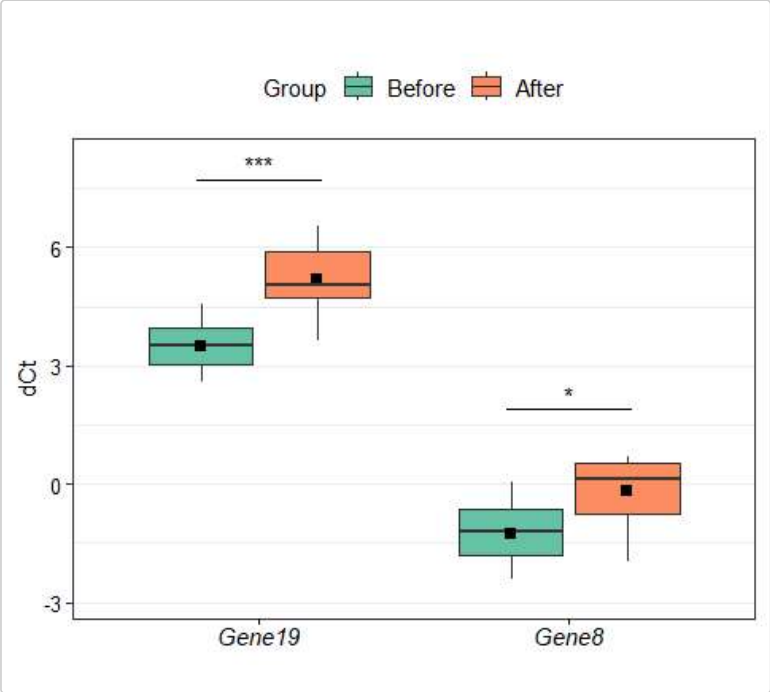
For plots without faceting, x axis annotations should be added:

```
final.boxplot.pairwise <- results_boxplot(data = data.dCt.pairwise.F,
                                          sel.Gene = c("Gene8","Gene19"),
                                          by.group = TRUE,
                                          signif.show = TRUE,
                                          signif.labels = c("***","*"),
                                          signif.dist = 1.2,
                                          faceting = FALSE,
                                          y.exp.up = 0.1,
                                          y.axis.title = "dCt")
```



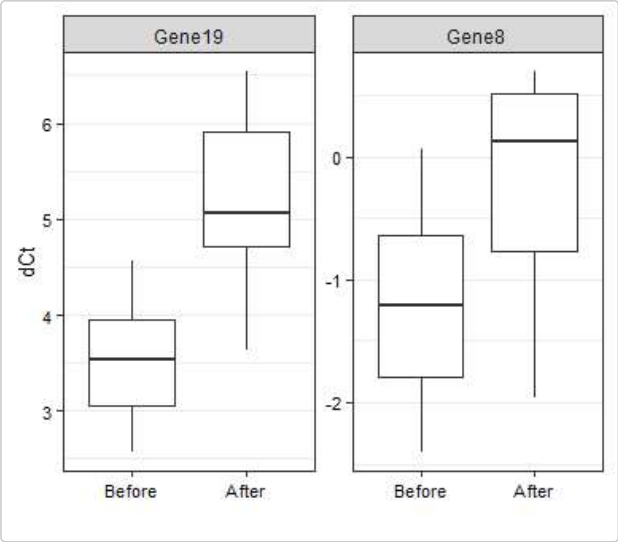

```
# Add x axis annotations and ticks:
final.boxplot.pairwise <- final.boxplot.pairwise +
                          theme(axis.text.x = element_text(size = 11, colour = "black",
        face="italic"),
                                axis.ticks.x = element_line(colour = "black"))
```

```
final.boxplot.pairwise
```



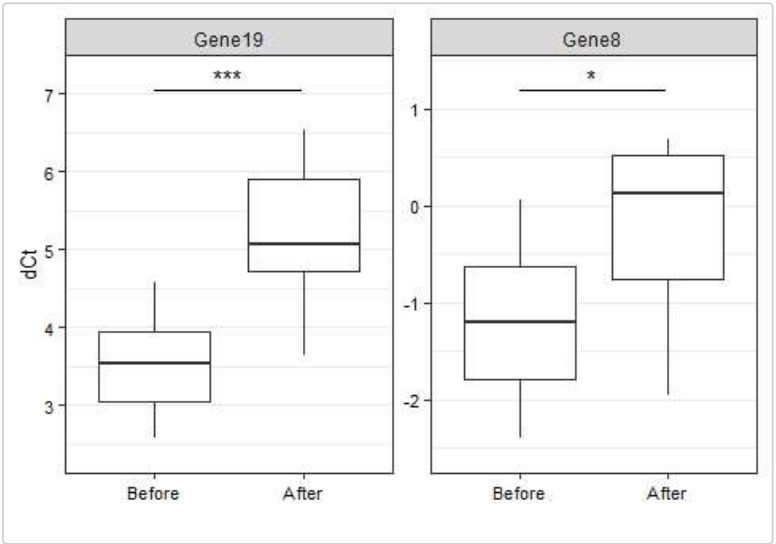
A situation may arise in which the user prefers to generate a faceted plot without a color legend, displaying group names as annotations on the x-axis. This can be the solution for this situation:

```
library(tidyverse)
colnames(data.dCt.pairwise.F)
#>  [1] "Group"  "Sample" "Gene10" "Gene13" "Gene14" "Gene15" "Gene16" "Gene17"
#>  [9] "Gene18" "Gene19" "Gene20" "Gene3"  "Gene7"  "Gene8"
data.dCt.pairwise.F.slim <- pivot_longer(data.dCt.pairwise.F,
                                         cols = Gene10:Gene8,
                                         names_to = "gene",
                                         values_to = "exp")

# Select genes
data.dCt.pairwise.F.slim.sel <- data.dCt.pairwise.F.slim[data.dCt.pairwise.F.slim$gene %in%
         c("Gene19","Gene8"), ]

# Change order of groups if needed
data.dCt.pairwise.F.slim.sel$Group <- factor(data.dCt.pairwise.F.slim.sel$Group,
                                             levels = c("Before","After"))

# Create plot
final_boxplot_no_colors <- ggplot(data.dCt.pairwise.F.slim.sel, aes(x = Group, y = exp)) +
                           geom_boxplot(outlier.shape = NA, coef = 2) +
                           theme_bw() +
                           ylab("dCt") +
                           xlab("") +
                           theme(axis.text = element_text(size = 8, color = "black")) +
                           theme(axis.title = element_text(size = 10, color = "black")) +
                           theme(panel.grid.major.x = element_blank()) +
                           facet_wrap(vars(gene), nrow = 1, dir = "h", scales = "free")

final_boxplot_no_colors
```



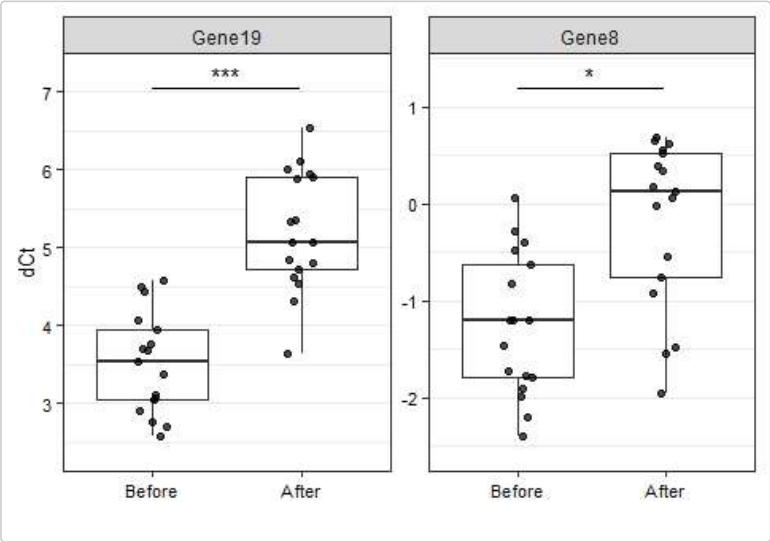
To add significance labels, the following code can be used:

```
data.label <- data.frame(matrix(nrow = 2, ncol = 4)) # Number of rows is equal to number of genes
rownames(data.label) <- c("Gene19","Gene8")
colnames(data.label) <- c("x", "xend", "y", "annotation")
data.label$gene <- rownames(data.label) # Name of column with gene symbols in this table
# must be the same as name of the column with gene symbols in data used for create the plot.

data.label$y <- 0.5 + c(max(data.dCt.pairwise.F$Gene19), max(data.dCt.pairwise.F$Gene8))
data.label$x <- c(1,1)
data.label$xend <- c(1.98,1.98)
data.label$annotation <- c("***","*")

final_boxplot_no_colors_labels <- final_boxplot_no_colors +
 geom_signif(
    stat = "identity",
    data = data.label,
    aes(x = x,
        xend = xend,
        y = y,
        yend = y,
        annotation = annotation),
    color = "black",
    manual = TRUE) +
   scale_y_continuous(expand = expansion(mult = c(0.1, 0.1)))

final_boxplot_no_colors_labels
```



If points of individual samples need to be added, it can be simply run:

```
final_boxplot_no_colors_labels_points <- final_boxplot_no_colors_labels +
                                         geom_point(position=position_jitter(w=0.1,h=0), alpha = 0.7,
          size = 1.5)

final_boxplot_no_colors_labels_points
```

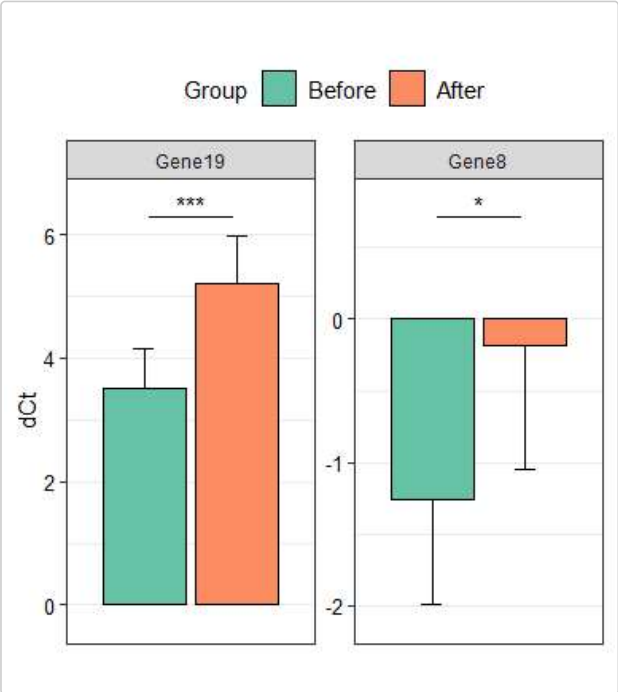


The `results_barplot()` function (a pairwise approach)

This function creates a barplot that illustrates mean and standard deviation values of the data for all or selected genes.

Data objects returned from the `make_Ct_ready()` and `delta_Ct() functions` can be directly applied to this function (see The summary of standard workflow).

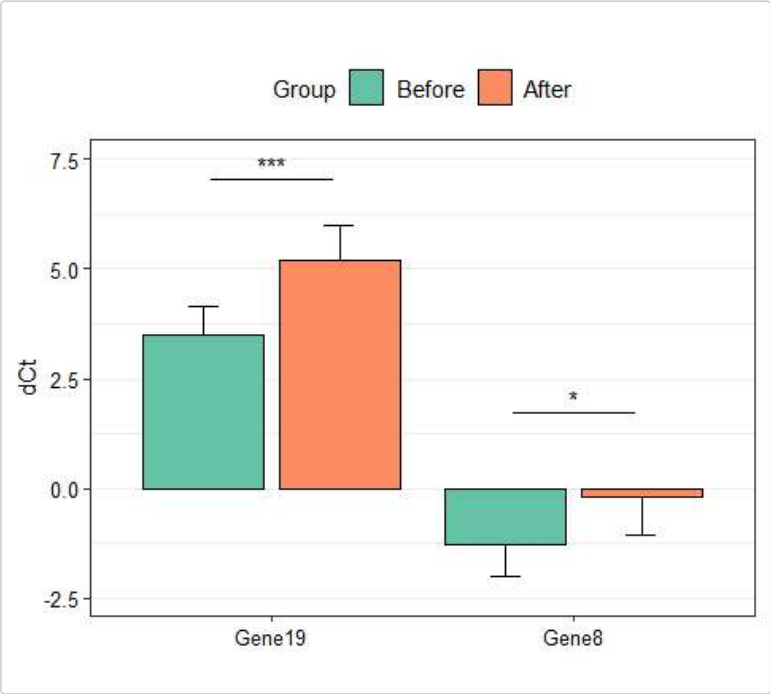
```
final.barplot.pairwise <- results_barplot(data = data.dCt.pairwise.F,
                                          sel.Gene = c("Gene8","Gene19"),
                                          signif.show = TRUE,
                                          signif.labels = c("***","*"),
                                          angle = 30,
                                          signif.dist = 1.05,
                                          faceting = TRUE,
                                          facet.row = 1,
                                          facet.col = 4,
                                          y.exp.up = 0.1,
                                          y.axis.title = "dCt")
```



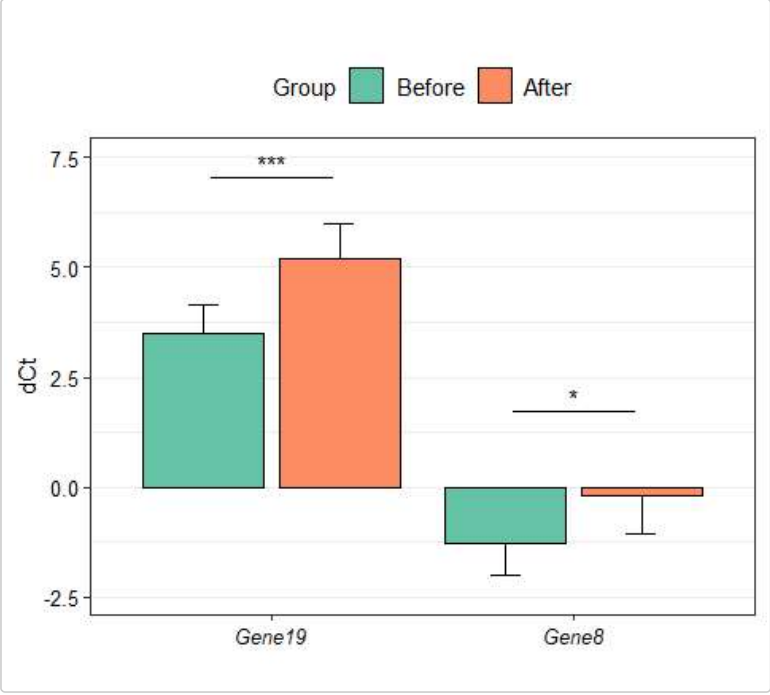
NOTE: At least two samples in each group are required to calculate the standard deviation and properly generate the plot.

If faceting is not needed, simply run this function with `faceting = FALSE`:

```
final.barplot.pairwise <- results_barplot(data = data.dCt.pairwise.F,
                                          sel.Gene = c("Gene8","Gene19"),
                                          signif.show = TRUE,
                                          signif.labels = c("***","*"),
                                          angle = 0,
                                          signif.dist = 1.05,
                                          faceting = FALSE,
                                          y.exp.up = 0.1,
                                          y.axis.title = "dCt")
```

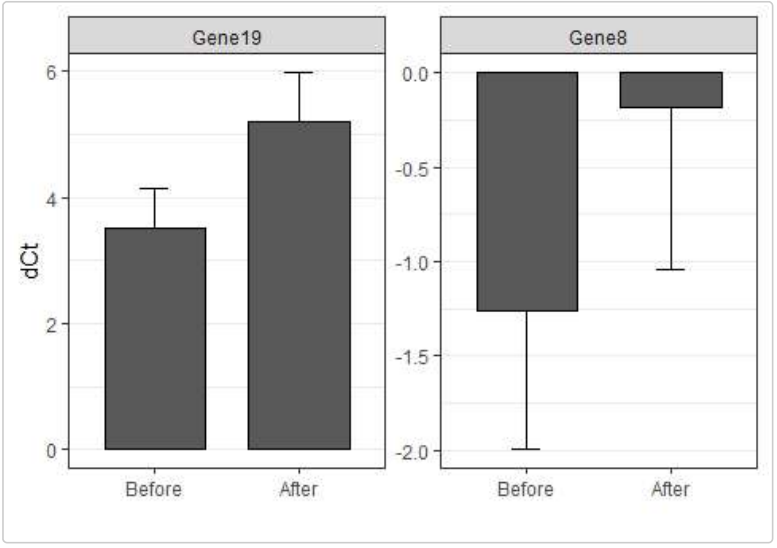



```
# Add italic font to the x axis:
final.barplot.pairwise <- final.barplot.pairwise +
                          theme(axis.text.x = element_text(face="italic"))

final.barplot.pairwise
```



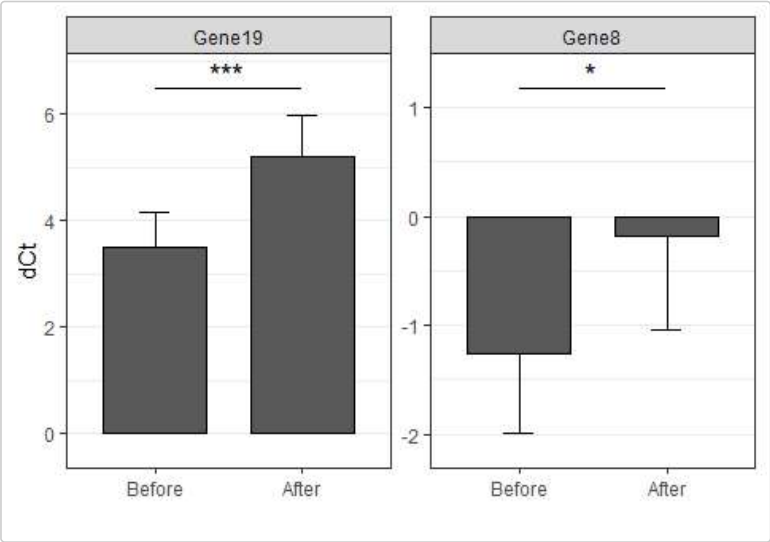
A situation may arise in which the user prefers to generate a faceted plot without a color legend, displaying group names as annotations on the x axis. This can be the solution for this situation:

```
library(tidyverse)
colnames(data.dCt.pairwise.F)
#>  [1] "Group"  "Sample" "Gene10" "Gene13" "Gene14" "Gene15" "Gene16" "Gene17"
#>  [9] "Gene18" "Gene19" "Gene20" "Gene3"  "Gene7"  "Gene8"
data.dCt.pairwise.F.slim <- pivot_longer(data.dCt.pairwise.F,
                                         cols = Gene10:Gene8,
                                         names_to = "gene",
                                         values_to = "exp")

# Select genes
data.dCt.pairwise.F.slim.sel <- data.dCt.pairwise.F.slim[data.dCt.pairwise.F.slim$gene %in%
         c("Gene19","Gene8"), ]

# Change order of groups if needed
data.dCt.pairwise.F.slim.sel$Group <- factor(data.dCt.pairwise.F.slim.sel$Group,
                                             levels = c("Before","After"))

data.mean <- data.dCt.pairwise.F.slim.sel %>%
              group_by(Group, gene) %>%
              summarise(mean = mean(exp, na.rm = TRUE), .groups = "keep")

data.sd <- data.dCt.pairwise.F.slim.sel %>%
            group_by(Group, gene) %>%
            summarise(sd = sd(exp, na.rm = TRUE), .groups = "keep")

data.mean$sd <- data.sd$sd

final_barplot_no_colors <- ggplot(data.mean, aes(x = Group, y = mean)) +
                                geom_errorbar(aes(group = Group,
                                                  y = mean,
                                                  ymin = ifelse(mean < 0, mean - abs(sd), mean),
                                                  ymax = ifelse(mean > 0, mean + abs(sd), mean)),
                                              width = .2,
                                              position = position_dodge(0.9)) +

                                geom_col(aes(group = Group),
                                         position = position_dodge(0.9),
                                         width = 0.7,
                                         color = "black") +
                                xlab("") +
                                ylab("dCt") +
                                theme_bw() +
                                theme(panel.grid.major.x = element_blank()) +
                                facet_wrap(vars(gene), scales = "free", nrow = 1)

final_barplot_no_colors
```



Significance labels can be subsequently added:

```
data.label <- data.frame(matrix(nrow = 2, ncol = 4)) # Number of rows is equal to number of genes
rownames(data.label) <- c("Gene19","Gene8")
colnames(data.label) <- c("x", "xend", "y", "annotation")
data.label$gene <- rownames(data.label) # Name of column with gene symbols
# in this table must be the same as name of the column with gene symbols
# in data used for create the plot.

data.mean <- data.mean %>%
               mutate(max = mean + sd) %>%
               group_by(gene) %>%
               summarise(height = max(max, na.rm = TRUE), .groups = "keep")
data.label$y <- 0.5 + data.mean$height
data.label$x <- c(1,1)
data.label$xend <- c(1.98,1.98)
data.label$annotation <- c("***","*")

final_barplot_no_colors_labels <- final_barplot_no_colors +
                                  geom_signif(
                                  stat = "identity",
                                  data = data.label,
                                  aes(x = x,
                                     xend = xend,
                                     y = y,
                                     yend = y,
                                     annotation = annotation,
                                     textsize = 5),
                                  color = "black",
                                  manual = TRUE) +
                                  scale_y_continuous(expand = expansion(mult = c(0.1, 0.1)))

final_barplot_no_colors_labels
```



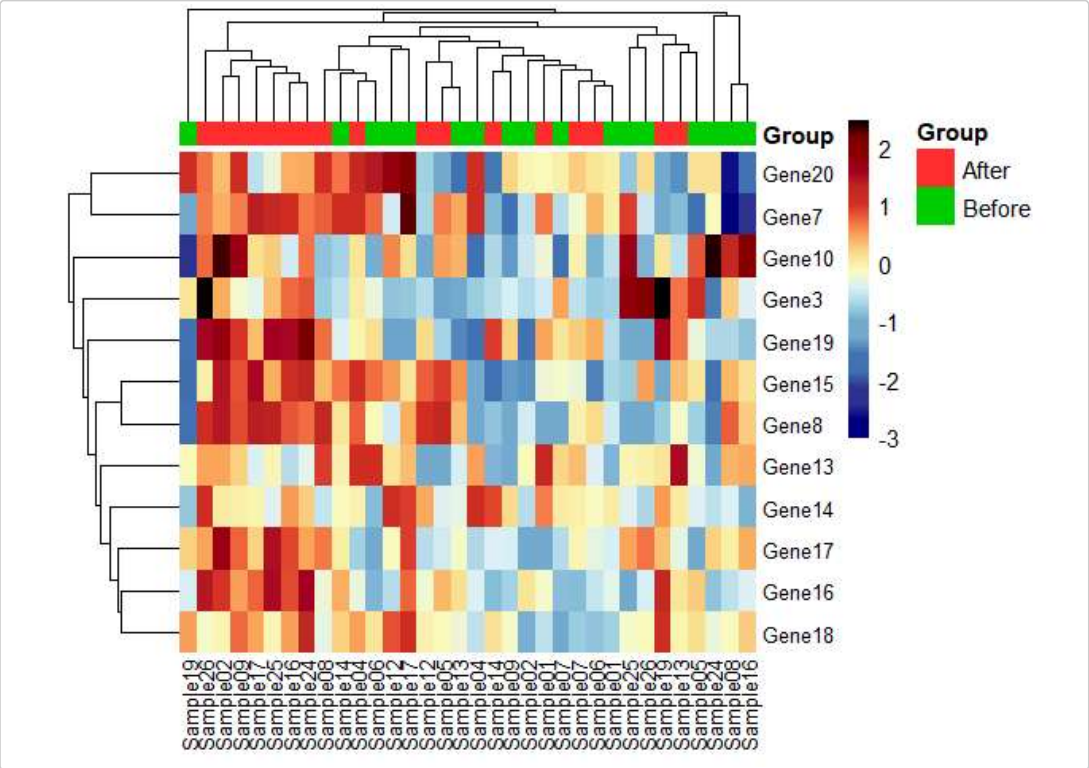
The `results_heatmap()` function (a pairwise approach)

This function allows to draw heatmap with hierarchical clustering. Various methods of distance calculation (e.g. euclidean, canberra) and agglomeration (e.g. complete, average, single) can be used.

NOTE: Remember to create named list with colors for groups annotation and pass it in `col.groups` parameter.

```
# Create named list with colors for groups annotation:
colors.for.groups = list("Group" = c("After"="firebrick1", "Before"="green3"))
# Vector of colors for heatmap can be also specified to fit the user needings:
colors <- c("navy","navy","#313695","#313695","#4575B4","#4575B4","#74ADD1","#74ADD1",
            "#ABD9E9","#E0F3F8","#FFFFBF","#FEE090","#FDAE61","#F46D43",
            "#D73027","#C32B23","#A50026","#8B0000",
            "#7E0202","#000000")
library(pheatmap)
results_heatmap(data.dCt.pairwise.F,
```

```
                sel.Gene = "all",
                col.groups = colors.for.groups,
                colors = colors,
                show.colnames = TRUE,
                show.rownames = TRUE,
                fontsize = 11,
                fontsize.row = 10,
                cellwidth = 8,
                angle.col = 90)
```



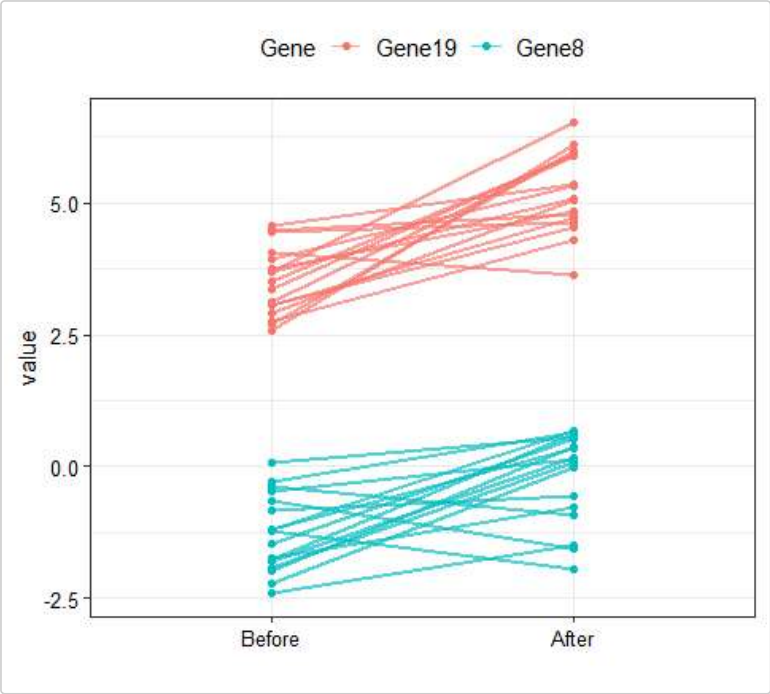

```
# Cellwidth parameter was set to 8 to avoid cropping the image on the right side.
```

The created plot is displayed on the graphic device (if `save.to.tiff = FALSE`) or saved to .tiff file (if `save.to.tiff = TRUE`).

The `parallel_plot()` function (a pairwise approach)

This function can be used to illustrate a pairwise changes in genes expression.

```
parallel.plot <- parallel_plot(data = data.dCt.pairwise.F,
                               sel.Gene = c("Gene19","Gene8"),
                               order = c(4,3))
```



Further analyses (a pairwise approach)

Gene expression levels and differences between groups can be further analysed using the following methods and corresponding functions of the `RQdeltaCT` package:

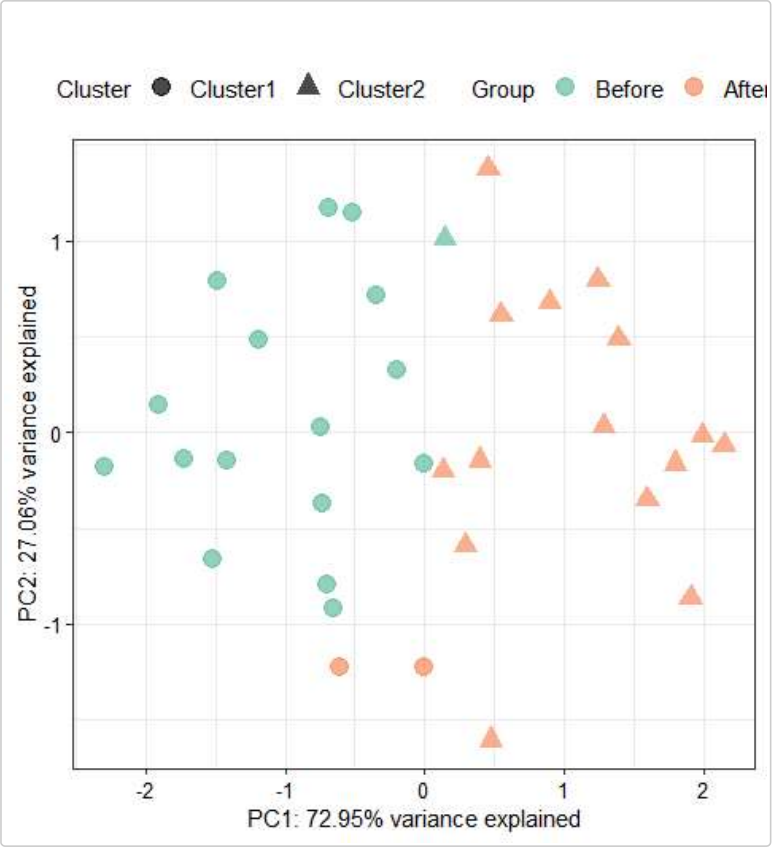
- principal component analysis (PCA) and k means clustering - used to assess samples clustering based on the gene expression data (`pca_kmeans()` function)
- correlation analysis - used to generate and visualise the correlation matrix of samples (`corr_sample()` function) and genes (`corr_gene()` function).
- simple linear regression - used to analysis and visualisation of relationships between pair of samples (`single_pair_sample()` function) and genes (`single_pair_gene()` function).
- Receiver Operating Characteristic (ROC) analysis - used to evaluate performance of sample classification by gene expression data (`ROCh()` function).
- simple logistic regression analysis - used to calculate odds ratio values for genes (`log_reg()` function).

Data objects returned from the `make_Ct_ready()` and `delta_Ct() functions` can be directly applied to all of these functions (see The summary of standard workflow). Moreover, all functions can be run on the entire data or only on selected samples/genes.

PCA and k means clustering (a pairwise approach)

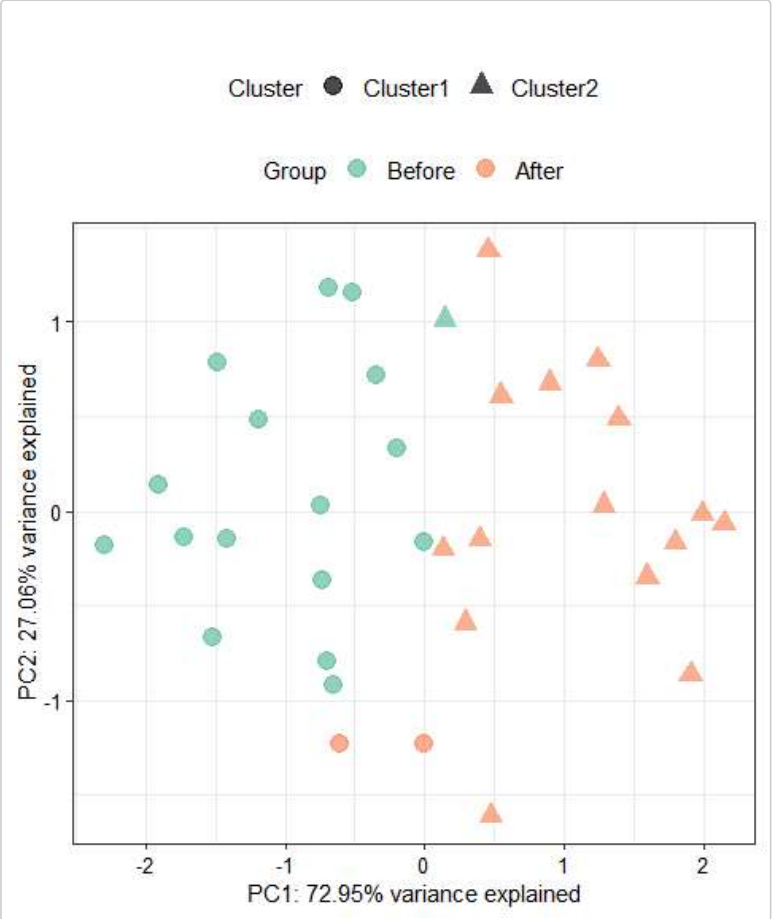
This function allows to simultaneously perform principal component analysis (PCA) and (if `do.k.means = TRUE`) samples classification using k means method. Number of clusters can be set using `k.clust` parameter. Results obtained from both methods can be presented on one plot. Obtained clusters are distinguishable by point shapes. Confusion matrix of sample classification is returned as the second element of the returned object and can be used for calculation further parameters of classification performance like precision, accuracy, recall, and others.

```
pca.kmeans <- pca_kmeans(data.dCt.pairwise.F,
                         sel.Gene = c("Gene8","Gene19"),
                         legend.position = "top")
```



If the legend is cropped, using a vertical position of legend should solve this problem:

```
pca.kmeans[[1]] + theme(legend.box = "vertical")
```



Access to the confusion matrix:

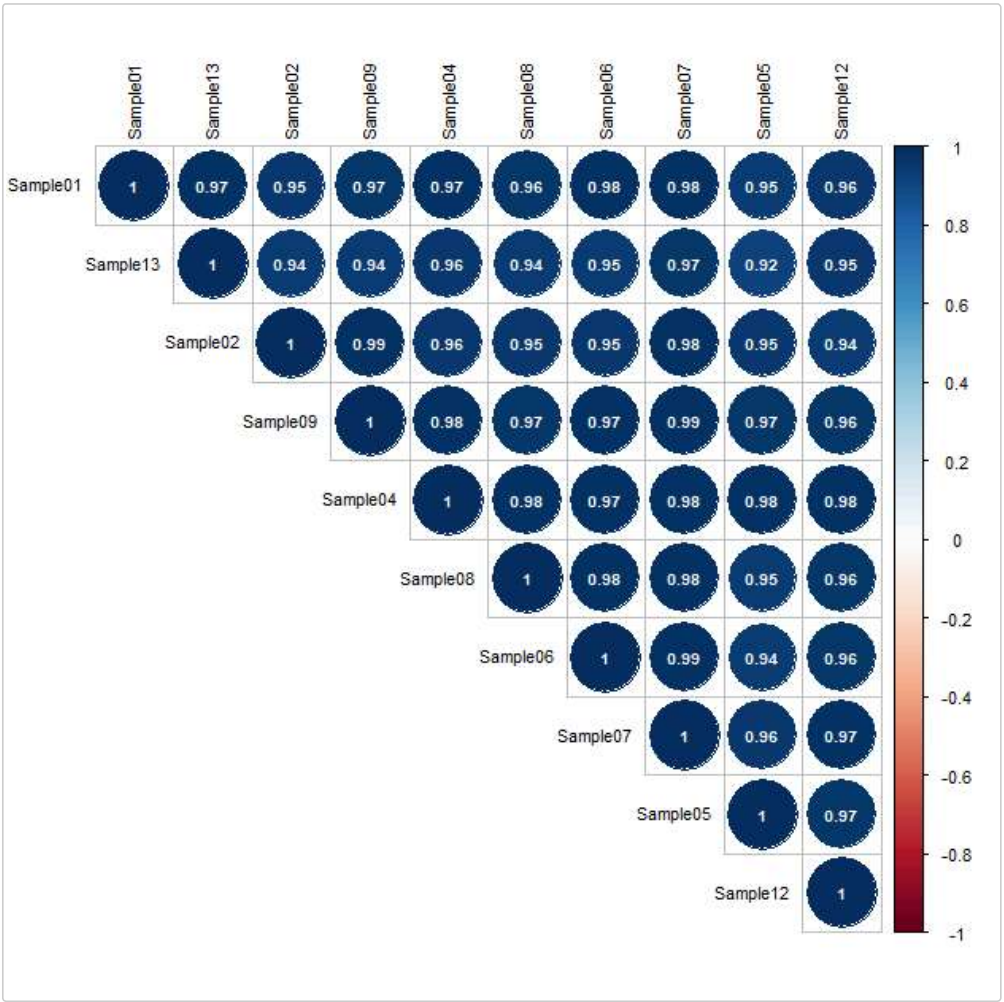
```
pca.kmeans[[2]]
#>
#>          Cluster1 Cluster2
```

```
#>   Before       16        1
#>   After         2       15
```

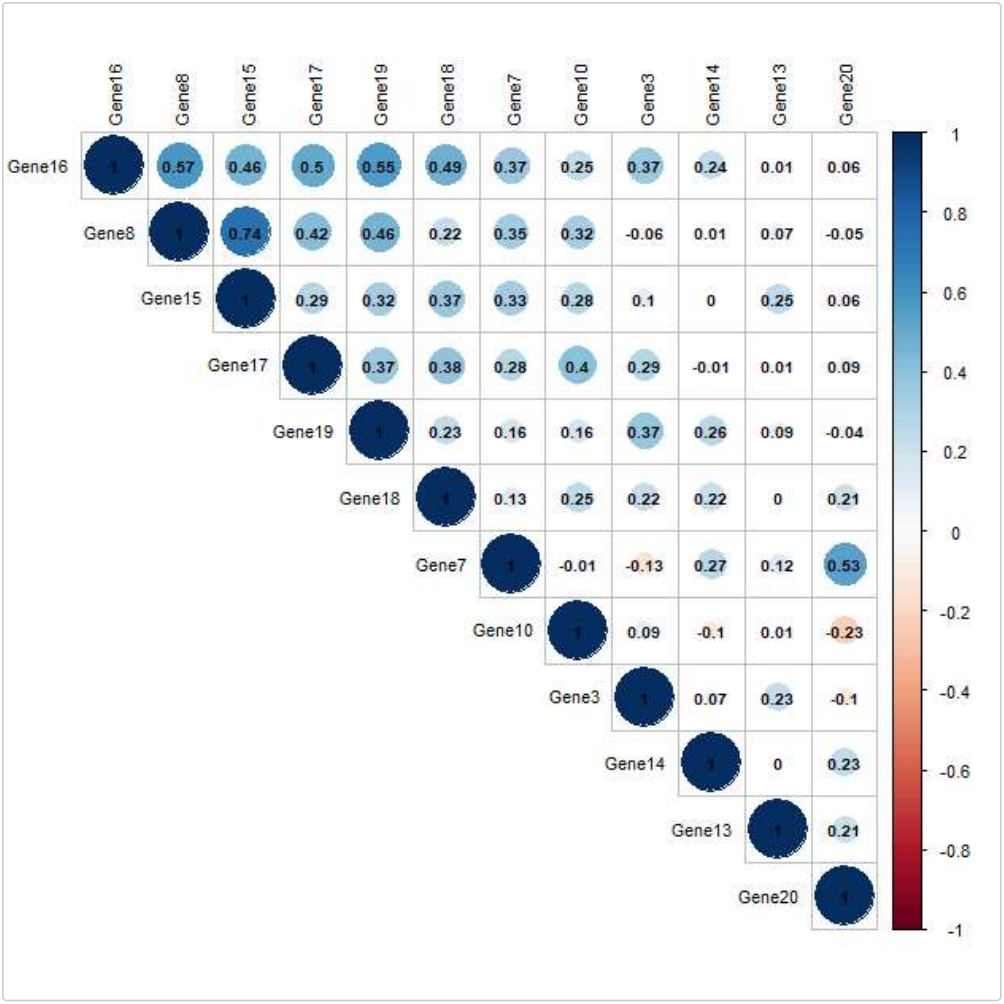
Correlation analysis (a pairwise approach)

The `RQdeltaCT` package offers `corr_sample()` and `corr_gene()` functions to generate and plot correlation matrices of samples and genes, respectively. The correlation coefficients can be calculated using either the Pearson or Spearman algorithm. To facilitate plot interpretation, these functions also have possibilities to order samples or genes according to several methods, e.g. hierarchical clustering or PCA first component (see `order` parameter).

```
library(Hmisc)
library(corrplot)
# To make the plot more readable, only part of the data was used:
corr.samples <- corr_sample(data = data.dCt.pairwise.F[1:10, ],
                            method = "pearson",
                            order = "hclust",
                            size = 0.7,
                            p.adjust.method = "BH",
                            add.coef = "white")
```




```
library(Hmisc)
library(corrplot)
corr.genes <- corr_gene(data = data.dCt.pairwise.F,
                        method = "pearson",
                        order = "FPC",
                        size = 0.7,
                        p.adjust.method = "BH")
```



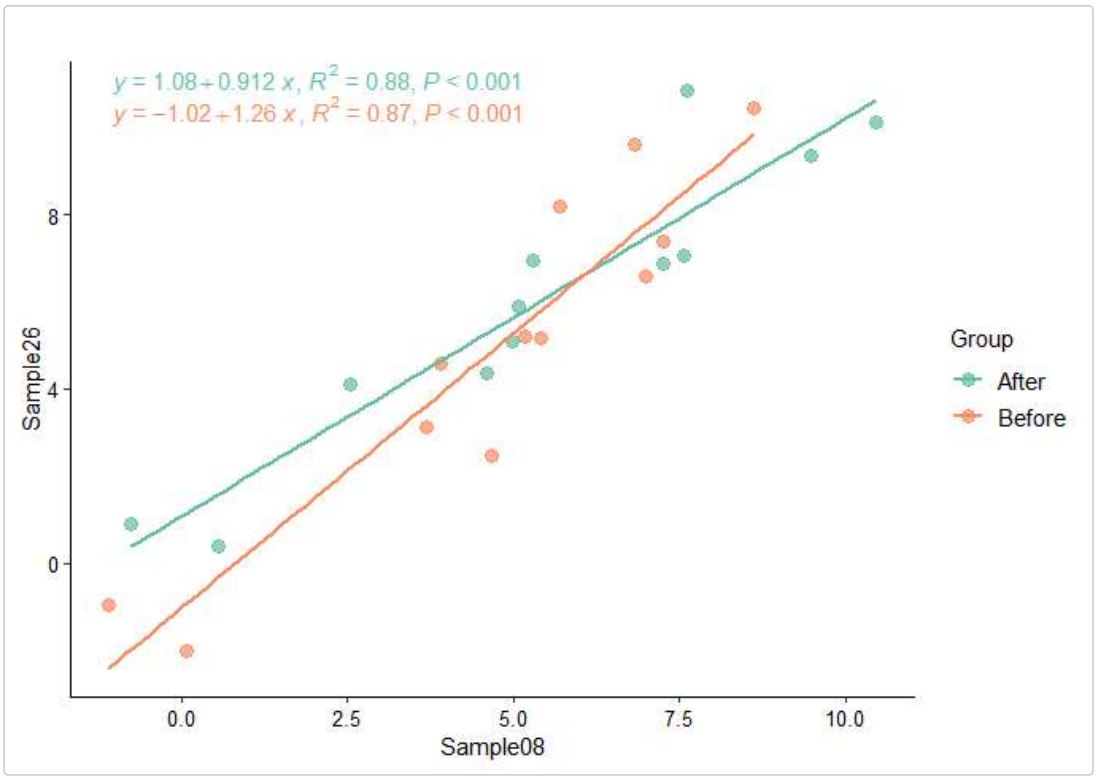
The created plots are displayed on the graphic device. The returned objects are tables with computed correlation coefficients, p values, and p values adjusted by Benjamini-Hochberg correction (by default). Tables are sorted by the absolute values of correlation coefficients in descending order.

NOTE: A minimum of 5 samples/target are required for correlation analysis.

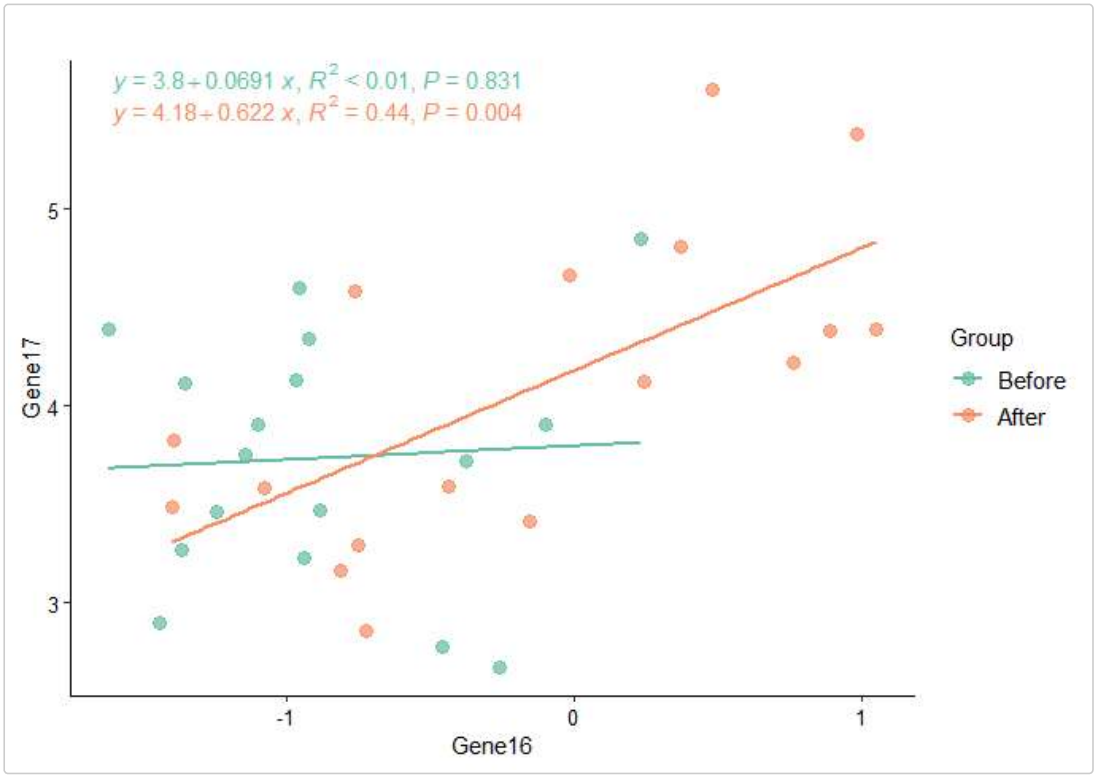
Simple linear regression analysis (a pairwise approach)

Linear relationships between pairs of samples or genes can be further analysed using simple linear regression models using the `single_pair_sample()` function (for analysis of samples) and the `single_pair_gene()` function for analysis of genes. These functions draw a scatter plot with a simple linear regression line. Regression results such as regression equation, coefficient of determination, F value, or p value can be optionally added to the plot.

```
library(ggpmisc)
Sample08_Sample26 <- single_pair_sample(data = data.dCt.pairwise,
                                        pairwise.data = TRUE,
                                        by.group = TRUE,
                                        x = "Sample08",
                                        y = "Sample26",
                                        point.size = 3,
                                        labels = TRUE,
                                        label = c("eq", "R2", "p"),
                                        label.position.x = 0.05,
                                        label.position.y = c(1, 0.95))
```




```
library(ggpmisc)
Gene16_Gene17 <- single_pair_gene(data.dCt.pairwise.F,
                                  x = "Gene16",
                                  y = "Gene17",
                                  by.group = TRUE,
                                  point.size = 3,
                                  labels = TRUE,
                                  label = c("eq", "R2", "p"),
                                  label.position.x = c(0.05),
                                  label.position.y = c(1,0.95))
```



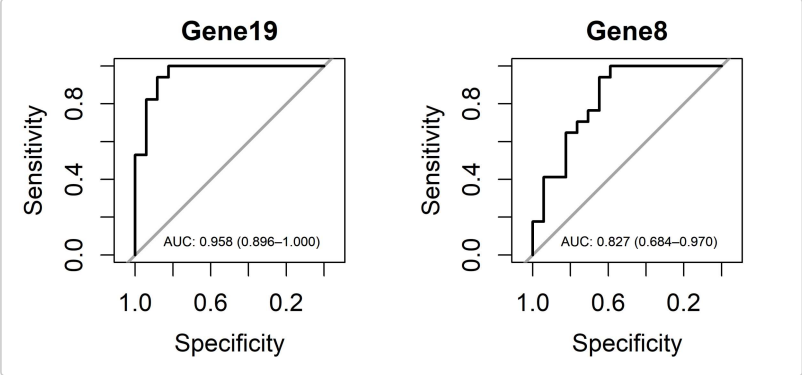
Receiver Operating Characteristic (ROC) analysis (a pairwise approach)

The Receiver Operating Characteristic (ROC) analysis is useful for assessing the performance of sample classification to the particular group, based on gene expression data. In this analysis, ROC curves together with

parameters such as the area under curve (AUC), specificity, sensitivity, accuracy, positive and negative predictive value are received. The `ROCh()` function was designed to perform all of these tasks. This function returns a table with calculated parameters and a plot with multiple panels, each with ROC curve for one gene.

NOTE: The created plot is not displayed on the graphic device, but should be saved as .tiff image (`save.to.txt = TRUE`) and can be opened directly from that file in the working directory.

```
library(pROC)
# Remember to specify the numbers of rows (panels.row parameter) and columns (panels.col parameter)
# to provide sufficient place to arrange panels:
roc_parameters <- ROCh(data = data.dCt.pairwise.F,
                       sel.Gene = c("Gene8","Gene19"),
                       groups = c("After","Before"),
                       panels.row = 1,
                       panels.col = 2)
# Access to calculated parameters:
roc_parameters
#>     Gene Threshold Specificity Sensitivity  Accuracy       ppv npv   youden
#> 1 Gene19    4.5975   0.8235294           1 0.9117647 0.8500000   1 1.823529
#> 2  Gene8    0.0615   0.5882353           1 0.7941176 0.7083333   1 1.588235
#>         AUC
#> 1 0.9584775
#> 2 0.8269896
```



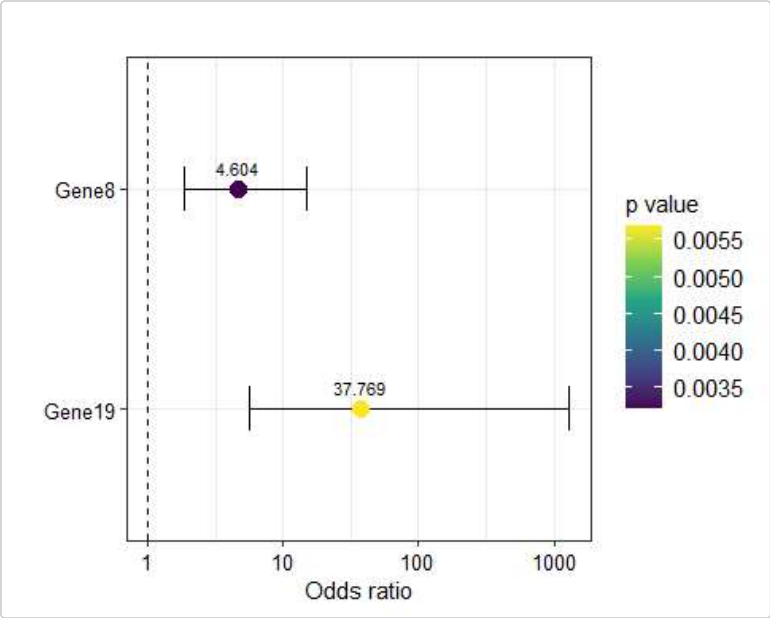
Obtained warning informed us that analyzed genes have more than 1 threshold value for calculated Youden J statistic. To get all thresholds, ROC analysis should be performed separately for each gene:

```
# Filter data:
data <- data.dCt.pairwise.F[, colnames(data.dCt.pairwise.F) %in% c("Group", "Sample", "Gene19")]
# Perform analysis:
data_roc <- roc(response = data$Group,
                predictor = as.data.frame(data)$Gene19,
                levels = c("Before","After"),
                smooth = FALSE,
                auc = TRUE,
                plot = FALSE,
                ci = TRUE,
                of = "auc",
                quiet = TRUE)
# Gain parameters:
parameters <- coords(data_roc,
                     "best",
                     ret = c("threshold",
                             "specificity",
                             "sensitivity",
                             "accuracy",
                             "ppv",
                             "npv",
                             "youden"))
parameters
#>   threshold specificity sensitivity  accuracy    ppv       npv   youden
#> 1    4.5130   0.9411765   0.8823529 0.9117647 0.9375 0.8888889 1.823529
#> 2    4.5975   1.0000000   0.8235294 0.9117647 1.0000 0.8500000 1.823529
# Gain AUC
data_roc$auc
#> Area under the curve: 0.9585
```

Simple logistic regression (a pairwise approach)

Logistic regression is a useful method to investigate the impact of the analysed variable on the odds of the occurrence of the studied experimental condition. In the `RQdeltaCT` package, `log_reg()` function allows to calculate for each gene a chances (odds ratio, OR) of being included in the study group when gene expression level increases by one unit (suitable for non-transformed data) or by mean of expression levels (more suitable for transformed data). This function returns a plot and table with the calculated parameters (OR, confidence interval, intercept, coefficient, and p values).

```
library(oddsratio)

# Remember to set the increment parameter.
log.reg.results <- log_reg(data = data.dCt.pairwise.F,
                           increment = 1,
                           sel.Gene = c("Gene8","Gene19"),
                           group.study = "After",
                           group.ref = "Before",
                           log.axis = TRUE,
                           p.adjust = FALSE)
```

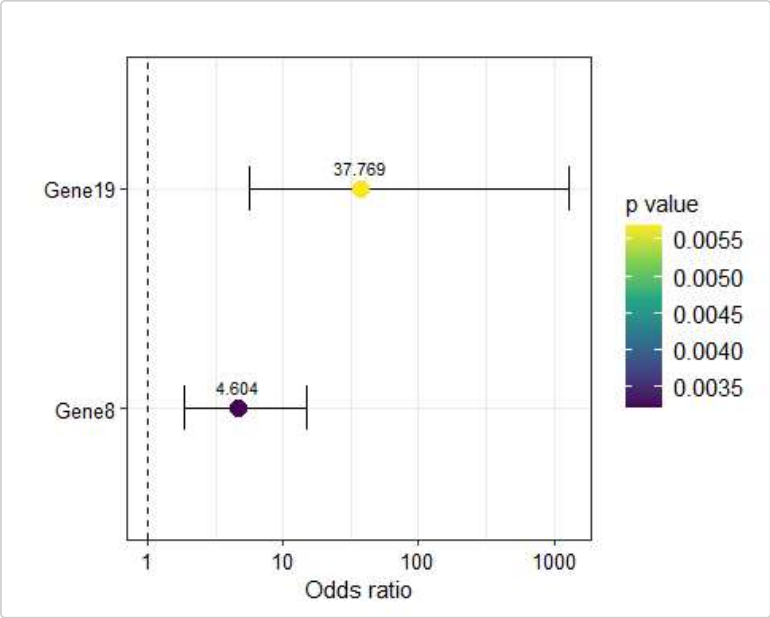



```
log.reg.results[[2]]
#>     Gene oddsratio CI_low  CI_high Increment  Intercept coeficient p_intercept
#> 1 Gene19    37.769  5.552 1304.777         1 -15.769827   3.631485 0.006397974
#> 2  Gene8     4.604  1.858   14.908         1   1.095075   1.526832 0.049469132
#>        p_coef  p_coef_adj
#> 1 0.005683362 0.005683362
#> 2 0.003213778 0.005683362
```

Increase in dCt values by 1 (two-fold decrease in expression) gives the higher chance to be in the “After” group.

If genes should be displayed in alphabetical order, simply sort them:

```
log.reg.results.sorted <- log.reg.results[[1]] +
                          scale_y_discrete(limits = rev(sort(log.reg.results[[2]]$Gene)))
log.reg.results.sorted
```



Part C: Analysis of more than two groups

Many functions of the `RQdeltaCT` package work well with data containing more than two groups. To demonstrate this functionality, an example of analysis regarding data with three groups is presented below (variant with a comparison of independent groups). For the purpose of this analysis, a dataset named `data.Ct.3groups` included in the package was used. Of course, data with multiple groups can be imported to R using the options previously described in the chapter **Data import**.

```
data("data.Ct.3groups")
str(data.Ct.3groups)
#> 'data.frame':    2048 obs. of  5 variables:
#>  $ Sample: chr  "AAA1" "AAA10" "AAA12" "AAA13" ...
#>  $ Gene  : chr  "ANGPT1" "ANGPT1" "ANGPT1" "ANGPT1" ...
#>  $ Ct    : chr  "32.563" "34.648" "35.059" "37.135" ...
#>  $ Group : chr  "AAA" "AAA" "AAA" "AAA" ...
#>  $ FLAG  : chr  "OK" "OK" "Undetermined" "Undetermined" ...
table(data.Ct.3groups$Group)
#>
#>     AAA Control      VV
#>     760     528     760
```

The `data.Ct.3groups` dataset contains a table with gene expression data obtained for 20 genes and 104 samples using qPCR experiments with TaqMan assays. The samples are divided into three groups: Disease AAA (40 samples), Disease VV (40 samples), and Control (24 samples). This dataset is a part of the data used in the Article1 and Article2. This table is a data frame with 2048 rows and 5 variables (Sample, Gene, Ct, Group, and FLAG).

The workflow for multiple groups presented here is very similar to these for two groups presented in chapter **Part A: The workflow for analysis of independent groups of samples**. Therefore, this chapter includes the most important points and differences, and prior reading of this previous chapter is highly recommended.

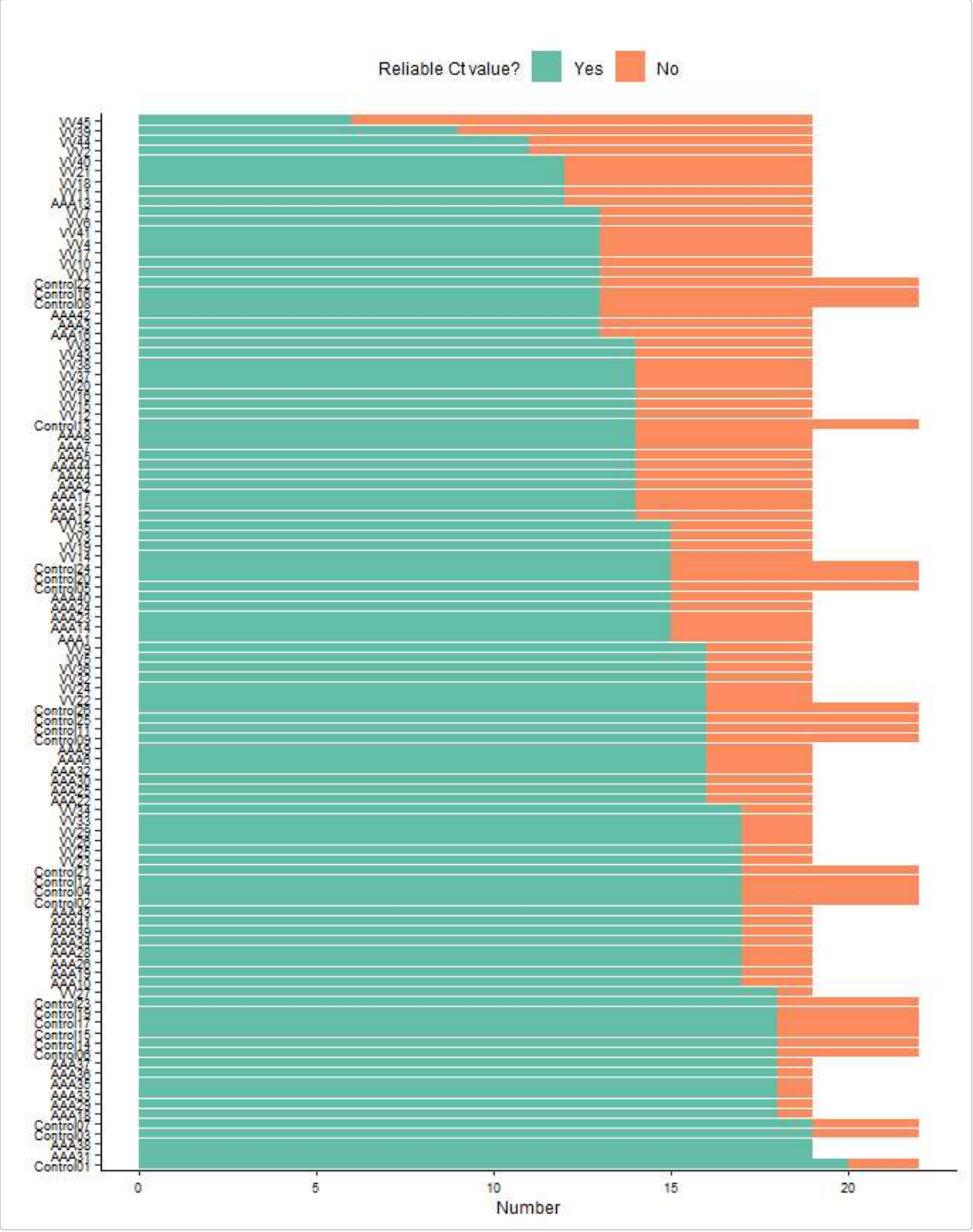
Quality control of raw Ct data - a multigroup variant of analysis

Two functions for quality control of raw Ct data: `control_Ct_barplot_sample()` (for quality control of samples) and `control_Ct_barplot_gene()` (for quality control of genes) can be used for data with more than two groups. Both functions label each Ct value as reliable or not, based on the reliability criteria established by the user. The default criteria will be used in this example:

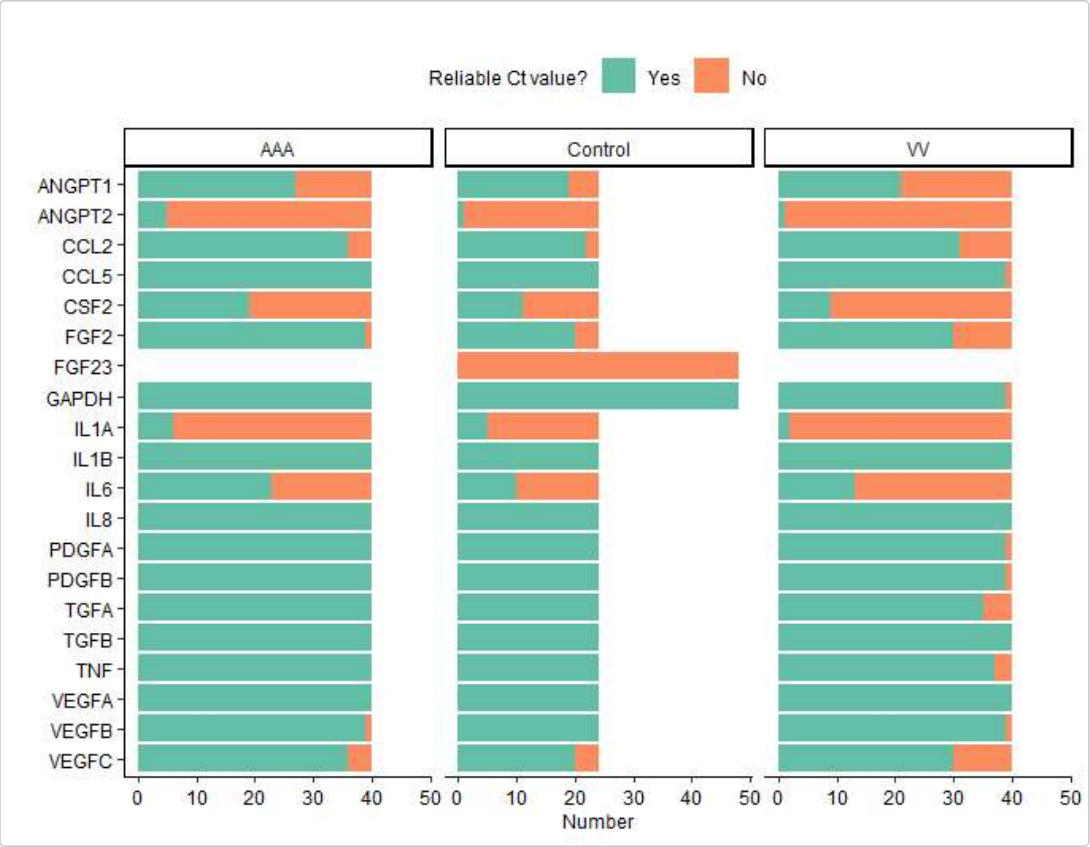
- a flag used for undetermined Ct values - “Undetermined”.
- a maximum of Ct value allowed - 35.
- a flag used in the Flag column for values that are unreliable - “Undetermined”.

```
sample.Ct.control.3groups <- control_Ct_barplot_sample(data = data.Ct.3groups,
                                                       flag.Ct = "Undetermined",
```

```
                                                maxCt = 35,
                                                flag = c("Undetermined"),
                                                axis.title.size = 9,
                                                axis.text.size = 7,
                                                plot.title.size = 9,
                                                legend.title.size = 9,
                                                legend.text.size = 9)
```




```
gene.Ct.control.3groups <- control_Ct_barplot_gene(data = data.Ct.3groups,
                                                   flag.Ct = "Undetermined",
                                                   maxCt = 35,
                                                   flag = c("Undetermined"),
                                                   axis.title.size = 9,
                                                   axis.text.size = 9,
                                                   plot.title.size = 9,
                                                   legend.title.size = 9,
                                                   legend.text.size = 9)
```



Remember: These functions do not perform data filtering, but only report numbers of Ct values labelled as reliable or not and present them graphically.

Conclusions:

- The results obtained in the examples show that the AAA and VV groups contain more samples than the Control group.
- Most Ct values are labelled as reliable.
- FGF23 was analysed only in the Control group and has all values labeled as unreliable; therefore, it is obvious that this gene should be excluded from the analysis.
- Some other genes also have many unreliable Ct values (e.g. ANGPT2, IL1A) and should probably be considered to be removed from the data.

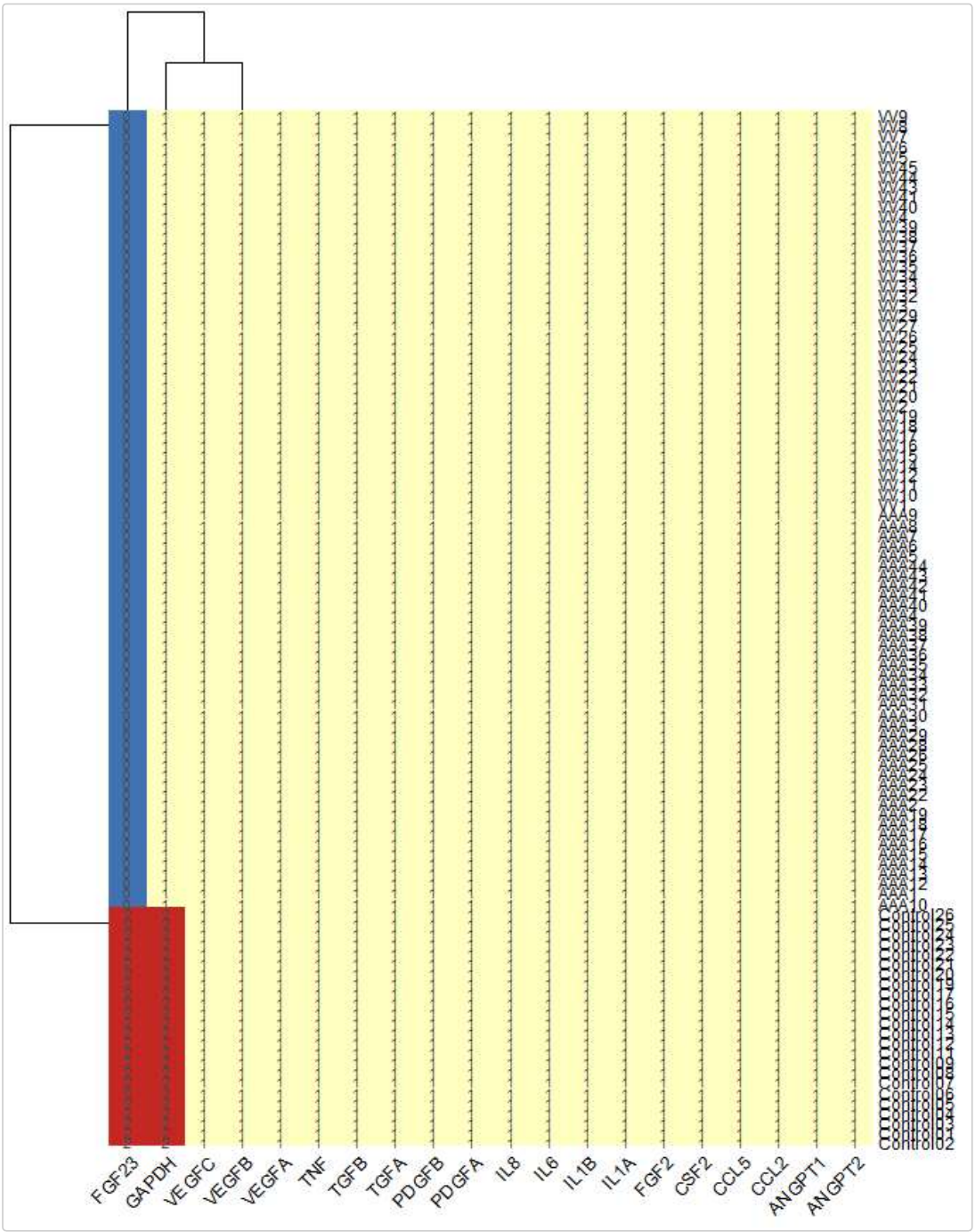
We can access tables with numbers of Ct values labelled as reliable (Yes) or unreliable (No), and with calculated fraction of unreliable Ct values in each gene:

```
head(sample.Ct.control.3groups[[2]])
#> # A tibble: 6 × 4
#>   Sample    Not.reliable Reliable Not.reliable.fraction
#>   <fct>            <int>    <int>                <dbl>
#> 1 VV45                13        6                0.684
#> 2 VV39                10        9                0.526
#> 3 Control08            9       13                0.409
#> 4 Control16            9       13                0.409
#> 5 Control22            9       13                0.409
#> 6 Control13            8       14                0.364
```

```
head(gene.Ct.control.3groups[[2]])
#> # A tibble: 6 × 5
#>   Gene   Group   Not.reliable Reliable Not.reliable.fraction
#>   <fct>  <fct>          <int>    <int>                <dbl>
#> 1 FGF23  Control           48        0                1
#> 2 ANGPT2 VV                39        1                0.975
#> 3 IL1A   VV                38        2                0.95
#> 4 ANGPT2 AAA               35        5                0.875
#> 5 IL1A   AAA               34        6                0.85
#> 6 CSF2   VV                31        9                0.775
```

The `control_heatmap()` function also can be used for data with multiple groups:

```
library(tidyverse)
library(pheatmap)
data("data.Ct.3groups")

# Vector of colors to fill the heatmap can be specified to fit the user needs:
colors <- c("#4575B4","#FFFFBF","#C32B23")
control_heatmap(data.Ct.3groups,
                sel.Gene = "all",
                colors = colors,
                show.colnames = TRUE,
                show.rownames = TRUE,
                fontsize = 9,
                fontsize.row = 9,
                angle.col = 45)
```



On the heatmap we can clearly observe that some of the samples have more Ct values (more technical replicates) than other samples. Genes FGF23 and GAPDH were investigated in duplicates in the Control group, while other genes have single Ct values.

Remember: The `control_heatmap()` works only if various numbers of replicates are in the data, otherwise error will appear, because inherited `pheatmap()` function can not deal with the situation where all values are equal.

In the example below, for filtering purposes, samples and genes with more than half of the unreliable data were identified:

```
# Finding samples with more than half of the unreliable Ct values.
low.quality.samples.3groups <- filter(sample.Ct.control.3groups[[2]], Not.reliable.fraction >
        0.5)$Sample
low.quality.samples.3groups <- as.vector(low.quality.samples.3groups)
low.quality.samples.3groups
#> [1] "VV45" "VV39"


# Finding genes with more than half of the unreliable Ct values in given group.
low.quality.genes.3groups <- filter(gene.Ct.control.3groups[[2]], Not.reliable.fraction > 0.5)$Gene
low.quality.genes.3groups <- unique(as.vector(low.quality.genes.3groups))
low.quality.genes.3groups
#> [1] "FGF23"  "ANGPT2" "IL1A"   "CSF2"   "IL6"
```

In the above examples, the VV45 and VV39 samples have more than half of the unreliable data. This criterion was also met by 5 genes (FGF23, ANGPT2, IL1A, CSF2, and IL6). Therefore, these samples and genes will be removed from the data in the next step of analysis.

Filtering of raw Ct data, collapsing technical replicates, and imputation of missing data - a multigroup variant of analysis

For these steps, a similar code can be used to that presented for two groups.

```
# Data filtering
data.CtF.3groups <- filter_Ct(data = data.Ct.3groups,
                              flag.Ct = "Undetermined",
                              maxCt = 35,
                              flag = c("Undetermined"),
                              remove.Gene = low.quality.genes.3groups,
                              remove.Sample = low.quality.samples.3groups)

# Collapsing technical replicates without imputation:
data.CtF.ready.3groups <- make_Ct_ready(data = data.CtF.3groups,
                                        imput.by.mean.within.groups = FALSE)
# A part of the data with missing values:
as.data.frame(data.CtF.ready.3groups)[25:30,]
#>    Group Sample ANGPT1   CCL2   CCL5   FGF2  GAPDH   IL1B    IL8  PDGFA  PDGFB
#> 25   AAA  AAA43 33.471 32.162 22.556 33.499 22.804 25.799 28.884 29.082 27.531
#> 26   AAA   AAA6 33.660 30.905 23.273 32.345 22.721 25.711 24.810 29.472 28.947
#> 27   AAA   AAA9 33.777 31.086 22.900 32.242 23.070 27.980 32.393 29.717 28.723
#> 28   AAA  AAA12     NA 32.077 24.332 31.318 24.677 26.697 29.426 29.889 28.120
#> 29   AAA  AAA13     NA 33.982 24.895     NA 23.973 27.697 32.464 33.008 30.780
#> 30   AAA  AAA15     NA 33.497 24.937 31.363 23.714 26.849 30.607 31.256 29.749
#>      TGFA   TGFB    TNF  VEGFA  VEGFB  VEGFC
#> 25 30.866 22.849 26.406 28.878 27.926 32.749
#> 26 28.651 22.194 26.855 27.769 28.062 32.800
#> 27 30.641 22.883 27.564 28.962 28.246 33.375
#> 28 31.990 23.580 27.622 29.613 28.867 31.011
#> 29 32.096 24.461 28.483 30.154 29.951     NA
#> 30 31.263 23.971 27.648 29.287 29.018 34.362

# Collapsing technical replicates with imputation:
data.CtF.ready.3groups <- make_Ct_ready(data = data.CtF.3groups,
                                        imput.by.mean.within.groups = TRUE)
# Missing values were imputed:
as.data.frame(data.CtF.ready.3groups)[25:30,]
#>    Group Sample   ANGPT1   CCL2   CCL5     FGF2  GAPDH   IL1B    IL8  PDGFA
#> 25   AAA  AAA43 33.47100 32.162 22.556 33.49900 22.804 25.799 28.884 29.082
#> 26   AAA   AAA6 33.66000 30.905 23.273 32.34500 22.721 25.711 24.810 29.472
#> 27   AAA   AAA9 33.77700 31.086 22.900 32.24200 23.070 27.980 32.393 29.717
#> 28   AAA  AAA12 30.46893 32.077 24.332 31.31800 24.677 26.697 29.426 29.889
#> 29   AAA  AAA13 30.46893 33.982 24.895 32.04556 23.973 27.697 32.464 33.008
#> 30   AAA  AAA15 30.46893 33.497 24.937 31.36300 23.714 26.849 30.607 31.256
#>     PDGFB   TGFA   TGFB    TNF  VEGFA  VEGFB    VEGFC
```

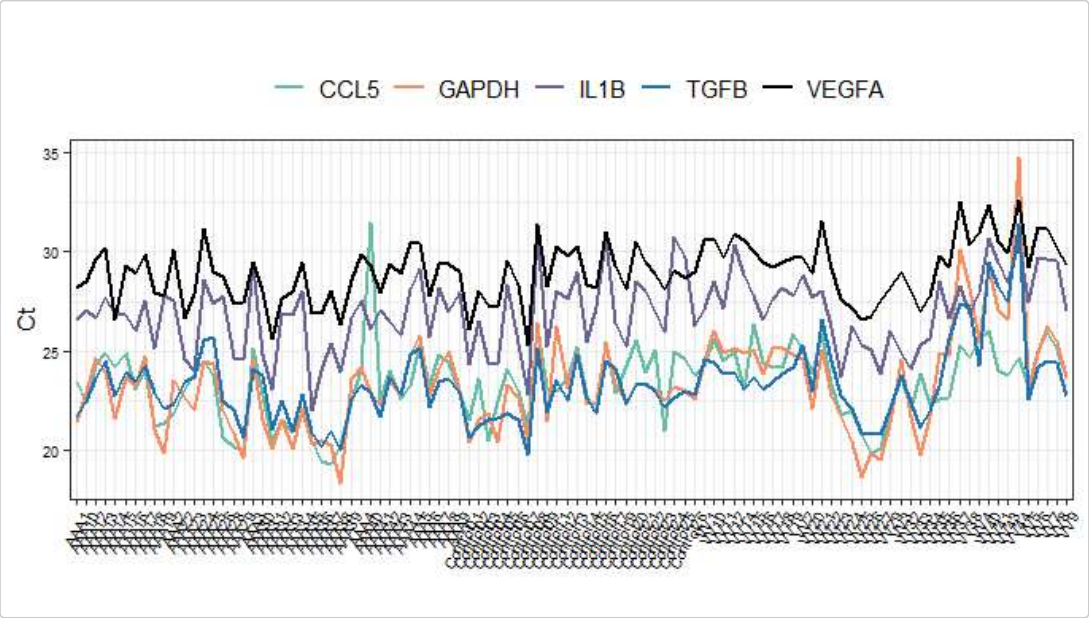
```
#> 25 27.531 30.866 22.849 26.406 28.878 27.926 32.74900
#> 26 28.947 28.651 22.194 26.855 27.769 28.062 32.80000
#> 27 28.723 30.641 22.883 27.564 28.962 28.246 33.37500
#> 28 28.120 31.990 23.580 27.622 29.613 28.867 31.01100
#> 29 30.780 32.096 24.461 28.483 30.154 29.951 31.99058
#> 30 29.749 31.263 23.971 27.648 29.287 29.018 34.36200
```

The data imputation process can significantly influence the data; therefore, no default value was set to the `imput.by.mean.within.groups` parameter to force the specification by the user. If there are missing data for a certain gene in the entire group, they will not be imputed and will remain NA.

Reference gene selection - a multigroup variant of analysis

This step is also similar to that for two groups. A list of group names needs to be provided in the `groups` argument. In the example below, five genes are tested for suitability to be a reference gene:

```
library(ctrlGene)
# Remember that the number of colors in col parameter should be equal to the number of tested genes:
ref.3groups <- find_ref_gene(data = data.CtF.ready.3groups,
                             groups = c("AAA","Control","VV"),
                             candidates = c("CCL5", "GAPDH","IL1B","TGFB", "VEGFA"),
                             col = c("#66c2a5", "#fc8d62","#6A6599", "#1F77B4", "black"),
                             angle = 60,
                             axis.text.size = 7,
                             norm.finder.score = TRUE,
                             genorm.score = TRUE)
```




```
ref.3groups[[2]]
#>         Gene    min    max       sd      var NormFinder_score geNorm_score
#> 1       CCL5 19.295 31.508 1.900276 3.611049             0.41     1.487003
#> 2      GAPDH 18.314 34.796 2.476982 6.135439             0.41           NA
#> 3       IL1B 22.043 30.892 1.904221 3.626056             0.37     1.355319
#> 4       TGFB 19.801 31.415 1.921946 3.693877             0.41           NA
#> 5      VEGFA 25.323 32.602 1.490384 2.221245             0.17     1.222317
#> 6 GAPDH-TGFB     NA     NA       NA       NA               NA     1.126981
```

Among tested genes, VEGFA seems to have the best characteristics to be a reference gene (low variance, sd, NormFinder and geNorm scores).

Data normalization using reference gene - a multigroup variant of analysis

The code for this step is similar as that for two groups.

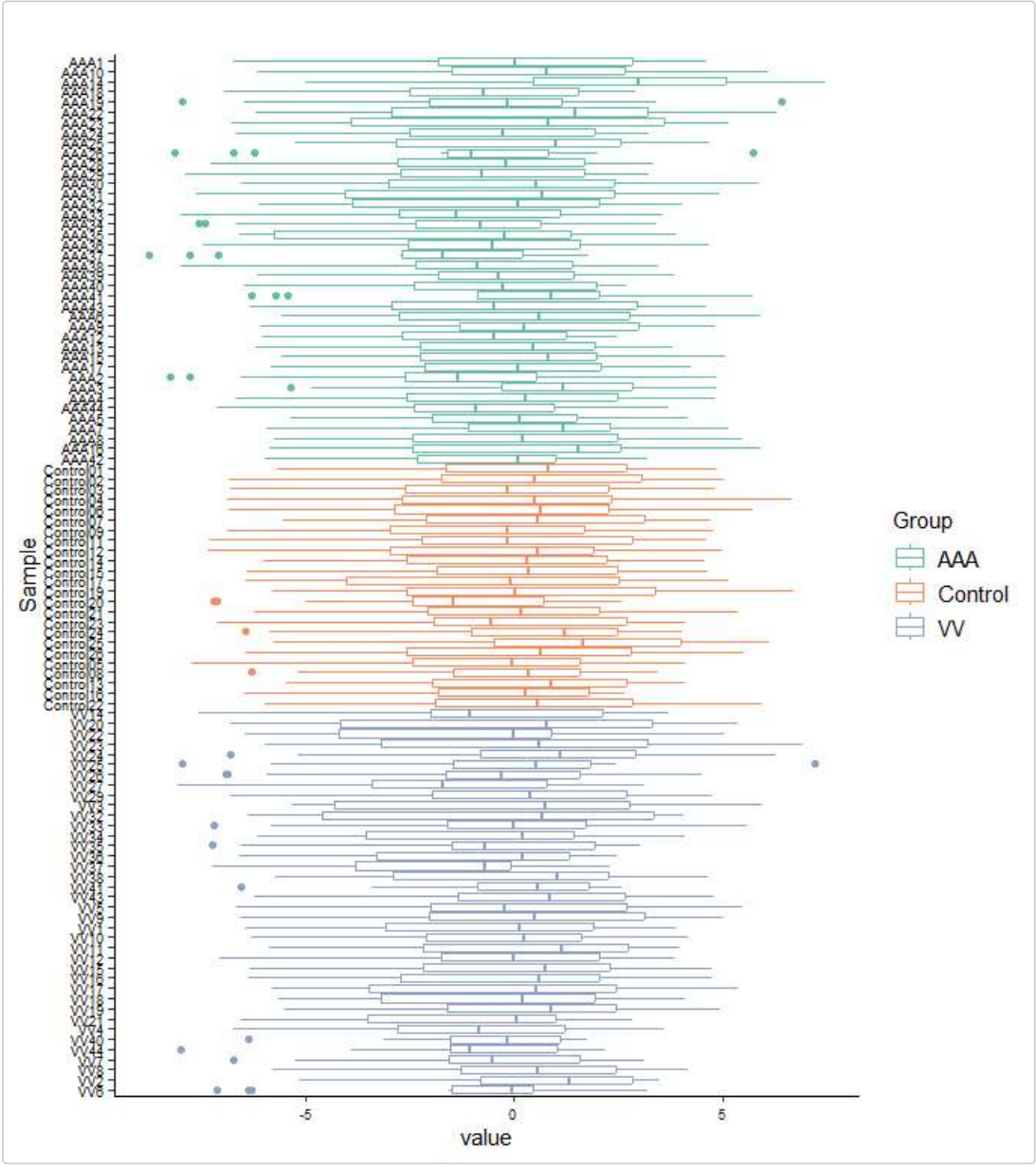
```
# For 2-dCt method:
data.dCt.exp.3groups <- delta_Ct(data = data.CtF.ready.3groups,
                                 normalise = TRUE,
                                 ref = "VEGFA",
                                 transform = TRUE)

# For 2-ddCt method:
data.dCt.3groups <- delta_Ct(data = data.CtF.ready.3groups,
                             normalise = TRUE,
                             ref = "VEGFA",
                             transform = FALSE)
```

In the rare scenario where unnormalised data should be analysed, the `normalise` parameter should be set to `FALSE`.

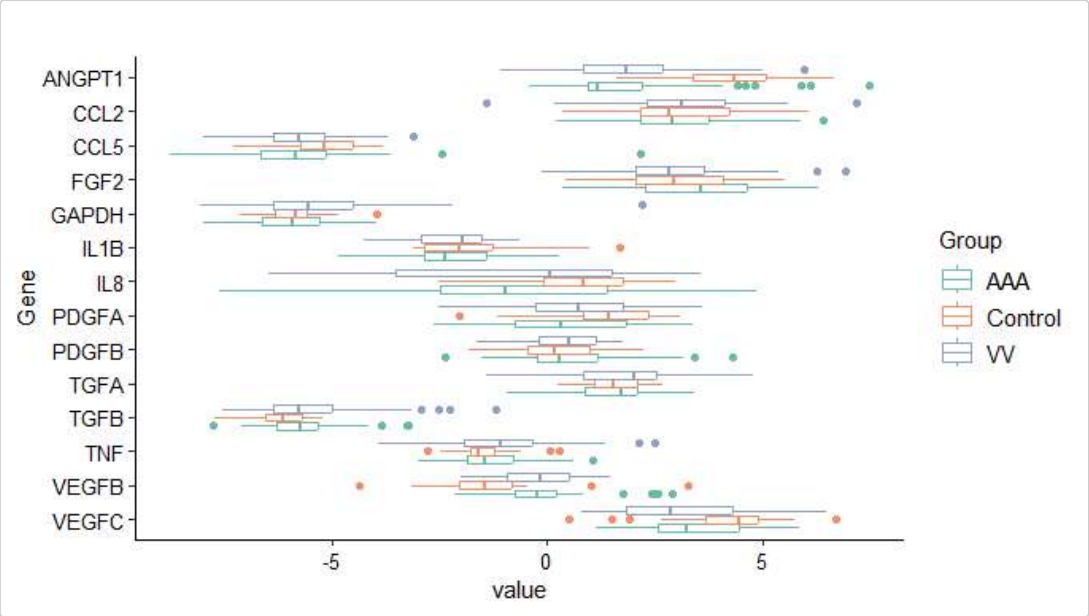
Quality control and filtering of normalized Ct data - a multigroup variant of analysis

If more than two groups are present in the data, for proper action of the `control_boxplot_sample()`, `control_boxplot_gene()`, and `control_pca_sample()` functions, the vector of colors must be specified in the `colors` argument.

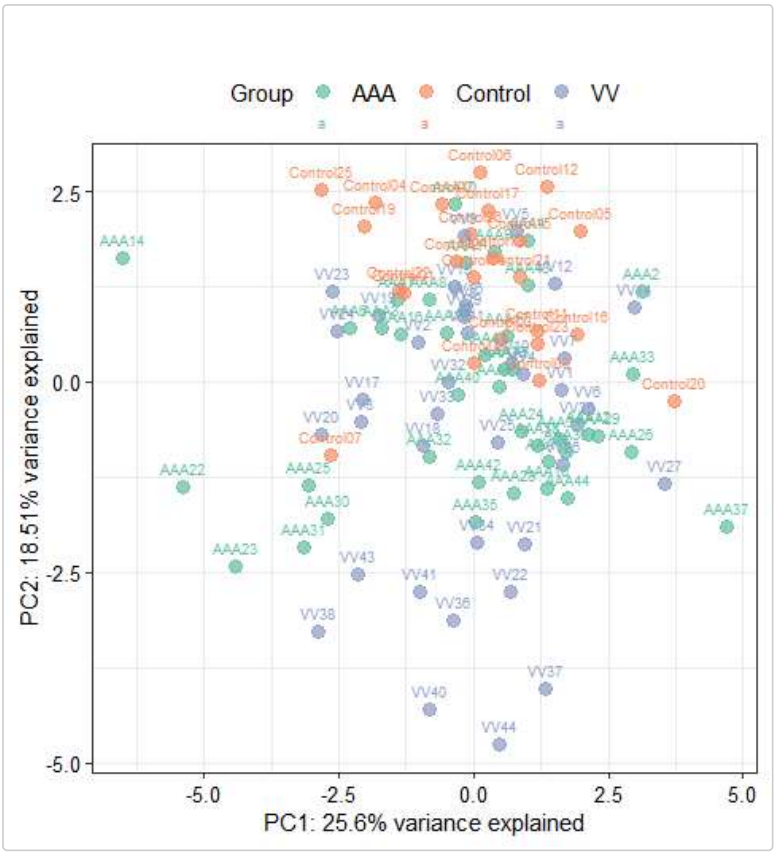
```
control_boxplot_sample_3groups <- control_boxplot_sample(data = data.dCt.3groups,
                                                         colors = c("#66c2a5", "#fc8d62", "#8DA0CB"),
                                                         axis.text.size = 7)
```



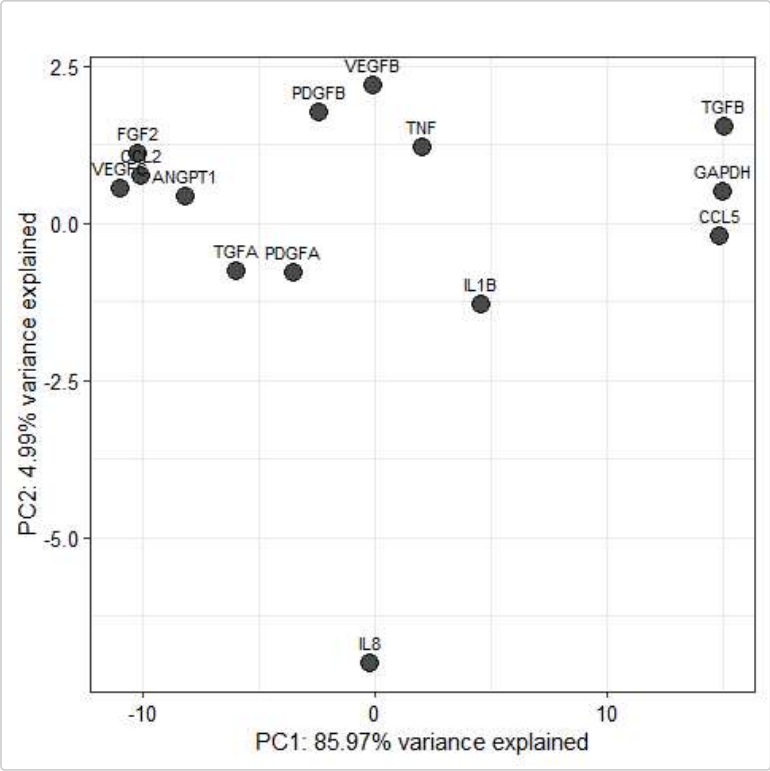
```r
control_boxplot_gene_3groups <- control_boxplot_gene(data = data.dCt.3groups,
                                                     by.group = TRUE,
                                                     colors = c("#66c2a5", "#fc8d62", "#8DA0CB"),
                                                     axis.text.size = 10)
```




```r
control.pca.sample.3groups <- control_pca_sample(data = data.dCt.3groups,
                                                 colors = c("#66c2a5", "#fc8d62", "#8DA0CB"),
                                                 point.size = 3,
                                                 label.size = 2.5,
                                                 legend.position = "top")
```



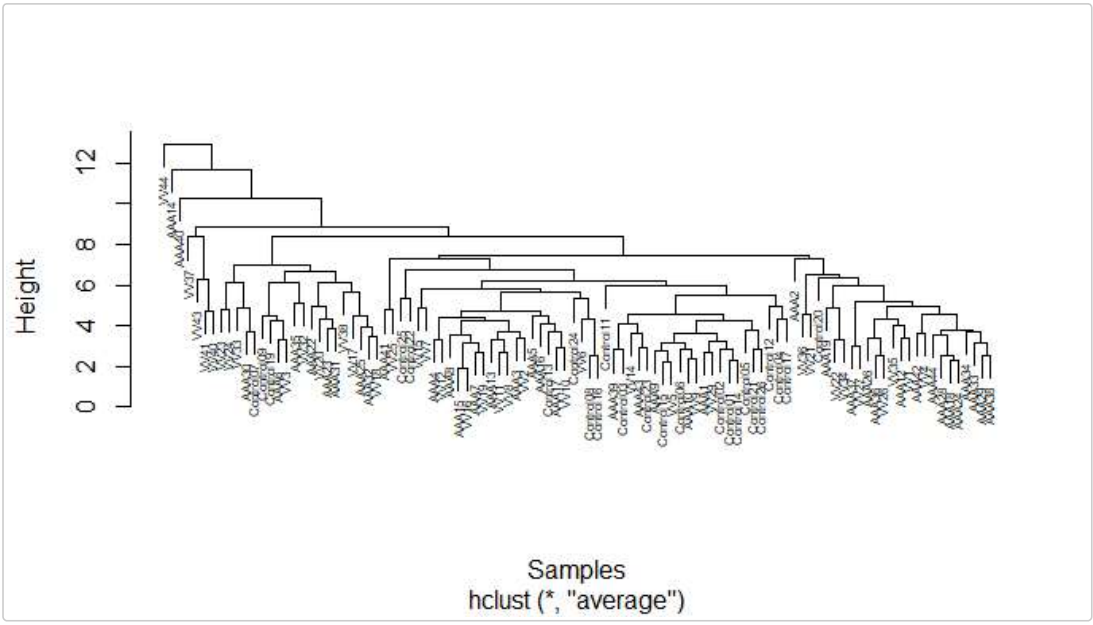

```r
control.pca.gene.3groups <- control_pca_gene(data = data.dCt.3groups)
```



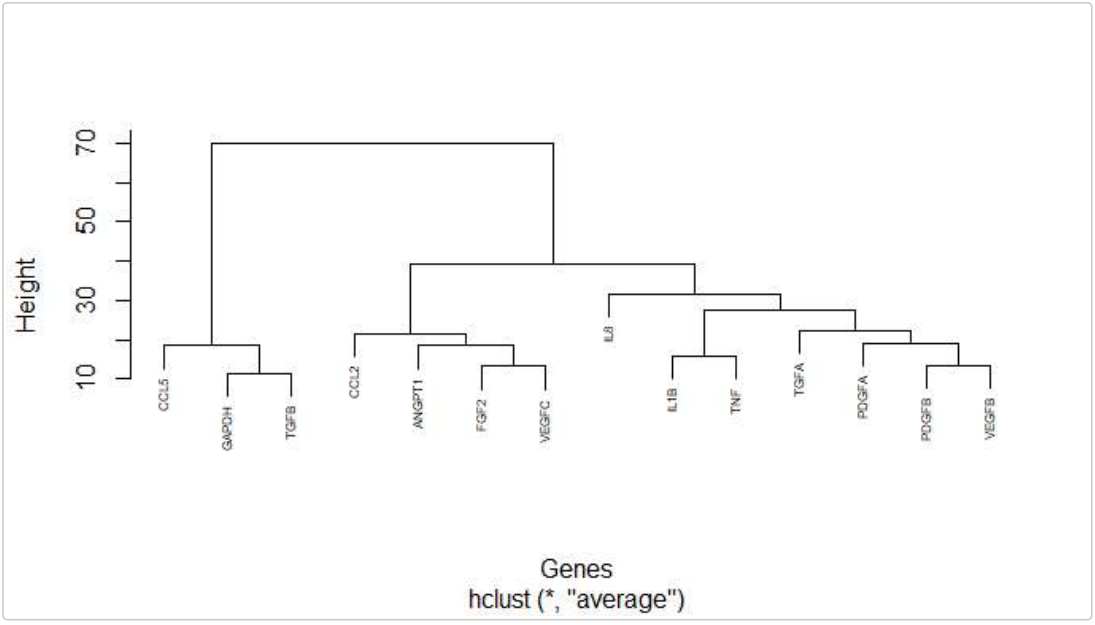
NOTE: PCA algorithm can not deal with missing data (NAs); therefore, variables with NA values are automatically removed before PCA analysis. If at least one NA value occurs in all variables in at least one of the compared group, the analysis can not be done. The imputation of missing data will avoid this issue. Also, a minimum of three samples or genes in the data are required for analysis.

The code for hierarchical clustering also does not need substantial modifications:

```
control_cluster_sample(data = data.dCt.3groups,
                       method.dist = "euclidean",
                       method.clust = "average",
                       label.size = 0.5)
```




```
control_cluster_gene(data = data.dCt.3groups,
                     method.dist = "euclidean",
                     method.clust = "average",
                     label.size = 0.5)
```



NOTE: Minimum three samples or genes in data is required for clustering analysis.

Data filtering after quality control - a multigroup variant of analysis

If any sample or gene was decided to be removed from the data with multiple groups, the `filter_transformed_data()` function can be used as previously:

```
data.dCtF.3groups <- filter_transformed_data(data = data.dCt.3groups,
                                             remove.Sample = c("AAA14"))
```

Relative quantification: 2^{-dCt} and 2^{-ddCt} methods - a multigroup variant of analysis

Currently, the `RQ_dCt()` and `RQ_ddCt()` functions work only with two groups (study group and reference group) specified in the `group.study` and `group.ref` arguments. The functionalities for the analysis of more than two groups will be extended in the future.

Example of using the 2^{-dCt} method for comparison of VV vs. Control group

```
data.dCt.exp.3groups <- delta_Ct(data = data.CtF.ready.3groups,
                                 ref = "VEGFA",
                                 transform = TRUE)
library(coin)
results.dCt.3groups <- RQ_dCt(data = data.dCt.exp.3groups,
                              do.tests = TRUE,
                              group.study = "VV",
                              group.ref = "Control")

# Obtained table can be sorted by, e.g. p values from the Mann-Whitney U test:
head(as.data.frame(arrange(results.dCt.3groups, MW_test_p)))
#>     Gene Control_mean    VV_mean  Control_sd      VV_sd Control_norm_p
#> 1 ANGPT1   0.07843201  0.4276454  0.07877386  0.4454233   8.006735e-05
#> 2  VEGFB   3.74243468  1.3192002  4.05446247  0.7828158   7.813077e-07
#> 3  VEGFC   0.10083276  0.1883991  0.15339340  0.1528416   1.368336e-07
#> 4   TGFB  81.67181583 59.9789149 42.24994621 41.0469440   1.442374e-03
#> 5   CCL5  49.83556638 69.2702784 38.83838810 52.5417712   3.580423e-04
#> 6    IL8   0.95537499  8.7445761  1.24131170 16.7218789   1.483006e-06
#>      VV_norm_p       FCh   log10FCh     t_test_p t_test_stat    MW_test_p
#> 1 1.242687e-06 5.4524342  0.7365904 2.820521e-05   -4.717524 8.304374e-07
#> 2 1.027350e-03 0.3524979 -0.4528435 7.952080e-03    2.894107 1.768299e-05
#> 3 2.078404e-03 1.8684317  0.2714772 3.313052e-02   -2.192486 4.413183e-03
#> 4 4.408624e-02 0.7343894 -0.1340736 5.219443e-02    1.990965 5.459396e-02
#> 5 1.974077e-05 1.3899767  0.1430075 1.003532e-01   -1.669590 5.833259e-02
#> 6 2.157575e-09 9.1530302  0.9615649 6.891373e-03   -2.858994 6.028024e-02
#>   MW_test_stat t_test_p_adj MW_test_p_adj
```

```
#> 1    -4.928075 0.0003948729  1.162612e-05
#> 2     4.292302 0.0371097078  1.237809e-04
#> 3    -2.847011 0.1159568224  2.059485e-02
#> 4     1.922094 0.1461443908  1.406539e-01
#> 5    -1.893190 0.2074583979  1.406539e-01
#> 6    -1.878738 0.0371097078  1.406539e-01
```

Example of using the 2^{-ddCt} method for comparison of VV vs. Control group

```
data.dCt.3groups <- delta_Ct(data = data.CtF.ready.3groups,
                             ref = "VEGFA",
                             transform = FALSE)
library(coin)
results.ddCt.3groups <- RQ_ddCt(data = data.dCt.3groups,
                                group.study = "VV",
                                group.ref = "Control",
                                do.tests = TRUE)

# Obtained table can be sorted by, e.g. p values from the Mann-Whitney U test:
head(as.data.frame(arrange(results.ddCt.3groups, MW_test_p)))
#>     Gene Control_mean    VV_mean Control_sd     VV_sd Control_norm_p
#> 1 ANGPT1    4.2454189  1.9176830  1.2988093 1.5523854    0.904482961
#> 2  VEGFB   -1.3487500 -0.1703222  1.4244457 0.8273129    0.007826011
#> 3  VEGFC    4.1123583  3.0042211  1.3710167 1.4608126    0.054543696
#> 4   TGFB   -6.2007083 -5.4181842  0.6489496 1.4408463    0.396006299
#> 5   CCL5   -5.2874583 -5.7657817  1.0038688 1.0330656    0.317980588
#> 6    IL8    0.8040833 -0.7316316  1.4227554 2.8220314    0.679854574
#>     VV_norm_p       ddCt       FCh   log10FCh     t_test_p t_test_stat
#> 1 0.507221576 -2.3277359 5.0201689  0.7007183 4.030165e-08    6.365873
#> 2 0.727511574  1.1784278 0.4418327 -0.3547421 8.287595e-04   -3.679797
#> 3 0.060486057 -1.1081373 2.1556714  0.3335826 3.910982e-03    3.021821
#> 4 0.001016122  0.7825241 0.5813488 -0.2355632 5.157434e-03   -2.912659
#> 5 0.461705168 -0.4783233 1.3931237  0.1439897 7.677790e-02    1.806930
#> 6 0.028799148 -1.5357149 2.8993207  0.4622963 6.346543e-03    2.832678
#>      MW_test_p MW_test_stat t_test_p_adj MW_test_p_adj
#> 1 8.304374e-07     4.928075 5.642232e-07  1.162612e-05
#> 2 1.768299e-05    -4.292302 5.801317e-03  1.237809e-04
#> 3 4.413183e-03     2.847011 1.777032e-02  2.059485e-02
#> 4 5.459396e-02    -1.922094 1.777032e-02  1.406539e-01
#> 5 5.833259e-02     1.893190 1.194323e-01  1.406539e-01
#> 6 6.028024e-02     1.878738 1.777032e-02  1.406539e-01
```

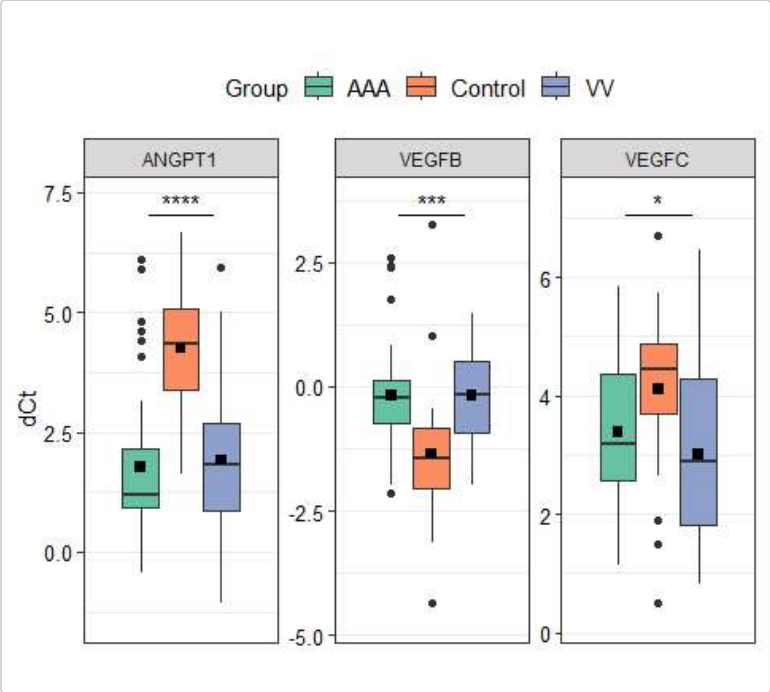
Final visualisations - a multigroup variant of analysis

Five functions: `FCh_plot()`, `results_volcano()`, `results_barplot()`, `results_boxplot()`, and `results_heatmap()` were created for visualisation of results. Two of them, `FCh_plot()` and `results_volcano()`, process data returned from group-specific `RQ_dCt()` and `RQ_ddCt()` functions; therefore, multigroup analysis is not applicable. The remaining three functions (`results_barplot()`, `results_boxplot()`, and `results_heatmap()`) can be used successfully for data with multiple groups, but the user must provide a vector of colors for groups.

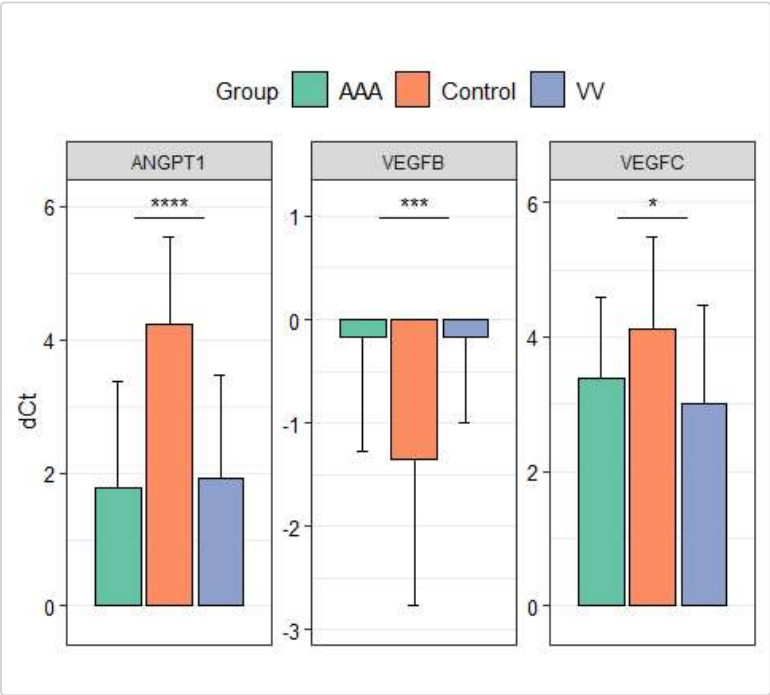
However, the `results_barplot()` and `results_boxplot()` functions are designed to draw the statistical significance labels for two groups (see The `results_boxplot()` function and The `results_barplot()` function chapters). If more groups are drawn, the labels will appear above all boxplots. This is not an optimal solution:

```
final_boxplot_3groups <- results_boxplot(data = data.dCtF.3groups,
                                 sel.Gene = c("ANGPT1","VEGFB", "VEGFC"),
                                 colors = c("#66c2a5", "#fc8d62", "#8DA0CB"),
                                 by.group = TRUE,
                                 signif.show = TRUE,
                                 signif.labels = c("****","***","*"),
                                 signif.dist = 1.05,
                                 faceting = TRUE,
                                 facet.row = 1,
                                 facet.col = 4,
                                 y.exp.up = 0.1,
```

```
                        angle = 20,
                        y.axis.title = "dCt")
```




```
final_barplot_3groups <- results_barplot(data = data.dCtF.3groups,
                        sel.Gene = c("ANGPT1","VEGFB","VEGFC"),
                        colors = c("#66c2a5", "#fc8d62", "#8DA0CB"),
                        signif.show = TRUE,
                        signif.labels = c("****","***","*"),
                        angle = 30,
                        signif.dist = 1.05,
                        faceting = TRUE,
                        facet.row = 1,
                        facet.col = 4,
                        y.exp.up = 0.1,
                        y.axis.title = "dCt")
```

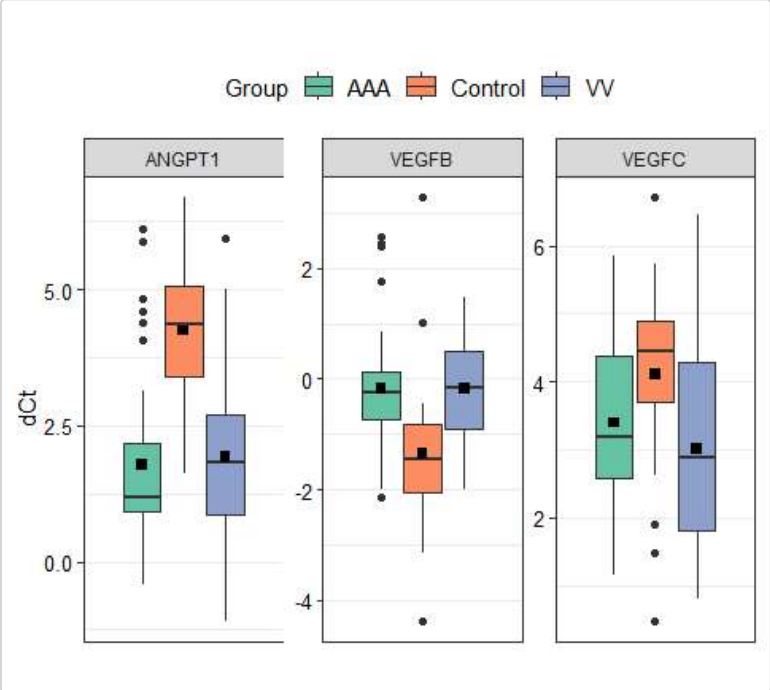


Typically, statistical significance labels for each **pair** of groups (above each pair of distributions or bars) are needed to show significance status. To properly locate labels on the plot, customized coordinates for labels must be specified for each pair outside of the `results_boxplot()` or `results_barplot()` functions. Below, a solutions for these situations is provided.

Solution for the `results_boxplot()` function

Step 1: The `results_boxplot()` function should be run with `signif.show = FALSE` to avoid drawing labels:

```
# Draw plot without statistical significance labels:
final_boxplot_3groups <- results_boxplot(data = data.dCtF.3groups,
                                         sel.Gene = c("ANGPT1","VEGFB", "VEGFC"),
                                         colors = c("#66c2a5", "#fc8d62", "#8DA0CB"),
                                         by.group = TRUE,
                                         signif.show = FALSE, # It avoids drawing labels
                                         faceting = TRUE,
                                         facet.row = 1,
                                         facet.col = 4,
                                         y.exp.up = 0.2,
                                         angle = 20,
                                         y.axis.title = "dCt")
```



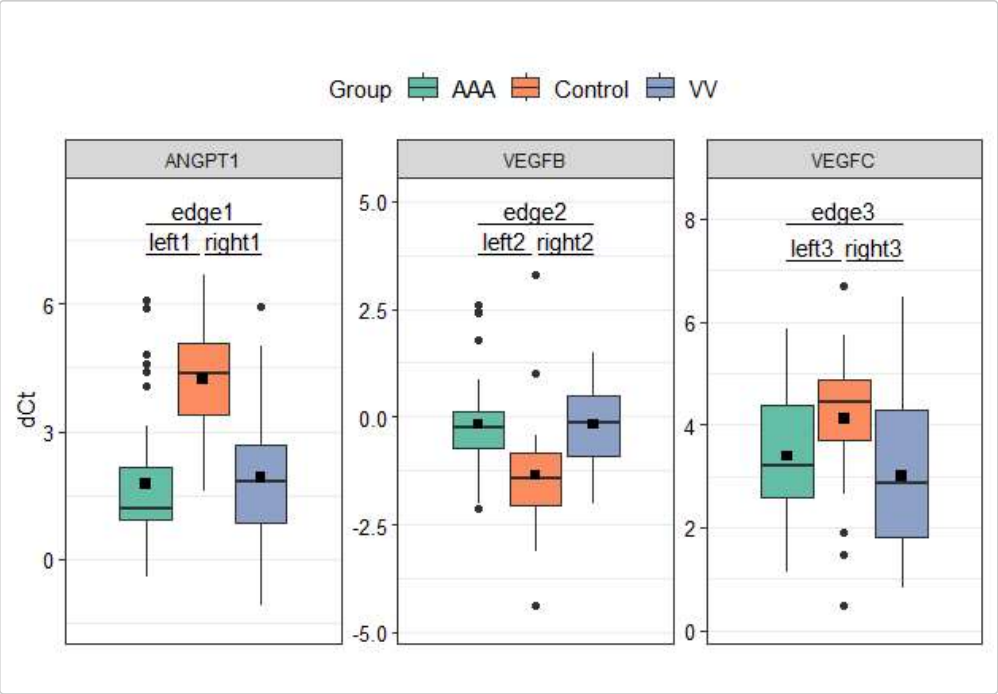
Step2: Data frames with coordinates for statistical significance labels for each pair of distributions (left, right, edges) are subsequently prepared:

```
# Prepare expression data:
data <- pivot_longer(data.dCtF.3groups,
                     !c(Sample, Group),
                     names_to = "Gene" ,
                     values_to = "value")

# filter for genes:
data <- filter(data, Gene %in% c("ANGPT1","VEGFB", "VEGFC"))

# Find maximum value in each group:
label.height <- data %>%
                group_by(Gene) %>%
                summarise(height = max(value), .groups = "keep")

# Prepare empty data frame:
data.label.empty <- data.frame(matrix(nrow = length(unique(label.height$Gene)), ncol = 4))
rownames(data.label.empty) <- label.height$Gene
colnames(data.label.empty) <- c("x", "xend", "y", "annotation")
data.label.empty$Gene <- rownames(data.label.empty)

# Fill a data frame with coordinates for right pair:
data.label.right <- data.label.empty
data.label.right$x <- rep(1.01, nrow(data.label.right))
data.label.right$xend <- rep(1.25, nrow(data.label.right))
data.label.right$y <- label.height$height + 0.5
```

```
data.label.right$annotation <- c("right1","right2","right3")

# Fill a data frame with coordinates for left pair:
data.label.left <- data.label.empty
data.label.left$x <- rep(0.98, nrow(data.label.left))
data.label.left$xend <- rep(0.75, nrow(data.label.left))
data.label.left$y <- label.height$height + 0.5
data.label.left$annotation <- c("left1","left2","left3")

# Fill a data frame with coordinates for edge pair:
data.label.edge <- data.label.empty
data.label.edge$x <- rep(0.75, nrow(data.label.edge))
data.label.edge$xend <- rep(1.25, nrow(data.label.edge))
data.label.edge$y <- label.height$height + 1.2
data.label.edge$annotation <- c("edge1","edge2","edge3")
```

Step 3: Plot with customized localization of labels can be drawn using three prepared data frames:

```
final_boxplot_3groups +
  geom_signif(
          stat = "identity",
          data = data.label.right,
          aes(x = x,
              xend = xend,
              y = y,
              yend = y,
              annotation = annotation),
          color = "black",
          manual = TRUE) +

  geom_signif(
          stat = "identity",
          data = data.label.left,
          aes(x = x,
              xend = xend,
              y = y,
              yend = y,
              annotation = annotation),
          color = "black",
          manual = TRUE) +

   geom_signif(
          stat = "identity",
          data = data.label.edge,
          aes(x = x,
              xend = xend,
              y = y,
              yend = y,
              annotation = annotation),
          color = "black",
          manual = TRUE) +

  scale_y_continuous(expand = expansion(mult = c(0.1, 0.12))) # it makes space for labels
```

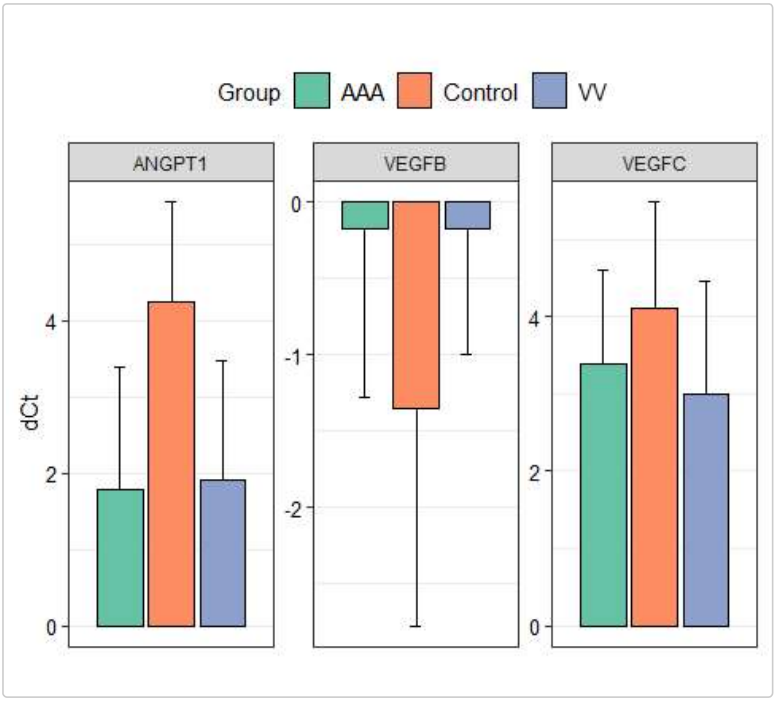


Solution for the `results_barplot()` function

Firstly, the `results_barplot()` function should be run with `signif.show = FALSE` to avoid drawing labels, and then data frames with coordinates for statistical significance labels are added for each pair of bars (left, right, edges):

Plot without default labels:

```
final_barplot_3groups <- results_barplot(data = data.dCtF.3groups,
                                         sel.Gene = c("ANGPT1","VEGFB","VEGFC"),
                                         colors = c("#66c2a5", "#fc8d62", "#8DA0CB"),
                                         signif.show = FALSE,
                                         angle = 30,
                                         faceting = TRUE,
                                         facet.row = 1,
                                         facet.col = 4,
                                         y.exp.up = 0.1,
                                         y.axis.title = "dCt")
```



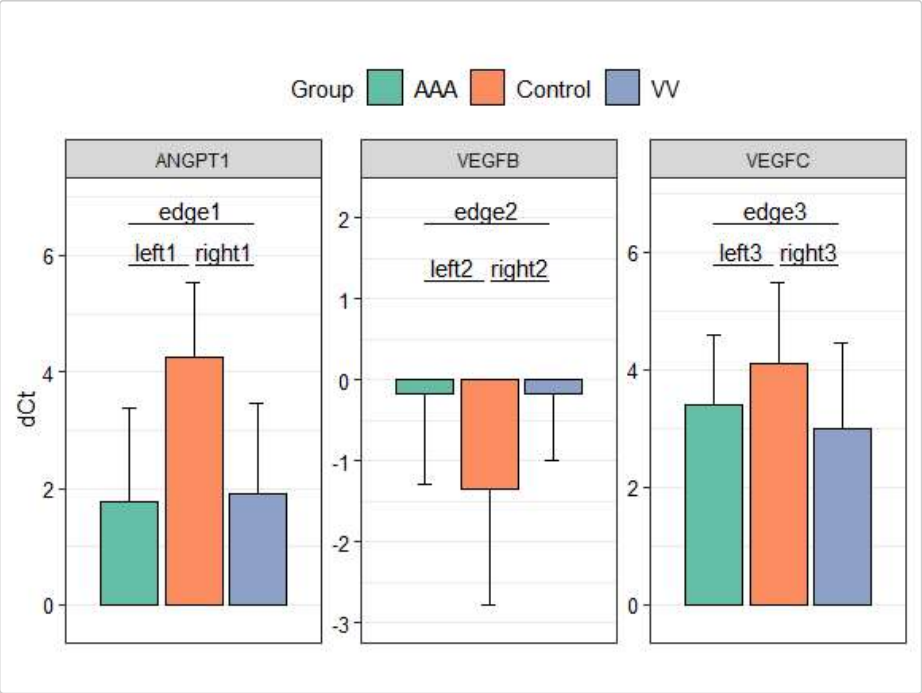
Preparation of data frames with specified coordinates:

```
# Prepare expression data:
data <- pivot_longer(data.dCtF.3groups,
                     !c(Sample, Group),
                     names_to = "Gene" ,
                     values_to = "value")

# filter for genes:
data <- filter(data, Gene %in% c("ANGPT1","VEGFB", "VEGFC"))

# Calculate mean and standard deviation for each group:
data.mean <- data %>%
             group_by(Group, Gene) %>%
             summarise(mean = mean(value, na.rm = TRUE), .groups = "keep")

data.sd <- data %>%
           group_by(Group, Gene) %>%
           summarise(sd = sd(value, na.rm = TRUE), .groups = "keep")

data.mean$sd <- data.sd$sd

#Find the highest values:
label.height <- data.mean %>%
                mutate(max = mean + sd) %>%
                group_by(Gene) %>%
                summarise(height = max(max, na.rm = TRUE), .groups = "keep")

# Prepare empty data frame:
data.label.empty <- data.frame(matrix(nrow = length(unique(data.mean$Gene)), ncol = 4))
rownames(data.label.empty) <- unique(data.mean$Gene)
colnames(data.label.empty) <- c("x", "xend", "y", "annotation")
data.label.empty$Gene <- rownames(data.label.empty)

# Fill a data frame with coordinates for left pair:
data.label.left <- data.label.empty
data.label.left$x <- rep(0.97, nrow(data.label.left))
data.label.left$xend <- rep(0.7, nrow(data.label.left))
data.label.left$y <- label.height$height + 0.3
data.label.left$annotation <- c("left1","left2","left3")

# Fill a data frame with coordinates for left pair:
data.label.right <- data.label.empty
data.label.right$x <- rep(1.01, nrow(data.label.right))
data.label.right$xend <- rep(1.28, nrow(data.label.right))
data.label.right$y <- label.height$height + 0.3
data.label.right$annotation <- c("right1","right2","right3")

# Fill a data frame with coordinates for edge pair:
data.label.edge <- data.label.empty
data.label.edge$x <- rep(0.7, nrow(data.label.edge))
data.label.edge$xend <- rep(1.28, nrow(data.label.edge))
data.label.edge$y <- label.height$height + 1
data.label.edge$annotation <- c("edge1","edge2","edge3")
```

And finally the plot with labels can be drawn:

```
final_barplot_3groups +
  geom_signif(
          stat = "identity",
          data = data.label.left,
          aes(x = x,
              xend = xend,
              y = y,
              yend = y,
              annotation = annotation),
          color = "black",
          manual = TRUE) +

   geom_signif(
```

```
          stat = "identity",
          data = data.label.right,
          aes(x = x,
              xend = xend,
              y = y,
              yend = y,
              annotation = annotation),
          color = "black",
          manual = TRUE) +

   geom_signif(
          stat = "identity",
          data = data.label.edge,
          aes(x = x,
              xend = xend,
              y = y,
              yend = y,
              annotation = annotation),
          color = "black",
          manual = TRUE) +

  scale_y_continuous(expand = expansion(mult = c(0.1, 0.12)))
```

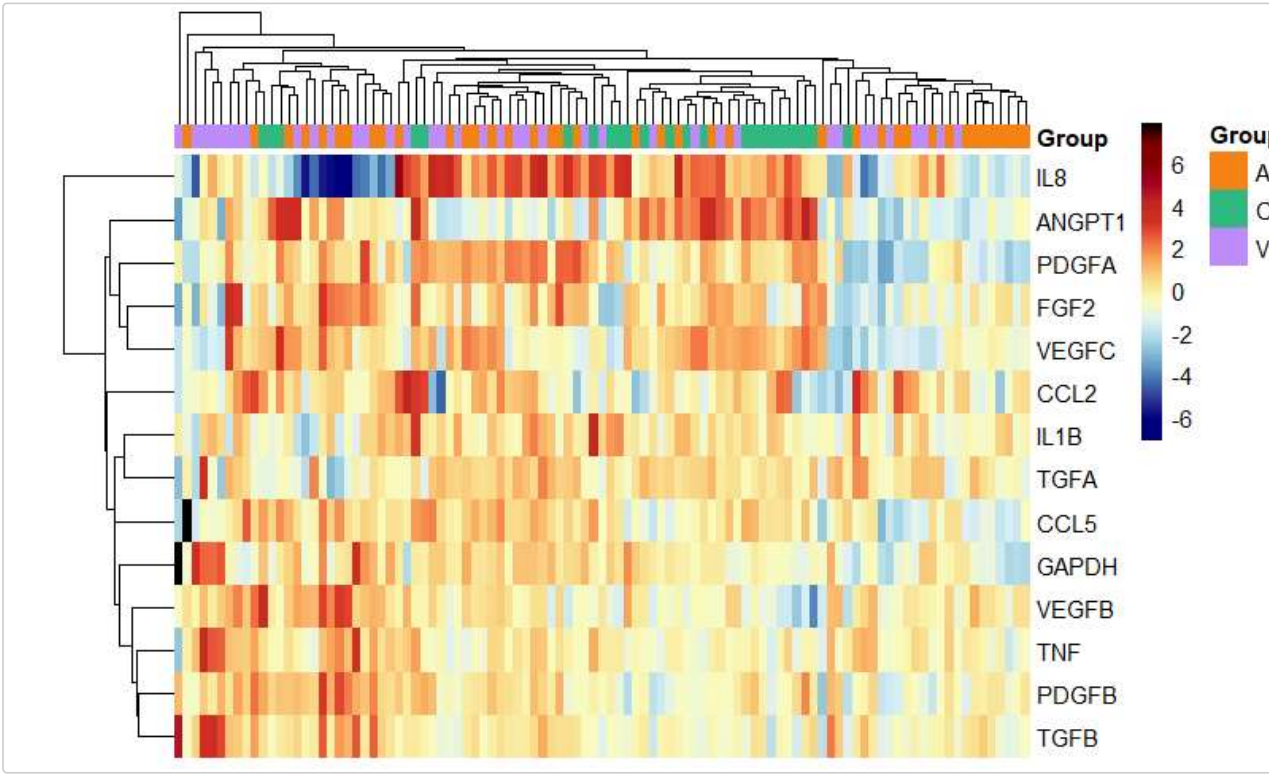


Note: For generating a faceted plot without a color legend, but with group names displaying as annotations on the x-axis, the solutions provided in chapters The `results_boxplot()` function and The `results_barplot()` function can be adapted.

The `results_heatmap()` function also works well with data containing more than two groups. Remember to create a named list with colors for the group annotation and pass it into `col.groups` argument.

```
# Create named list with colors for groups annotation:
colors.for.groups = list("Group" = c("AAA"="#f98517","Control"="#33b983", "VV"="#bf8cfc"))
# Vector of colors for heatmap:
colors <- c("navy","navy","#313695","#4575B4","#74ADD1","#ABD9E9",
            "#E0F3F8","#FFFFBF","#FEE090","#FDAE61","#F46D43",
            "#D73027","#C32B23","#A50026","#8B0000",
            "#7E0202","#000000")
results_heatmap(data.dCtF.3groups,
                sel.Gene = "all",
                col.groups = colors.for.groups,
                colors = colors,
                show.colnames = FALSE,
                show.rownames = TRUE,
```

```
                fontsize = 11,
                fontsize.row = 11,
                cellwidth = 4)
```

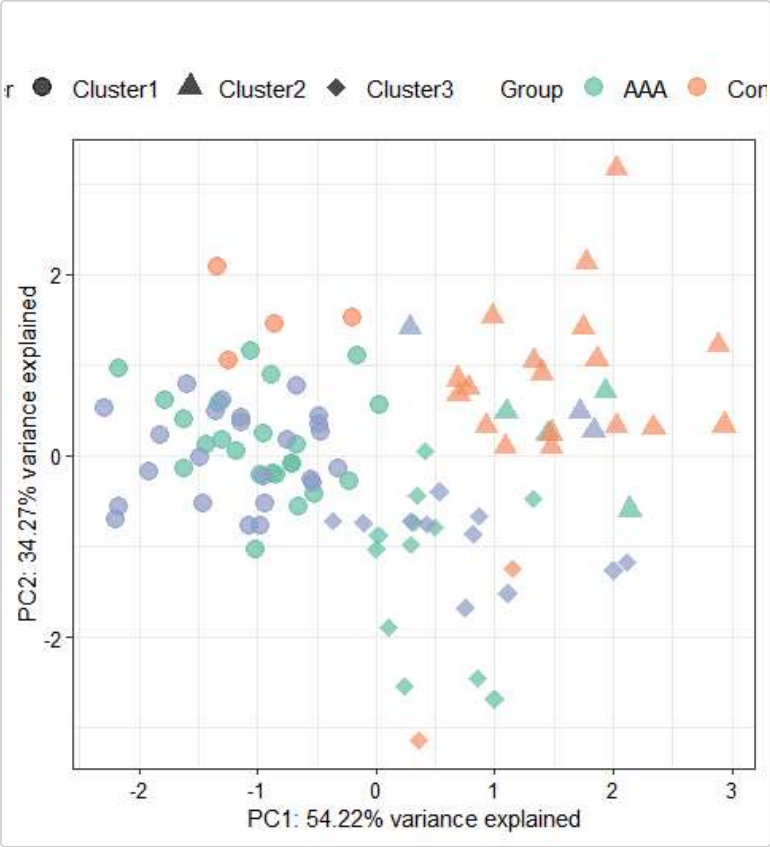



```
# Cellwidth parameter was set to 4 to avoid cropping the image on the right side.
```

Further analyses - a multigroup variant of analysis

Principal component analysis (PCA) and k means clustering work well with multiple groups, but the number of colors for points must be equal to the number of groups.

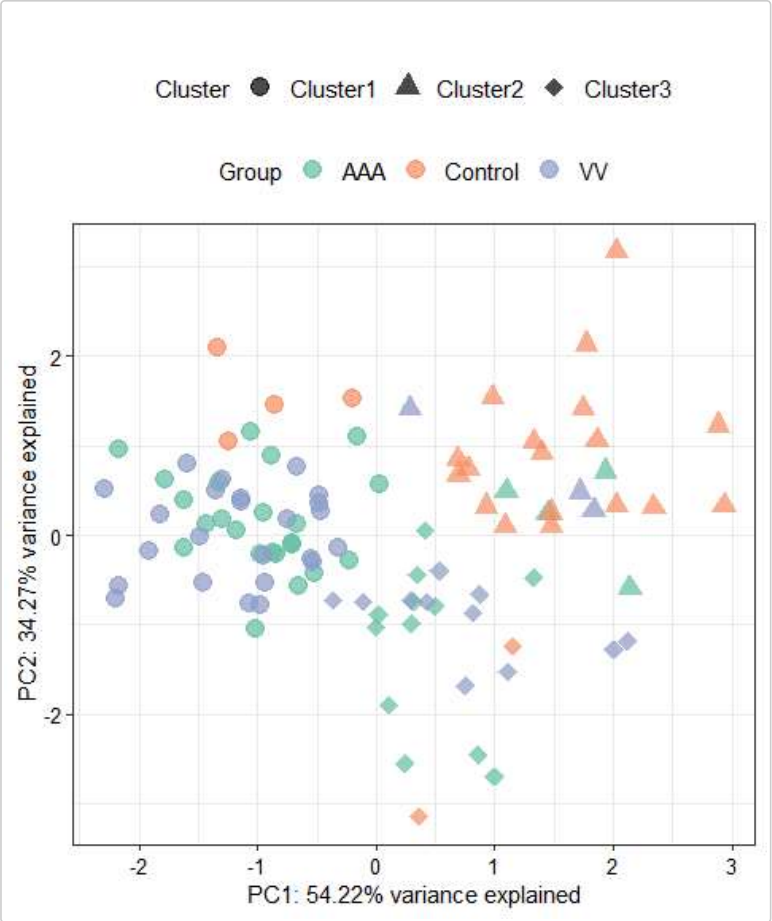
```
pca.kmeans <- pca_kmeans(data.dCtF.3groups,
                         sel.Gene = c("ANGPT1","VEGFB", "VEGFC"),
                         k.clust = 3,
                         clust.names = c("Cluster1", "Cluster2", "Cluster3"),
                         point.shape = c(19, 17, 18),
                         point.color = c("#66c2a5", "#fc8d62", "#8DA0CB"),
                         legend.position = "top")
```




```
# Access to the confusion matrix:
pca.kmeans[[2]]
#>
#>           Cluster1 Cluster2 Cluster3
#>   AAA           23        4       12
#>   Control        4       18        2
#>   VV            24        3       11
```

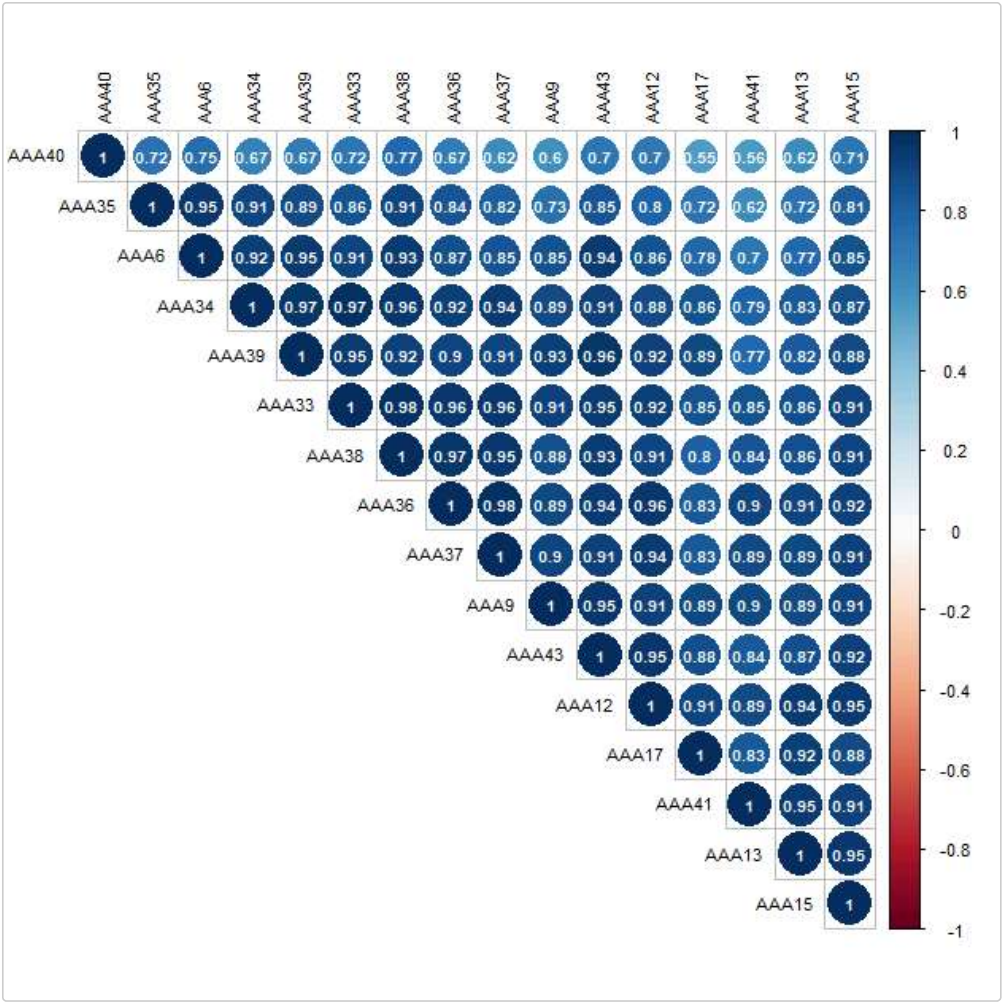
If the legend is to wide and is cropped, setting a vertical position of legend should solve this problem:

```
pca.kmeans[[1]] + theme(legend.box = "vertical")
```

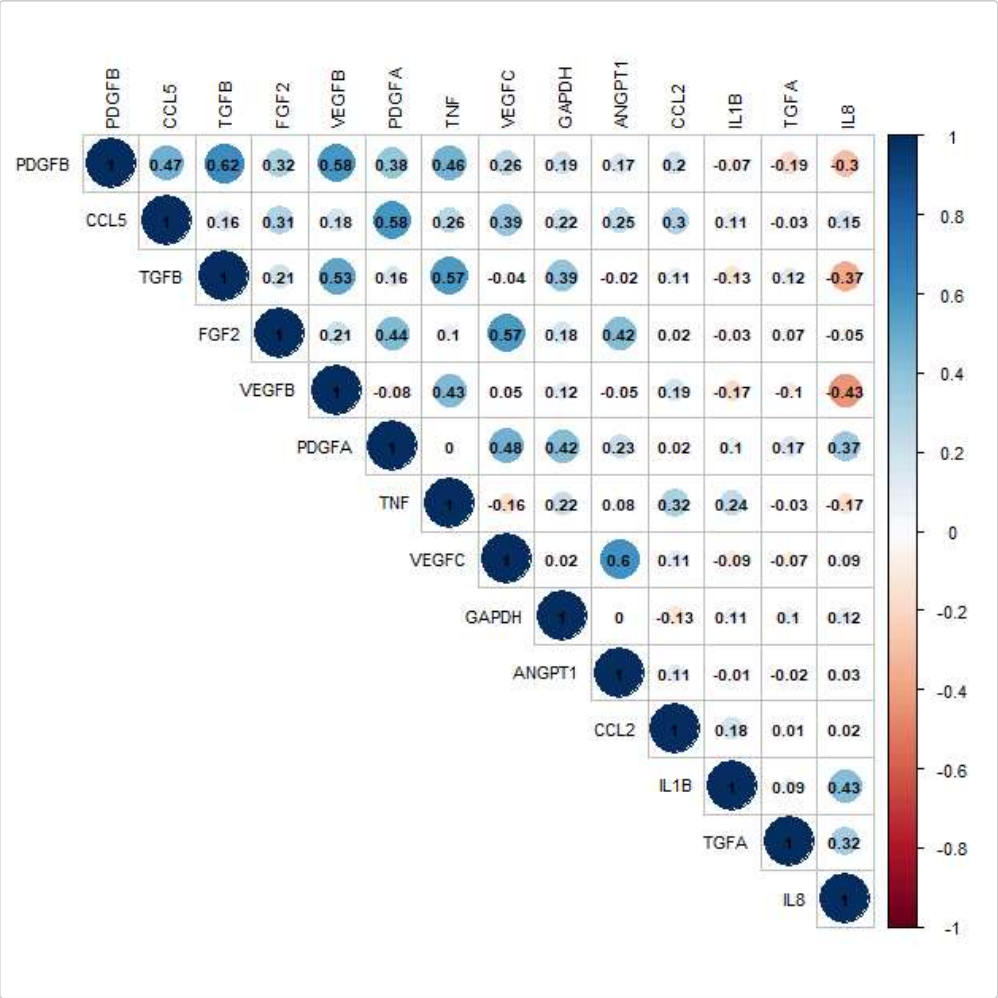


The functions performing correlation analysis of samples (`corr_sample()` function) and genes (`corr_gene()` function) can be used for multigroup data.

```
library(Hmisc)
library(corrplot)
# To make the plot more readable, only part of the data was used:
corr.samples <- corr_sample(data = data.dCtF.3groups[15:30, ],
                            method = "pearson",
                            order = "hclust",
                            size = 0.7,
                            p.adjust.method = "BH",
                            add.coef = "white")
```



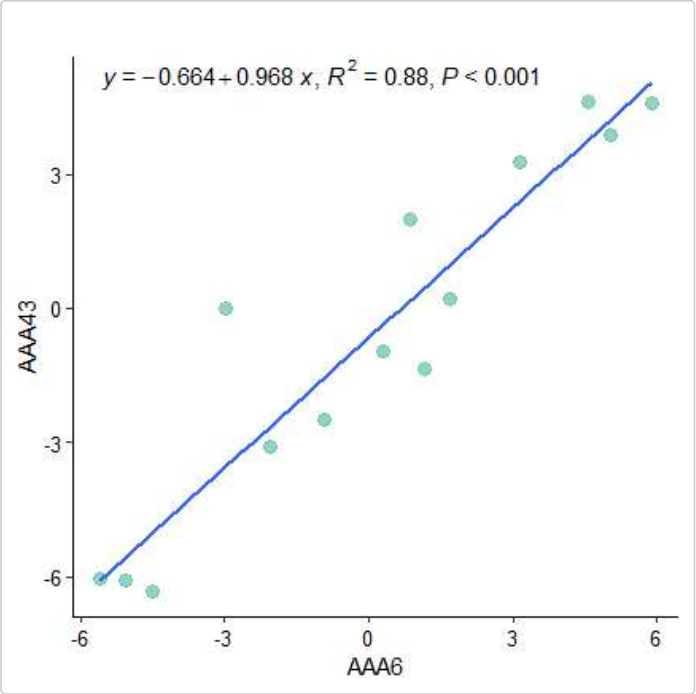

```
library(Hmisc)
library(corrplot)
corr.genes <- corr_gene(data = data.dCtF.3groups,
                        method = "spearman",
                        order = "FPC",
                        size = 0.7,
                        p.adjust.method = "BH")
```



Remember: A minimum of 5 samples/genes are required for correlation analysis.

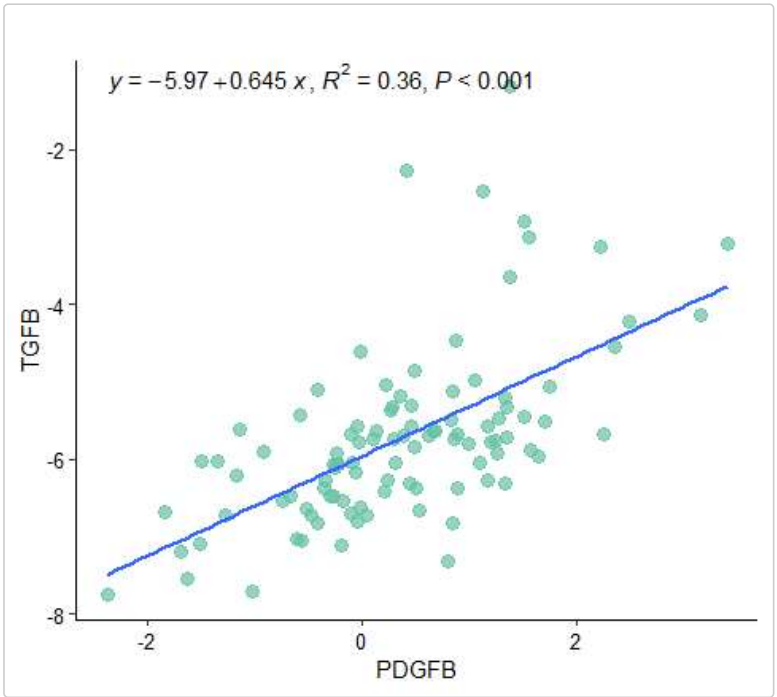
The function designed for simple linear regression analysis of samples (`single_pair_sample()`) can be used for multigroup data.

```
library(ggpmisc)
AAA6_AAA43 <- single_pair_sample(data = data.dCtF.3groups,
                                 x = "AAA6",
                                 y = "AAA43",
                                 point.size = 3,
                                 labels = TRUE,
                                 label = c("eq", "R2", "p"),
                                 label.position.x = 0.05)
```



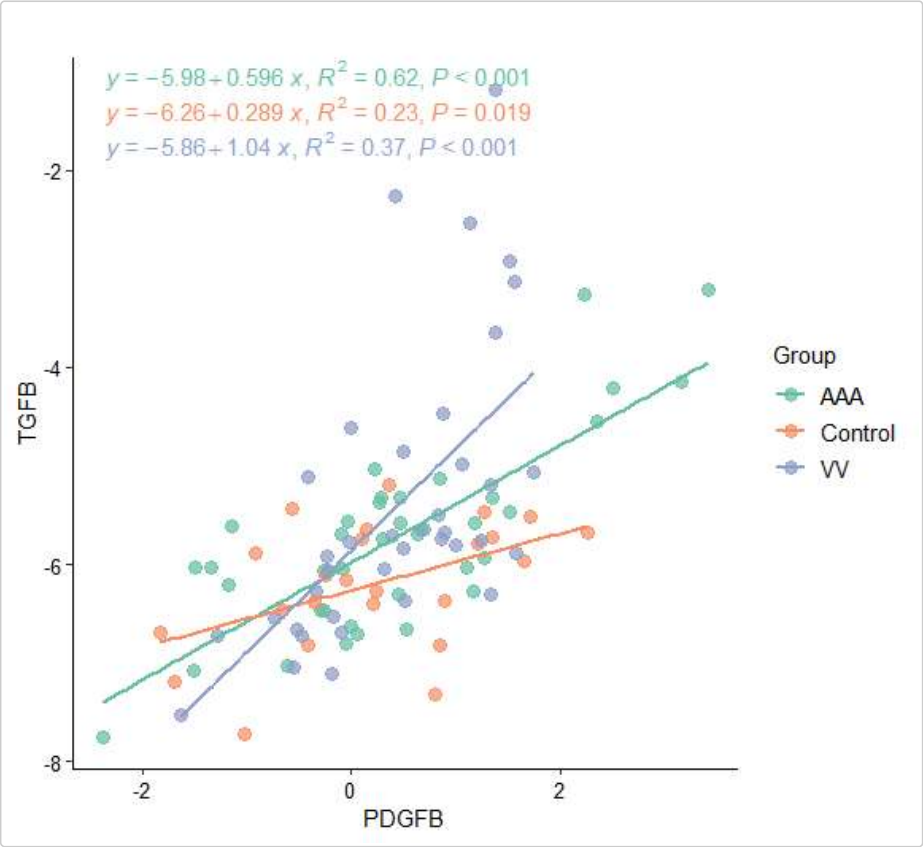
The function for simple linear regression analysis of genes (`single_pair_gene()`) can also be used for multigroup data; but if the analysis is performed with stratification by group (`by.group = FALSE`), a vector with colors must be provided in `colors` arguments, and the position of labels should be adjusted.

```
library(ggpmisc)
# Without stratification by groups:
PDGFB_TGFB <- single_pair_gene(data.dCtF.3groups,
                               x = "PDGFB",
                               y = "TGFB",
                               by.group = FALSE,
                               point.size = 3,
                               labels = TRUE,
                               label = c("eq", "R2", "p"),
                               label.position.x = c(0.05),
                               label.position.y = c(1,0.95))
```




```
library(ggpmisc)
# With stratification by groups:
PDGFB_TGFB <- single_pair_gene(data.dCtF.3groups,
                               x = "PDGFB",
                               y = "TGFB",
```

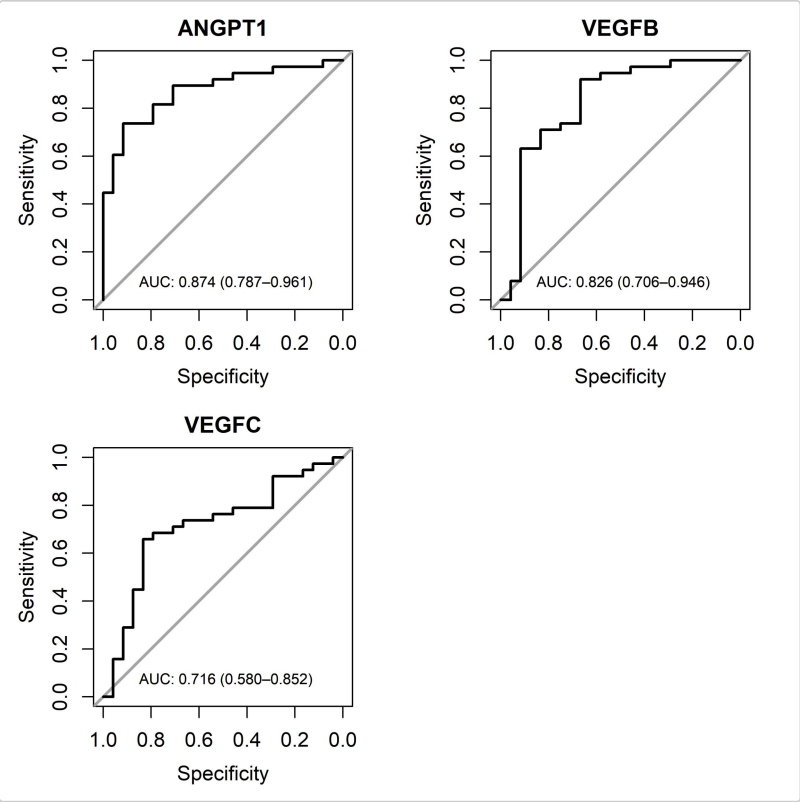
```
                    by.group = TRUE,
                    colors = c("#66c2a5", "#fc8d62", "#8DA0CB"), # Vector of colors
                    point.size = 3,
                    labels = TRUE,
                    label = c("eq", "R2", "p"),
                    label.position.x = c(0.05),
                    label.position.y = c(1,0.95,0.9)) # Labels position
```



The Receiver Operating Characteristic method works with only two groups, thus the `ROCh()` function also requires only two groups, specified in `groups` argument.

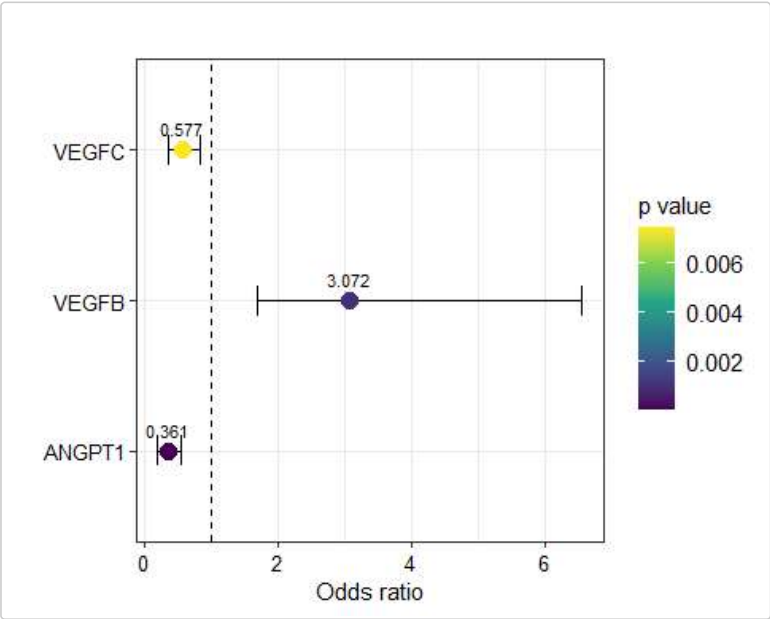
```
library(pROC)
# Remember to specify the numbers of rows (panels.row parameter) and columns (panels.col parameter) to
         be sufficient to arrange panels:
roc_parameters <- ROCh(data = data.dCtF.3groups,
                       sel.Gene = c("ANGPT1","VEGFB", "VEGFC"),
                       groups = c("Control","VV"),
                       panels.row = 2,
                       panels.col = 2)
roc_parameters
#>     Gene Threshold Specificity Sensitivity  Accuracy       ppv       npv
#> 1 ANGPT1    2.4900   0.9166667   0.7368421 0.8064516 0.9333333 0.6875000
#> 2  VEGFB   -1.1125   0.6666667   0.9210526 0.8225806 0.8139535 0.8421053
#> 3  VEGFC    3.4775   0.8333333   0.6578947 0.7258065 0.8620690 0.6060606
#>     youden       AUC
#> 1 1.653509 0.8739035
#> 2 1.587719 0.8256579
#> 3 1.491228 0.7160088
```

Remember: The created plot is not displayed on the graphic device, but should be saved as .tiff image (`save.to.txt = TRUE`) and can be opened directly from the file in the working directory.



Similarly to the Receiver Operating Characteristic method, logistic regression also works with only two groups, thus the `log_reg()` function requires also only two groups, specified in the `group.study` and `group.ref` arguments.

```
library(oddsratio)
# Remember to set the increment parameter.
log.reg.results <- log_reg(data = data.dCtF.3groups,
                           increment = 1,
                           sel.Gene = c("ANGPT1","VEGFB", "VEGFC"),
                           group.study = "VV",
                           group.ref = "Control")
```

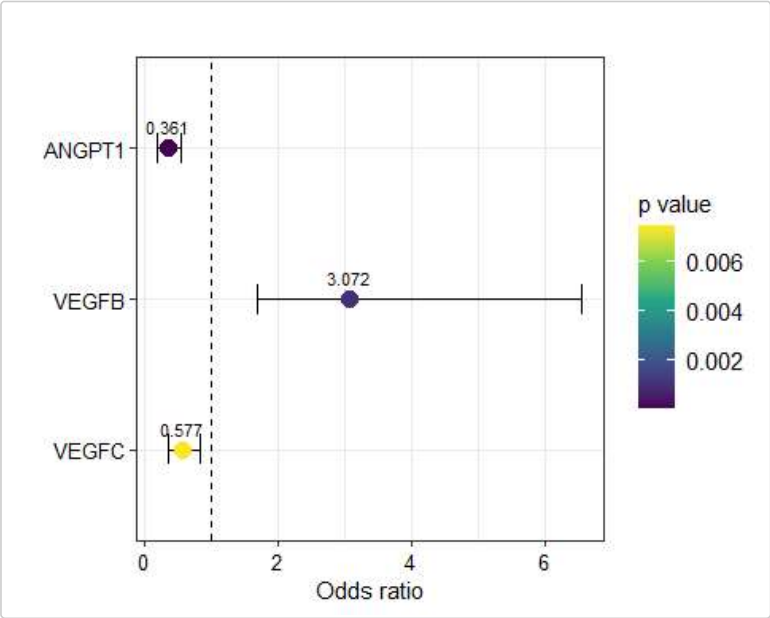



```
log.reg.results[[2]]
#>     Gene oddsratio CI_low CI_high Increment Intercept coeficient  p_intercept
#> 1 ANGPT1     0.361  0.207   0.559         1  3.616960 -1.0180064 4.656368e-05
#> 2  VEGFB     3.072  1.683   6.543         1  1.312207  1.1223027 1.611383e-03
#> 3  VEGFC     0.577  0.373   0.843         1  2.421392 -0.5490967 2.821054e-03
#>         p_coef   p_coef_adj
```

```
#> 1 4.830687e-05 0.0001449206
#> 2 1.048294e-03 0.0015724407
#> 3 7.466786e-03 0.0074667861
```

If genes should be displayed in alphabetical order, they can be simply sorted as follows:

```
log.reg.results.sorted <- log.reg.results[[1]] +
                          scale_y_discrete(limits = rev(sort(log.reg.results[[2]]$Gene)))
log.reg.results.sorted
```



Session info

```
sessionInfo()
#> R version 4.3.2 (2023-10-31 ucrt)
#> Platform: x86_64-w64-mingw32/x64 (64-bit)
#> Running under: Windows 10 x64 (build 19045)
#>
#> Matrix products: default
#>
#>
#> locale:
#> [1] LC_COLLATE=Polish_Poland.utf8  LC_CTYPE=Polish_Poland.utf8
#> [3] LC_MONETARY=Polish_Poland.utf8 LC_NUMERIC=C
#> [5] LC_TIME=Polish_Poland.utf8
#>
#> time zone: Europe/Warsaw
#> tzcode source: internal
#>
#> attached base packages:
#> [1] stats     graphics  grDevices utils     datasets  methods   base
#>
#> other attached packages:
#>  [1] oddsratio_2.0.1 pROC_1.18.5     ggpmisc_0.5.5   ggpp_0.5.6
#>  [5] corrplot_0.92   Hmisc_5.1-2     ggsignif_0.6.4  coin_1.4-3
#>  [9] survival_3.6-4  ctrlGene_1.0.1  pheatmap_1.0.12 lubridate_1.9.3
#> [13] forcats_1.0.0   stringr_1.5.0   dplyr_1.1.4     purrr_1.0.2
#> [17] readr_2.1.4     tidyr_1.3.1     tibble_3.2.1    ggplot2_3.5.0
#> [21] tidyverse_2.0.0 RQdeltaCT_1.3.3
#>
#> loaded via a namespace (and not attached):
#>  [1] tidyselect_1.2.1   viridisLite_0.4.2  farver_2.1.1       libcoin_1.0-10
#>  [5] fastmap_1.1.1      TH.data_1.1-2      GGally_2.2.1       digest_0.6.35
#>  [9] rpart_4.1.23       timechange_0.2.0   lifecycle_1.0.4    cluster_2.1.6
#> [13] magrittr_2.0.3     compiler_4.3.2     rlang_1.1.5        sass_0.4.9
#> [17] tools_4.3.2        utf8_1.2.3         yaml_2.3.8         data.table_1.15.4
```

```
#> [21] knitr_1.46          labeling_0.4.3     htmlwidgets_1.6.4  plyr_1.8.9
#> [25] RColorBrewer_1.1-3 multcomp_1.4-25    withr_3.0.0        foreign_0.8-86
#> [29] nnet_7.3-19         grid_4.3.2         stats4_4.3.2       fansi_1.0.5
#> [33] colorspace_2.1-0    scales_1.3.0       MASS_7.3-60        cli_3.6.3
#> [37] mvtnorm_1.2-4       rmarkdown_2.26     generics_0.1.3     rstudioapi_0.16.0
#> [41] tzdb_0.4.0          polynom_1.4-1      cachem_1.0.8       modeltools_0.2-23
#> [45] splines_4.3.2       parallel_4.3.2     matrixStats_1.0.0  base64enc_0.1-3
#> [49] vctrs_0.6.5         Matrix_1.6-4       sandwich_3.1-0     jsonlite_1.8.8
#> [53] SparseM_1.81        confintr_1.0.2     hms_1.1.3          Formula_1.2-5
#> [57] htmlTable_2.4.2     jquerylib_0.1.4    glue_1.6.2         ggstats_0.6.0
#> [61] codetools_0.2-20    stringi_1.7.12     gtable_0.3.5       munsell_0.5.1
#> [65] pillar_1.9.0        htmltools_0.5.8.1  quantreg_5.97      R6_2.5.1
#> [69] evaluate_0.23       lattice_0.21-9     highr_0.10         backports_1.4.1
#> [73] bslib_0.7.0         MatrixModels_0.5-3 Rcpp_1.0.12        gridExtra_2.3
#> [77] nlme_3.1-164        checkmate_2.3.1    mgcv_1.9-1         xfun_0.43
#> [81] zoo_1.8-12          pkgconfig_2.0.3
```