Supplement to: “RDML: an R Package for Working with RDML Format Data”

The **RDML** package was created to work with the Real-time PCR Data Markup Language (RDML) -- a structured and universal data standard for exchanging quantitative PCR (qPCR) data. RDML belongs to the family of eXtensible Markup Languages (XML). It contains fluorescence data and information about the qPCR experiment. A description and the RDML schema and the RDML format are available at http://rdml.org.

The structure of the **RDML** package mimics the RDML format and provides several [R6](http://cran.r-project.org/web/packages/R6/vignettes/Introduction.html) classes, which corresponds to [RDML v1.2](http://rdml.org/files.php?v=1.2) format types. All major manipulations with RDML data can be done by a class called **RDML** through its public methods:

* $new() -- creates new **RDML** object (empty or from specified RDML file)
* $AsTable() -- represents data contained in **RDML** object (except fluorescent data) as data.frame.
* $GetFData() -- gets fluorescent data.
* $SetFData() -- sets fluorescent data to **RDML** object.
* $AsDendrogram() -- represents structure of **RDML** object as dendrogram.

## Opening and observing RDML file

In this section we will use the built-in RDML example file lc96\_bACTXY.rdml. This file was obtained during the measurement of human DNA concentration by a *LightCycler 96* (Roche) and the *XY-Detect* kit (Syntol, Russia).

To open the lc96\_bACTXY.rdml file we have to create a new **RDML** object with its class initializer -- $new() and the file name as parameter filename.

PATH <- path.package("RDML")  
filename <- paste(PATH, "/extdata/", "lc96\_bACTXY.rdml", sep ="")  
lc96 <- RDML$new(filename = filename)

Next we can check structure of our new object -- lc96 by printing it.

lc96  
#> experiment: [ca1eb225-ecea-4793-9804-87bfbb45f81d]  
#> thermalCyclingConditions: [2f78ed33-724e-4a29-97e9-92296eb868e1]  
#> target: [30116ec1-44f6-4c9c-9c69-5d6f00226d4e, 69b0b5cd-591c-4012-a995-7a8b53861548, 7797a698-1b2d-4819-bf7d-1188f2c8ca7f, c16f36ee-8636-40d2-ae72-b00d3b2eb89d, bACT, X, Y, IPC]  
#> sample: [Sample 39, Sample 41, Sample 43, Sample 45, Sample 51, Sample 53, Sample 54, Sample 55, Sample 56, Sample 57, Sample 58]  
#> dye: [FAM, Hex, Texas Red, Cy5]  
#> documentation: []  
#> experimenter: []  
#> id: [Roche Diagnostics]  
#> dateUpdated: 2014-08-27T12:06:21  
#> dateMade: 2014-08-19T11:25:48

As a result we can see field names and after : :

* names of the R6 objects contained at this field after ~,
* contained values after :,
* names of list elements enclosed in [].

Fields names for all **RDML** package classes correspond to fields names of RDML types described at http://rdml.org/files.php?v=1.2.

For the base class **RDML** they are:

* dateMade
* dateUpdated
* id -- publisher and id to the RDML file.
* experimenter -- contact details of the experimenter.
* documentation -- these elements should be used if the same description applies to many samples, targets, or experiments.
* dye -- information about a dye.
* sample -- defined template solutions.
* target -- defined PCR reactions.
* thermalCyclingConditions -- cycling programs for PCR or to amplify cDNA.
* experiment

These fields can be divided by two parts:

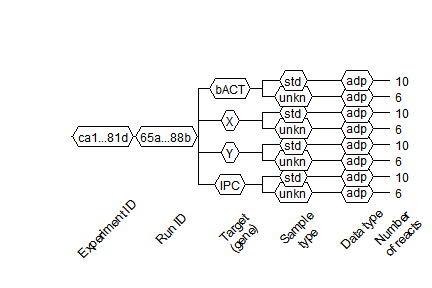
### Experiment field

Contains one or more experiments with fluorescence data. Fluorescence data are stored at the *data* level of an experiment. E.g., fluorescence data for reaction tube *45* and target *bACT* can be accessed with the following code:

fdata <-   
 lc96$  
 experiment$`ca1eb225-ecea-4793-9804-87bfbb45f81d`$  
 run$`65aeb1ec-b377-4ef6-b03f-92898d47488b`$  
 react$`45`$  
 data$bACT$  
 adp$fpoints #'adp' means amplification data points (qPCR)  
head(fdata)  
#> cyc tmp fluor  
#> [1,] 1 68.0054 0.0782385  
#> [2,] 2 68.0429 0.0753689  
#> [3,] 3 68.0451 0.0736838  
#> [4,] 4 68.0525 0.0723196  
#> [5,] 5 68.0537 0.0717019  
#> [6,] 6 68.0538 0.0714182

Structure of experiments can be visualized by plotting dendrogram.

lc96$AsDendrogram()



plot of chunk unnamed-chunk-5

In this dendrogram we can see that our file consists of one experiment and one run. Four targets, each with two sample types (*std* -- standard, *unkn* -- unknown), are part of the experiment. There is only qPCR data -- *adp* in this experiment. Ten reactions (tubes) for standard type (*std*) and six reaction for the unknown (*unkn*) type. The total number of reactions can be more than number of reactions on the plate because one tube can contain more than one target (e.g., multiplexing).

### Additional information fields

All fields other than **experiment**. This additional information can be referenced in other parts of the RDML file. E.g., to access sample added to react *39* and get its quantity we can use code like this:

ref <- lc96$  
 experiment$`ca1eb225-ecea-4793-9804-87bfbb45f81d`$  
 run$`65aeb1ec-b377-4ef6-b03f-92898d47488b`$  
 react$`39`$  
 sample$id  
sample <- lc96$sample[[ref]]  
sample$quantity$value  
#> [1] 25

## Copying RDML objects

**R6** objects are environments, that's why simple copying results in creating reference to existing object. Then modifying of copy leads to modification of original object. To create *real* copy of object we have to use method $clone(deep = TRUE) provided by **R6** class.

id1 <- idType$new("id\_1")  
id2 <- id1  
id3 <- id1$clone(deep = TRUE)  
id2$id <- "id\_2"  
id3$id <- "id\_3"  
cat(sprintf("Original object\t: %s ('id\_1' bacame 'id\_2')\nSimple copy\t\t: %s\nClone\t\t\t: %s\n",  
 id1$id, id2$id, id3$id))  
#> Original object : id\_2 ('id\_1' bacame 'id\_2')  
#> Simple copy : id\_2  
#> Clone : id\_3

From example above we can see that modification of id2 led to modification of original object id1 but modification of cloned object id3 didn't.

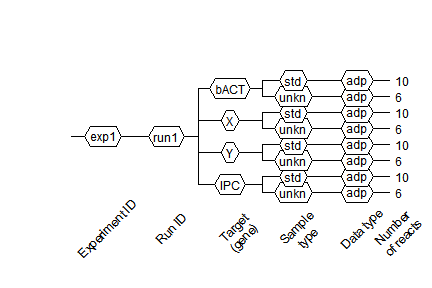
## Modifying RDML objects

To modify content of **RDML** objects we can use fields as setters. These setters provide type safe modification by input validation. In addition, setting lists of objects generates names of list elements.

# Create 'real' copy of object  
experiment <- lc96$experiment$`ca1eb225-ecea-4793-9804-87bfbb45f81d`$clone(deep = TRUE)  
# Try to set 'id' with wrong input type.  
# Correct type 'idType' can be seen at error message.  
tryCatch(experiment$id <- "exp1",  
 error = function(e) print(e))  
#> <assertError: id is not a 'idType' or length > 1>  
  
# Set 'id' with correct input type - 'idType'  
experiment$id <- idType$new("exp1")  
  
# Similar operations for 'run'  
run <- experiment$run$`65aeb1ec-b377-4ef6-b03f-92898d47488b`$clone(deep = TRUE)  
run$id <- idType$new("run1")  
  
# Replace original elements with modified  
experiment$run <- list(run)  
lc96$experiment <- list(experiment)

And we can see our modification with $AsDendrogram() method.

lc96$AsDendrogram()



plot of chunk unnamed-chunk-9

## AsTable() method

To get information about all fluorescence data in RDML file (type of added sample, used target, starting quantity etc.) as data.frame we can use $AsTable() method. By default, it provides such information as:

* fdata.name -- aggregated name for current fluorescence data. Default pattern is position\_sample\_sample.type\_target (e.g., *D03\_Sample 39\_std\_bACT*). This pattern can be modified by name.pattern argument.
* exp.id -- experiment id (e.g., *exp1*).
* run.id -- run id (e.g., *run1*).
* react.id -- react (tube) id (e.g., *39*).
* position -- react (tube) position (e.g., *D03*).
* sample -- name of the added sample (e.g., *Sample 39*).
* target -- detection target (e.g., *bACT*).
* target.dyeId -- detection dye (e.g., *FAM*).
* sample,type -- type of the added sample (e.g., *std*).
* adp -- TRUE if contains qPCR data.
* mdp -- TRUE if contains melting data.

To add custom columns for output *data.frame* we should pass it as named method argument with generating expression. Values of default columns can be used at custom name pattern and new columns referring to their names. Next example shows how to use $AsTable() method with a custom name pattern and additional column.

tab <- lc96$AsTable(  
 # Custom name pattern 'position~sample~sample.type~target~dye'  
 name.pattern = paste(  
 react$Position(run$pcrFormat),  
 react$sample$id,  
 private$.sample[[react$sample$id]]$type$value,  
 data$tar$id,  
 target[[data$tar$id]]$dyeId$id,  
 sep = "~"),  
 # Custom column 'quantity' - starting quantity of added sample   
 quantity = sample[[react$sample$id]]$quantity$value  
)  
# Remove row names for compact printing  
rownames(tab) <- NULL  
head(tab)  
#> fdata.name exp.id run.id react.id position sample  
#> 1 D03~Sample 39~std~bACT~FAM exp1 run1 39 D03 Sample 39  
#> 2 D03~Sample 39~std~X~Hex exp1 run1 39 D03 Sample 39  
#> 3 D03~Sample 39~std~Y~Texas Red exp1 run1 39 D03 Sample 39  
#> 4 D03~Sample 39~std~IPC~Cy5 exp1 run1 39 D03 Sample 39  
#> 5 D04~Sample 39~std~bACT~FAM exp1 run1 40 D04 Sample 39  
#> 6 D04~Sample 39~std~X~Hex exp1 run1 40 D04 Sample 39  
#> target target.dyeId sample.type adp mdp quantity  
#> 1 bACT FAM std TRUE FALSE 25  
#> 2 X Hex std TRUE FALSE 25  
#> 3 Y Texas Red std TRUE FALSE 25  
#> 4 IPC Cy5 std TRUE FALSE 25  
#> 5 bACT FAM std TRUE FALSE 25  
#> 6 X Hex std TRUE FALSE 25

Also, the generated data.frame is used as a query in $GetFData() and $SetFData() methods (see further sections).

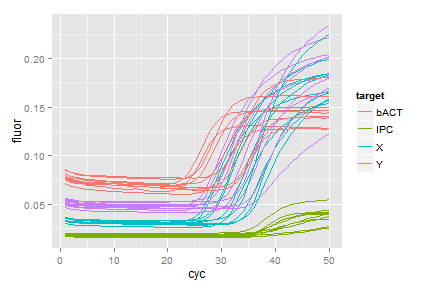
## Getting fluorescence data

We can get the fluorescence data two ways:

* direct access to data as it was described at *Experiment field* subsection
* using special method $GetFData()

Advantage of $GetFData() is that it can combine fluorescence data from whole plate to one data.frame. Major argument of this function is request, which defines fluorescence data to be got. This request is output from $AsTable() method and can be filtered with ease by the **dplyr** filter() function. Also limits of cycles, output data.frame format and data type (fdata.type = 'adp' for qPCR, fdata.type = 'mdp' for melting data) can by specified (see examples below).

library(dplyr)  
library(ggplot2)  
  
# Prepare request to get only 'std' type samples  
filtered.tab <- filter(tab,  
 sample.type == "std")  
  
fdata <- lc96$GetFData(filtered.tab,  
 # long table format for usage with ggplot2  
 long.table = TRUE)  
ggplot(fdata, aes(cyc, fluor)) +  
 geom\_line(aes(group = fdata.name,  
 color = target))



plot of chunk unnamed-chunk-11

Our curves are not background subtrackted ass visible in the plot. To do this we use the CPP() function from the [chipPCR](https://github.com/michbur/chipPCR) package.

library(chipPCR)  
tab <- lc96$AsTable(  
 # Custom name pattern 'position~sample~sample.type~target~run.id'  
 name.pattern = paste(  
 react$Position(run$pcrFormat),  
 react$sample$id,  
 private$.sample[[react$sample$id]]$type$value,  
 data$tar$id,  
 run$id$id, # run id added to names  
 sep = "~"))  
# Get all fluorescence data  
fdata <- lc96$GetFData(tab,  
 # We don't need long table format for CPP()  
 long.table = FALSE)  
  
fdata.cpp <- cbind(cyc = fdata[, 1],  
 apply(fdata[, -1], 2,  
 function(x) CPP(fdata[, 1],  
 x)$y))

Now we have preprocessed data, which we will add to our object and use during next section.

## Setting fluorescence data

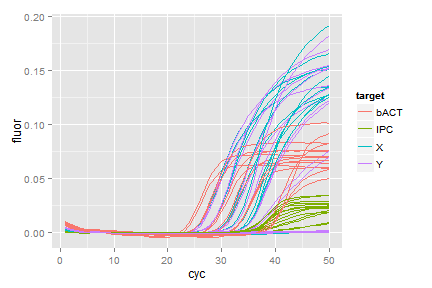
To set fluorescence data to **RDML** object we can use $SetFData() method. It takes three arguments:

* fdata -- fluorescence data in long.table = FALSE format
* request -- output from AsTable() function, which is used as path to set data;
* fdata.type -- fdata.type = 'adp' for qPCR, fdata.type = 'mdp' for melting data.

Next we will set preprocessed fluorescence data to the new run -- *run1\_cpp*. Such subelements of RDML as *experiment*, *run*, *react* and *data* that do not exist at **RDML** object create by SetFData automaticaly (read more at *Creating* ***RDML*** *from table* section).

Note that colnames in fdata and fdata.name in request have to be the same!

tab$run.id <- "run.cpp"  
# Set fluorescence data from previous section  
lc96$SetFData(fdata.cpp,  
 tab)  
  
# View setted data  
fdata <- lc96$GetFData(tab,  
 long.table = TRUE)  
ggplot(fdata, aes(cyc, fluor)) +  
 geom\_line(aes(group = fdata.name,  
 color = target))

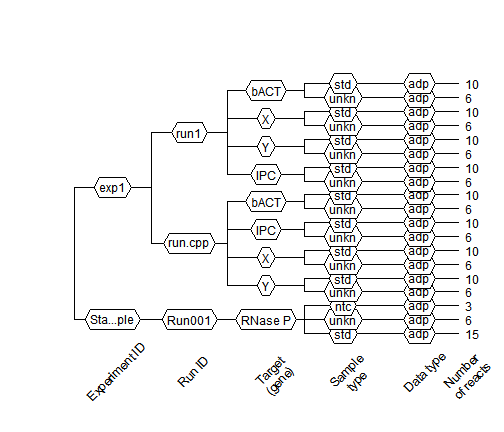


plot of chunk unnamed-chunk-13

## Merging RDML objects

Merging **RDML** objects can be done by MergeRDMLs() function. It takes list of **RDML** objects and returns one **RDML** object.

# Load another built in RDML file  
stepone <- RDML$new(paste0(path.package("RDML"),  
 "/extdata/", "stepone\_std.rdml"))  
# Merge it with our 'lc96' object  
merged <- MergeRDMLs(list(lc96, stepone))  
# View structure of new object  
merged$AsDendrogram()



plot of chunk unnamed-chunk-14

## Saving RDML object as RDML file

To save **RDML** object as RDML file v1.2 we can use $AsXML() method where file.name argument is name of new RDML file. Without file.name function returns XML tree.

XML package is pretty slow and file generating can take much time

lc96$AsXML("lc96.rdml")

You can use [RDML-ninja](http://sourceforge.net/projects/qpcr-ninja/) to validate a created file.

## Creating custom functions

**R6** classes allow add methods to existing classes. This can be done using the $set() method. Suppose that we decided add method to preprocess all fluorescence data and calculate Cq:

RDML$set("public", "CalcCq",  
 function() {  
 library(chipPCR)  
 fdata <- self$GetFData(  
 self$AsTable())  
 fdata <- cbind(cyc = fdata[, 1],  
 apply(fdata[, -1],  
 2,  
 function(x)  
 # Data preprocessing  
 CPP(fdata[, 1],  
 x)$y)  
 )  
   
 apply(fdata[, -1], 2,  
 function(x) {  
 tryCatch(  
 # Calculate Cq  
 th.cyc(fdata[, 1], x,  
 auto = TRUE)@.Data[1],  
 error = function(e) NA)  
 })  
 }  
)  
#> Error in RDML$set("public", "CalcCq", function() {: Can't add CalcCq because it already present in RDML generator.  
  
# Create new object with our advanced class  
stepone <- RDML$new(paste0(path.package("RDML"),  
 "/extdata/", "stepone\_std.rdml"))

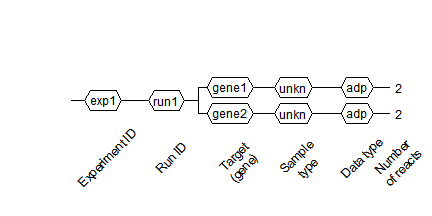
And then apply our new method:

stepone$CalcCq()  
#> A01\_NTC\_RNase P\_ntc\_RNase P A02\_NTC\_RNase P\_ntc\_RNase P   
#> 13.801661 12.964268   
#> A03\_NTC\_RNase P\_ntc\_RNase P A04\_pop1\_RNase P\_unkn\_RNase P   
#> 14.399910 13.529380   
#> A05\_pop1\_RNase P\_unkn\_RNase P A06\_pop1\_RNase P\_unkn\_RNase P   
#> 14.609048 14.160886   
#> A07\_pop2\_RNase P\_unkn\_RNase P A08\_pop2\_RNase P\_unkn\_RNase P   
#> 11.839277 10.947218   
#> B05\_pop2\_RNase P\_unkn\_RNase P B06\_STD\_RNase P\_10000.0\_std\_RNase P   
#> 13.167509 13.415532   
#> B07\_STD\_RNase P\_10000.0\_std\_RNase P B08\_STD\_RNase P\_10000.0\_std\_RNase P   
#> 13.363262 14.082731   
#> C01\_STD\_RNase P\_5000.0\_std\_RNase P C02\_STD\_RNase P\_5000.0\_std\_RNase P   
#> 13.039183 14.012804   
#> C03\_STD\_RNase P\_5000.0\_std\_RNase P C04\_STD\_RNase P\_2500.0\_std\_RNase P   
#> 9.790724 12.981337   
#> D01\_STD\_RNase P\_2500.0\_std\_RNase P D02\_STD\_RNase P\_2500.0\_std\_RNase P   
#> 15.666928 13.132126   
#> D03\_STD\_RNase P\_1250.0\_std\_RNase P D04\_STD\_RNase P\_1250.0\_std\_RNase P   
#> 13.905808 14.974769   
#> D05\_STD\_RNase P\_1250.0\_std\_RNase P D06\_STD\_RNase P\_625.0\_std\_RNase P   
#> 13.880906 12.850727   
#> D07\_STD\_RNase P\_625.0\_std\_RNase P D08\_STD\_RNase P\_625.0\_std\_RNase P   
#> 16.915642 13.706484

## Creating RDML from table

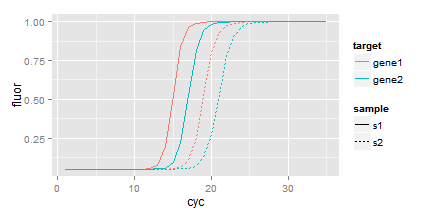
**RDML** objects can be generated not only from files but from user data contained in data.frames. To do this you have to create empty **RDML** object, create data.frame, which describes data and set data by $SetFData() method. Minimal needed information (samples, targets, dyes) will be created from data description.

### Create simulated data with AmpSim() from chipPCR package  
# Cq for data to be generated  
Cqs <- c(15, 17, 19, 21)  
# PCR si,ulation will be 35 cycles  
fdata <- data.frame(cyc = 1:35)  
for(Cq in Cqs) {  
 fdata <- cbind(fdata,  
 AmpSim(cyc = 1:35, Cq = Cq)[, 2])  
}  
# Set names for fluorescence curves  
colnames(fdata)[2:5] <- c("c1", "c2", "c3", "c4")  
  
# Create minimal description  
descr <- data.frame(  
 fdata.name = c("c1", "c2", "c3", "c4"),  
 exp.id = c("exp1", "exp1", "exp1", "exp1"),  
 run.id = c("run1", "run1", "run1", "run1"),  
 react.id = c(1, 1, 2, 2),  
 sample = c("s1", "s1", "s2", "s2"),  
 target = c("gene1", "gene2", "gene1", "gene2"),  
 target.dyeId = c("FAM", "ROX", "FAM", "ROX"),  
 stringsAsFactors = FALSE  
)  
  
# Create empty RDML object  
sim <- RDML$new()  
# Add fluorescence data  
sim$SetFData(fdata, descr)  
  
# Observe object  
sim$AsDendrogram()



plot of chunk unnamed-chunk-17

fdata <- sim$GetFData(sim$AsTable(),  
 long.table = TRUE)  
ggplot(fdata, aes(cyc, fluor)) +  
 geom\_line(aes(group = fdata.name,  
 color = target,  
 linetype = sample))



## Functional style

To provide functional programming style, which is more convenient in **R**, the **RDML** class methods have function wrappers:

* obj$AsTable(...) -- AsTable(obj, ...)
* obj$SetFData(...) -- SetFData(obj, ...)
* obj$GetFData(...) -- GetFData(obj, ...)
* obj$AsDendrogram(...) -- AsDendrogram(obj, ...)