Extending flowQ: how to implement QA processes

F. Hahne B. Ellis

May 12, 2008

Abstract

flowQ provides infrastructure to generate interactive quality reports based on a unified HTML output. The software is readily extendable via modules, where each module comprises a single QA process. This Vignette is a brief tutorial how to create your own QA process modules.

1 Basic idea of flowQ's QA reports

In flowCore, flow cytometry data is organized in *flowFrames* and *flowSets*. Usually, a *flowSet* comprises one experiment or one staining panel of one particular experiemnt. The initial step of all data analysis is typically a quality assessment (QA) check. Depending on the design of the experiment, the measurement channels and the biological question, there are various levels on which QA makes sense and also various different parameters that have to be checked.

In flowQ we tried to implement a framework that allows to create consise QA reports for one or several *flowSets* and that is readily extendable using self-defined modules. The general design of a flowQ QA process is:

- aggregator: a qualitative or quantitative value that indicates the outcome of a QA process or of one of its subprocesses for one single *flowFrame* in the set.
- summary graph: a plot summarizing the result of the QA process for the whole *flowSet*.
- frame graphs plots visualizing the outcome of a QA process or of one of its subprocesses for a single *flowFrame* (optional).

A single QA process may contain various subprocesses, for instance looking at each measurement channel in a *flowFrame* separately, and each of these subprocesses may have its own aggregator and/or graphs. However, one unified aggregator indicating the overall outcome of the QA process is mandatory.

Abstractions for each of these building blocks are avaible as classes and for each class there are constructors which will do the dirty work behind the scenes. All that needs to be provided by the user-defined QA functions are file paths to the respective plots and lists of aggregators indicating the outcome (based on cutoff values that have been computed before. For each of these classes, there are `writeLine` methods, which create the appropriate HTML output. The user doesn't have to care about this step, a fully formated report will be generated when calling the `writeQAReport` function.

2 Aggregators

There are several subclasses of aggregators, all inheriting from the virtual parent class *qaAggregator*, which defines a single slot, `passed`. This slot is the basic indicator whether the *flowFrame* has passes the particular quality check. More fine-grained output can be archived by the following types of sub-classes (see their documentation for details):

- *binaryAggregator*: the most basic aggregator, indicating "passed" or "not passed" by color coding.

- *discreteAggregator*: allows for three different states: "passed", "not passes" and "warn", also coded by colors. Not that "warn" will set the `passed` slot to `FALSE`.

- *factorAggregator*: multiple outcome states. The factor levels are plotted along with color coding for the overall outcome ("passed" or "not passed").

- *stringAggregator*: arbitrary character string describing the outcome. Font color indicates the overall outcome.

- *numericAggregator*: a numerical value describing the outcome. Currently, the value is plotted as a character string, but this might change in the future. Font color indicates the overall outcome.

- *rangeAggregator*: a numerical value within a certain range describing the outcome. A horizontal barplot is produced with color indicating the overall outcome.

Aggregator objects can be created using either `new` or the constructor functions. E.g., the following code creates instances of each of the six aggregator types:

```
> binaryAggregator()

Binary quality score passing the requirements

> discreteAggregator(2)

Discrete quality score not passing the requirements with state warn
```

```
> factorAggregator(factor("a", levels = letters[1:3]))

Factorized quality score passing the requirements of value=a

> stringAggregator("test", passed = FALSE)

Textual quality score not passing the requirements of value=test

> numericAggregator(20)

Numeric quality score passing the requirements of value=20

> rangeAggregator(10, 0, 100)

Range quality score passing the requirements of value=10
```

A special class *aggregatorList* exists that holds multiple aggregators, not necessarily of the same type, and this is used for QA processes with several subprocesses. The constructor takes an arbitrary number of *qaAggregator* objects, or a list of such objects. This class mainly exists for method dispatch.

```
> aggregatorList(bin = binaryAggregator(FALSE), disc = discreteAggregator(1))

List of 2 aggregators
```

3 Storing images as *qaGraph*s

While aggregators indicate the general outcome of a QA process, or, at most, a single quantitative value, the amount of information they can provide is very limited. flowQ's design allows to include additional diagnostic plots, both on the level of the whole *flowSet* and for each *flowFrame* individually. Smaller bitmap versions of the plots are used for the overview page, and each image is clickable, opening a bigger vectorized version of the plot that is better suited for detailed inspection. To take the burden of file conversion away from the user, the class *qaGraph* was implemented, which stores single images. The class constructor takes two mandatory arguments: `fileName` which is a valid path to an image file (either bitmap or vectorized), and `imageDir`, which is a file path to the output directory where the image files are to be stored. If you are planning to place the final QA report on a web server, you should make sure, that this path is accessable. The safet solution is to chose a directory below the root directory of the QA report, e.g., `qaReport/images` if the root directory is /codeqaReport.

You can control the final width of the bitmap version of the image through the optional `width` argument, and empty *qaGraph* objects can be created by setting `empty=TRUE`. During object instantiation, the file type is detected automatically and the image file will be converted, resized and copied if necessary.

```
> tmp <- tempdir()
> fn <- file.path(tmp, "test.jpg")
> jpeg(file = fn)
> plot(1:3)
> dev.off()
```

```
null device
          1

> idir <- file.path(tmp, "images")
> g <- qaGraph(fn, imageDir = idir)
> g

QA process image information

> qaGraph(imageDir = idir, empty = TRUE)

QA process image information
```

For the special case of QA processes with multiple subprocesses (e.g., individual plots for each channel), there is a class *qaGraphList* and an associated constructor, which will take a character vector of multiple file names. This class mainly exists for method dispatch and to facilitate batch processing of multiple image files.

4 Information for a single frame: class *qaProcessFrame*

All the information of a QA process for a single frame has to be bundled in objects of class *qaProcessFrame*. Again, a constructor facilitates instantiating these objects; the mandatory arguments of the constructor are:

- `frameID`: a unique identifier for the *flowFrame*. Most of the time, this will be the `sampleName` of the frame in the *flowSet*. The frame will be identified by this symbol in all of the following steps and you should make sure that you use unique values, otherwise the downstream functions will not work.
- `summaryAggregator`: an object inheriting from class *qaAggregator* indicating the overall outcome of the process for this frame.

Further optional arguments are:

- `summaryGraph`: an object of class *qaGraph* providing a graphical summary of the QA process for this frame.
- `frameAggregators`: an object of class *aggregatorList*. Each aggregator in the list indicates the outcome of one single subprocess for this frame, e.g., for every individual measurement channel.
- `frameGraphs`: an object of class *qaGraphList*. Each qaGraph in the list is a graphical overview over the outcome of one single subprocess for this frame. Note that the length of both `frameAggregators` and `frameGraphs` have to be the same if you want to use them. Assuming that you don't want to include images for one of the subprocesses, you have to provide empty *qaGraph* objects (see above). It is not possible to omit aggregators for subprocesses, because they are used to link to the repsective images.

- `details`: a list of additional information that you want to keep attached to the *qaProcessFrame*. For example, this can be the values of a quality score that was computed in order to decide whether the QA process has passed the requirements. Such information can be useful to update aggregators later without reproducing the images (e.g., when a cutoff value has been changed).

5 The whole QA process: class *qaProcess*

Now that we have all the information for the single frames together, we can proceed and bundle things up in a unified object of class *qaProcess*. Again, the constructor has a couple of mandatory arguments:

- `id`: a unique identifier for this QA process. This will be used to identify the process in all downstream functions, which will not work unless it really is unique (assuming that you want to combine multiple QA processes in one single report).
- `type`: A character scalar describing the type of the QA process. This might become useful for functions that operate on objects of class *qaProcess* in a type-specific way (e.g., updating agregators).
- `frameProcesses`: a list of *qaProcessFrame* objects. You have to make sure that the identifier for each *qaProcessFrame* is unique and that the length of the list is equal to the length of the *flowSet*.

Further optional arguments are:

- `name`: The name of the process that is used as caption in the output.
- `summaryGraph`: An object of class *qaGraph* summarizing the outcome of the QA process for the whole frame. Although this is mandatory, we strongly recommend including such a plot, as it provides a good initial overview.

The output of you own QA process function should always be an object of class *qaProcess*, which can be used in the downstream functions to produce the quality assessment report. Most of the time, the function would have a structure similar to the following:

1. iterate over frames to create the *qaProcessFrame* object, possibly with an additional level of iteration for each subprocess (e.g. each channel). Each iteration involves creation of at least one *qaGraph* and *qaAggregator* object.
2. create an *qaProcessFrame* object summarizing the process for the whole set.
3. bundle things up in a *qaProcess* object.