flowCore: data structures package for flow cytometry data

N. Le Meur F. Hahne B. Ellis P. Haaland

March 17, 2017

Abstract

Background The recent application of modern automation technologies to staining and collecting flow cytometry (FCM) samples has led to many new challenges in data management and analysis. We limit our attention here to the associated problems in the analysis of the massive amounts of FCM data now being collected. From our viewpoint, see two related but substantially different problems arising. On the one hand, there is the problem of adapting existing software to apply standard methods to the increased volume of data. The second problem, which we intend to address here, is the absence of any research platform which bioinformaticians, computer scientists, and statisticians can use to develop novel methods that address both the volume and multidimensionality of the mounting tide of data. In our opinion, such a platform should be Open Source, be focused on visualization, support rapid prototyping, have a large existing base of users, and have demonstrated suitability for development of new methods. We believe that the Open Source statistical software R in conjunction with the Bioconductor Project fills all of these requirements. Consequently we have developed a Bioconductor package that we call flowCore. The flowCore package is not intended to be a complete analysis package for FCM data, rather, we see it as providing a clear object model and a collection of standard tools that enable R as an informatics research platform for flow cytometry. One of the important issues that we have addressed in the flowCore package is that of using a standardized representation that will insure compatibility with existing technologies for data analysis and will support collaboration and interoperability of new methods as they are developed. In order to do this, we have followed the current standardized descriptions of FCM data analysis as being developed under NIH Grant xxxx [n]. We believe that researchers will find flowCore to be a solid foundation for future development of new methods to attack the many interesting open research questions in FCM data analysis.

Methods We propose a variety different data structures. We have implemented the classes and methods in the Bioconductor package flowCore. We illustrate their use with X case studies.

Results We hope that those proposed data structures will be the base for the development of many tools for the analysis of high throughput flow cytometry.

keywords Flow cytometry, high throughput, software, standard

1 Introduction

Traditionally, flow cytometry has been a tube-based technique limited to small-scale laboratory and clinical studies. High throughput methods for flow cytometry have recently been developed for drug

discovery and advanced research methods (**?**). As an example, the flow cytometry high content screening (FC-HCS) can process up to a thousand samples daily at a single workstation, and the results have been equivalent or superior to traditional manual multi-parameter staining and analysis techniques.

The amount of information generated by high throughput technologies such as FC-HCS need to be transformed into executive summaries (which are brief enough) for creative studies by a human researcher (**?**). Standardization is critical when developing new high throughput technologies and their associated information services (**???**). Standardization efforts have been made in clinical cell analysis by flow cytometry (**?**), however data interpretation has not been standardized for even low throughput FCM. It is one of the most difficult and time consuming aspects of the entire analytical process as well as a primary source of variation in clinical tests, and investigators have traditionally relied on intuition rather than standardized statistical inference (**????**). In the development of standards in high throughput FCM, few progress has been done in term of Open Source software. In this article we propose R data structures to handle flow cytometry data through the main steps of preprocessing: compensation, transformation, filtering.

The aim is to merge both prada and rflowcyt (**?**) into one core package which is compliant with the data exchange standards that are currently developed in the community (**?**).

Visualization as well as quality control will than be part of the utility packages that depend on the data structures defined in the flowCore package.

2 Representing Flow Cytometry Data

flowCore's primary task is the representation and basic manipulation of flow cytometry (or similar) data. This is accomplished through a data model very similar to that adopted by other Bioconductor packages using the `expressionSet` and `AnnotatedDataFrame` structures familiar to most Bioconductor users.

2.1 The *flowFrame* Class

The basic unit of manipulation in flowCore is the *flowFrame*, which corresponds roughly with a single "FCS" file exported from the flow cytometer's acquisition software. At the moment we support FCS file versions 2.0 through 3.0, and we expect to support FCS4/ACS1 as soon as the specification has been ratified.

2.1.1 Data elements

The primary elements of the `flowFrame` are the `exprs` and `parameters` slots, which contain the event-level information and column metadata respectively. The event information, stored as a single matrix, is accessed and manipulated via the `exprs()` and `exprs<-` methods, allowing `flowFrames` to be stitched together if necessary (for example, if the same tube has been collected in two acquisition files for memory reasons).

The `parameters` slot is an *AnnotatedDataFrame* that contains information derived from an FCS file's "$P¡n¿" keywords, which describe the detector and stain information. The entire list is available via the `parameter()` method, but more commonly this information is accessed through the `names`, `featureNames` and `colnames` methods. The `names` function returns a concatenated version of

`names` and `featureNames` using a format similar to the one employed by most flow cytometry analysis software. The `colnames` method returns the detector names, often named for the fluorochrome detected, while the `featureNames` methods returns the description field of the parameters, which will typically be an identifier for the antibody.

The `keyword` method allows access to the raw FCS keywords, which are a mix of standard entries such as "SAMPLE ID," vendor specific keywords and user-defined keywords that add more information about an experiment. In the case of plate-based experiments, there are also one or more keywords that identify the specific well on the plate.

Most vendor software also include some sort of unique identifier for the file itself. The specialized methods `identifier` attempts to locate an appropriate globally unique identifier that can be used to uniquely identify a frame. Failing that, this method will return the original file name offering some assurance that this frame is at least unique to a particular session.

2.1.2 Reading a `flowFrame`

FCS files are read into the R environment via the `read.FCS` function using the standard connection interface—allowing for the possibility of accessing FCS files hosted on a remote resource as well as those that have been compressed or even retrieved as a blob from a database interface. FCS files (version 2.0 and 3.0) and LMD (List Mode Data) extensions are currently supported.

There are also several immediate processing options available in this function, the most important of which is the `transformation` parameter, which can either "linearize" (the default) or "linearize-with-PnG-scaling" or "scale" our data. To see how this works, first we will examine an FCS file without any transformation at all:

```
file.name <- system.file("extdata","0877408774.B08", package="flowCore")
x <- read.FCS(file.name, transformation=FALSE)
summary(x)
```

```
##         FSC-H  SSC-H FL1-H  FL2-H FL3-H   FL1-A  FL4-H  Time
## Min.       85   11.0   0.0    0.0   0.0    0.00    0.0   1.0
## 1st Qu.   385  141.0 233.0  277.0  90.0    0.00  210.0 122.0
## Median    441  189.0 545.5  346.0 193.0   26.00  279.0 288.0
## Mean      492  277.9 439.1  366.2 179.7   34.08  323.5 294.8
## 3rd Qu.   518  270.0 610.0  437.0 264.0   51.00  390.0 457.5
## Max.     1023 1023.0 912.0 1023.0 900.0 1023.00 1022.0 626.0
```

As we can see, in this case the values from each parameter seem to run from 0 to 1023 ($2^{10} - 1$). However, inspection of the "exponentiation" keyword ($P¡n¿E) reveals that some of the parameters (3 and 4) have been stored in the format of the form $a \times 10^{x/R}$ where a is given by the first element of the string.

```
keyword(x,c("$P1E", "$P2E", "$P3E", "$P4E"))
```

```
## $`$P1E`
## [1] "0,0"
##
```

```
## $`$P2E`
## [1] "0,0"
##
## $`$P3E`
## [1] "4,0"
##
## $`$P4E`
## [1] "4,0"
```

The default "linearize" transformation option will convert these to, effectively, have a "$P¡n¿E" of "0,0":

```
summary(read.FCS(file.name))
```

```
##         FSC-H  SSC-H    FL1-H   FL2-H    FL3-H   FL1-A    FL4-H  Time
## Min.       85   11.0    1.000    1.00    1.000    0.00    1.000   1.0
## 1st Qu.   385  141.0    8.131   12.08    2.247    0.00    6.612 122.0
## Median    441  189.0  135.200   22.47    5.674   26.00   12.300 288.0
## Mean      492  277.9  157.800  106.00    8.465   34.08  140.400 294.8
## 3rd Qu.   518  270.0  241.400   50.94   10.750   51.00   33.380 457.5
## Max.     1023 1023.0 3652.000 9910.00 3278.000 1023.00 9822.000 626.0
```

The "linearize-with-PnG-scaling" option will perform the previous transformation and it will also apply a "division by gain" to pamameters stored on linear scale with specified gain. The gain is specified in the $PnG keywords. This option has been introduced as part of Gating-ML 2.0 compliance.

Finally, the "scale" option will both linearize values as well as ensure that output values are contained in $[0, 1]$, which is the proposed method of data storage for the ACS1.0/FCS4.0 specification:

```
summary(read.FCS(file.name,transformation="scale"))
```

```
##           FSC-H   SSC-H     FL1-H    FL2-H     FL3-H   FL1-A     FL4-H
## Min.    0.08309 0.01075 0.0000000 0.000000 0.0000000 0.00000 0.0000000
## 1st Qu. 0.37630 0.13780 0.0007132 0.001108 0.0001247 0.00000 0.0005612
## Median  0.43110 0.18480 0.0134200 0.002147 0.0004675 0.02542 0.0011300
## Mean    0.48090 0.27170 0.0156800 0.010500 0.0007466 0.03331 0.0139400
## 3rd Qu. 0.50640 0.26390 0.0240500 0.004994 0.0009747 0.04985 0.0032380
## Max.    1.00000 1.00000 0.3651000 0.991000 0.3277000 1.00000 0.9822000
##              Time
## Min.    0.0009775
## 1st Qu. 0.1193000
## Median  0.2815000
## Mean    0.2881000
## 3rd Qu. 0.4472000
## Max.    0.6119000
```

Another parameter of interest is the `alter.names` parameter, which will convert the parameter names into more "R friendly" equivalents, usually by replacing "-" with ".":

```
read.FCS(file.name,alter.names=TRUE)

## flowFrame object '0877408774.B08'
## with 10000 cells and 8 observables:
##      name               desc range minRange maxRange
## $P1 FSC.H              FSC-H  1024        0     1023
## $P2 SSC.H              SSC-H  1024        0     1023
## $P3 FL1.H               <NA>  1024        1    10000
## $P4 FL2.H               <NA>  1024        1    10000
## $P5 FL3.H               <NA>  1024        1    10000
## $P6 FL1.A               <NA>  1024        0     1023
## $P7 FL4.H               <NA>  1024        1    10000
## $P8  Time Time (51.20 sec.)  1024        0     1023
## 164 keywords are stored in the 'description' slot
```

When only a particular subset of parameters is desired the `column.pattern` parameter allows for the specification of a regular expression and only parameters that match the regular expression will be included in the frame. For example, to include on the Height parameters:

```
x <- read.FCS(file.name, column.pattern="-H")
x

## flowFrame object '0877408774.B08'
## with 10000 cells and 6 observables:
##      name  desc range minRange maxRange
## $P1 FSC-H FSC-H  1024        0     1023
## $P2 SSC-H SSC-H  1024        0     1023
## $P3 FL1-H  <NA>  1024        1    10000
## $P4 FL2-H  <NA>  1024        1    10000
## $P5 FL3-H  <NA>  1024        1    10000
## $P7 FL4-H  <NA>  1024        1    10000
## 160 keywords are stored in the 'description' slot
```

Note that `column.pattern` is applied after `alter.names` if it is used.

Finally, only a sample of lines can be read in case you need a quick overview of a large series of files.

```
lines <- sample(100:500, 50)
y <- read.FCS(file.name, which.lines = lines)
y

## flowFrame object '0877408774.B08'
## with 50 cells and 8 observables:
```

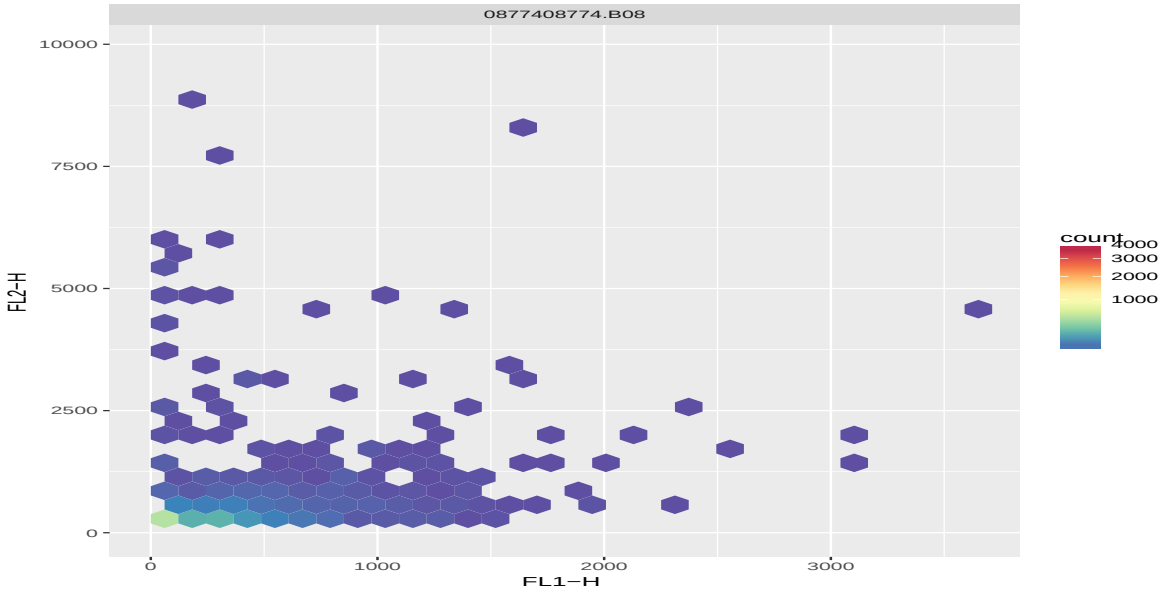
```
##       name                desc range minRange maxRange
## $P1 FSC-H               FSC-H  1024        0     1023
## $P2 SSC-H               SSC-H  1024        0     1023
## $P3 FL1-H                <NA>  1024        1    10000
## $P4 FL2-H                <NA>  1024        1    10000
## $P5 FL3-H                <NA>  1024        1    10000
## $P6 FL1-A                <NA>  1024        0     1023
## $P7 FL4-H                <NA>  1024        1    10000
## $P8  Time Time (51.20 sec.)  1024        0     1023
## 164 keywords are stored in the 'description' slot
```

2.1.3 Visualizing a *flowFrame*

Much of the more sophisticated visualization of *flowFrame* and *flowSet* objects, including an interface to the ggplot2 graphics system is implemented by the ggcyto package, also included as part of Bioconductor. Here, we will only introduce the `autoplot` function. See vignettes of ggcyto for more examples of how to visualize flow data.

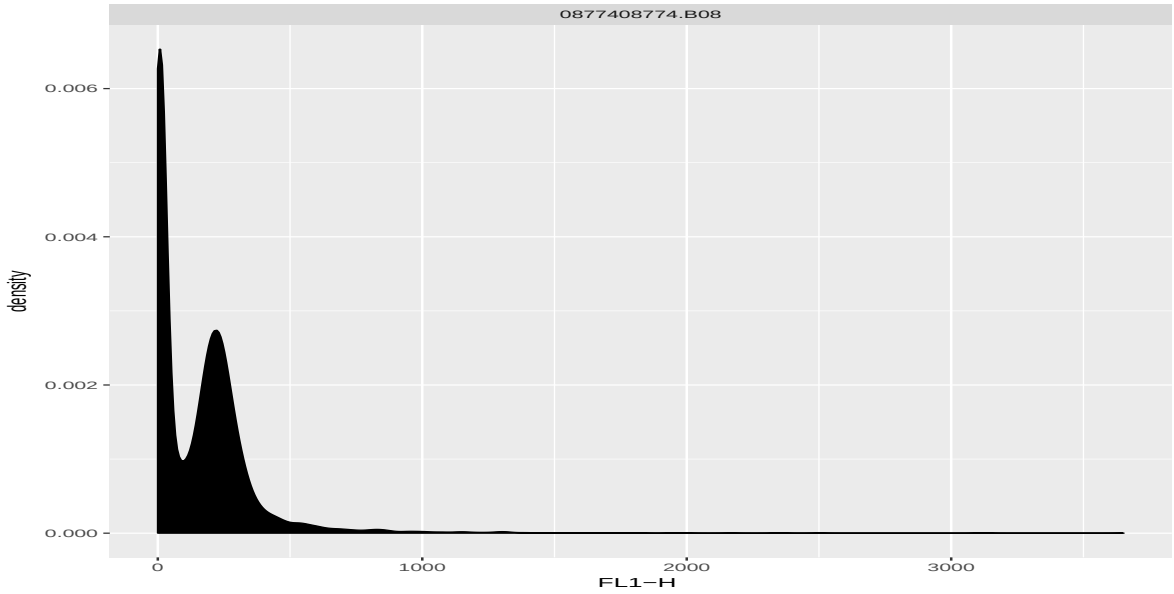
To create a bivariate density plot:

```
library(ggcyto)

## Loading required package:  ggplot2
## Loading required package:  ncdfFlow
## Loading required package:  RcppArmadillo
## Loading required package:  BH
## Loading required package:  flowWorkspace

autoplot(x, "FL1-H", "FL2-H")
```



To get a univariate densityplot:

```
autoplot(x, "FL1-H")
```



2.2 The *flowSet* Class

Most experiments consist of several *flowFrame* objects, which are organized using a *flowSet* object. This class provides a mechanism for efficiently hosting the *flowFrame* objects with minimal copying, reducing memory requirements, as well as ensuring that experimental metadata stays properly to the appropriate *flowFrame* objects.

2.2.1 Creating a *flowSet*

To facilitate the creation of *flowSet* objects from a variety of sources, we provide a means to coerce *list* and *environment* objects to a *flowSet* object using the usual coercion mechanisms. For example, if we have a directory containing FCS files we can read in a list of those files and create a *flowSet* out of them:

```
frames <- lapply(dir(system.file("extdata", "compdata", "data",
                                 package="flowCore"), full.names=TRUE),
                 read.FCS)
as(frames, "flowSet")

## A flowSet with 5 experiments.
##
##   column names:
##   FSC-H SSC-H FL1-H FL2-H FL3-H FL1-A FL4-H
```

Note that the original list is unnamed and that the resulting sample names are not particularly meaningful. If the list is named, the list constructed is much more meaningful. One such approach is to employ the `keyword` method for *flowFrame* objects to extract the "SAMPLE ID" keyword from each frame:

```
names(frames) <- sapply(frames, keyword, "SAMPLE ID")
fs <- as(frames, "flowSet")
fs

## A flowSet with 5 experiments.
##
##   column names:
##   FSC-H SSC-H FL1-H FL2-H FL3-H FL1-A FL4-H
```

2.2.2 Working with experimental metadata

Like most Bioconductor organizational classes, the *flowSet* has an associated *AnnotatedDataFrame* that provides metadata not contained within the *flowFrame* objects themselves. This data frame is accessed and modified via the usual `phenoData` and `phenoData<-` methods. You can also generally treat the phenotypic data as a normal data frame to add new descriptive columns. For example, we might want to track the original filename of the frames from above in the phenotypic data for easier access:

```
phenoData(fs)$Filename <- fsApply(fs,keyword, "$FIL")
pData(phenoData(fs))

##      name   Filename
## NA     NA 060909.001
## fitc fitc 060909.002
## pe     pe 060909.003
## apc   apc 060909.004
## 7AAD 7AAD 060909.005
```

Note that we have used the *flowSet*-specific iterator, `fsApply`, which acts much like `sapply` or `lapply`. Additionally, we should also note that the `phenoData` data frame **must** have row names that correspond to the original names used to create the *flowSet*.

2.2.3 Bringing it all together: `read.flowSet`

Much of the functionality described above has been packaged into the `read.flowSet` convenience function. In it's simplest incarnation, this function takes a `path`, that defaults to the current working directory, and an optional `pattern` argument that allows only a subset of files contained within the working directory to be selected. For example, to read a *flowSet* of the files read in by `frame` above:

```
read.flowSet(path = system.file("extdata", "compdata", "data",
             package="flowCore"))
```

```
## A flowSet with 5 experiments.
##
##   column names:
##   FSC-H SSC-H FL1-H FL2-H FL3-H FL1-A FL4-H
```

`read.flowSet` will pass on additional arguments meant for the underlying `read.FCS` function, such as `alter.names` and `column.pattern`, but also supports several other interesting arguments for conducting initial processing:

files An alternative to the `pattern` argument, you may also supply a vector of filenames to read.

name.keyword Like the example in the previous section, you may specify a particular keyword to use in place of the filename when creating the *flowSet*.

phenoData If this is an *AnnotatedDataFrame*, then this will be used in place of the data frame that is ordinarily created. Additionally, the row names of this object will be taken to be the filenames of the FCS files in the directory specified by `path`. This argument may also be a named list made up of a combination of `character` and `function` objects that specify a keyword to extract from the FCS file or a function to apply to each frame that will return a result.

To recreate the *flowSet* that we created by hand from the last section we can use `read.flowSets` advanced functionality:

```
fs <- read.flowSet(path=system.file("extdata", "compdata", "data",
                   package="flowCore"), name.keyword="SAMPLE ID",
                   phenoData=list(name="SAMPLE ID", Filename="$FIL"))
fs

## A flowSet with 5 experiments.
##
## An object of class 'AnnotatedDataFrame'
##   rowNames: NA fitc ... 7AAD (5 total)
##   varLabels: name Filename
##   varMetadata: labelDescription
##
##   column names:
##   FSC-H SSC-H FL1-H FL2-H FL3-H FL1-A FL4-H

pData(phenoData(fs))

##       name   Filename
## NA      NA 060909.001
## fitc fitc 060909.002
## pe      pe 060909.003
## apc   apc 060909.004
## 7AAD 7AAD 060909.005
```

2.2.4 Manipulating a *flowSet*

You can extract a *flowFrame* from a *flowSet* object in the usual way using the `[[` or `$` extraction operators. On the other hand using the `[` extraction operator returns a new *flowSet* by **copying** the environment. However, simply assigning the *flowFrame* to a new variable will **not** copy the contained frames.

The primary iterator method for a *flowSet* is the `fsApply` method, which works more-or-less like `sapply` or `lapply` with two extra options. The first argument, `simplify`, which defaults to `TRUE`, instructs `fsApply` to attempt to simplify it's results much in the same way as `sapply`. The primary difference is that if all of the return values of the iterator are *flowFrame* objects, `fsApply` will create a new *flowSet* object to hold them. The second argument, `use.exprs`, which defaults to `FALSE` instructs `fsApply` to pass the expression matrix of each frame rather than the *flowFrame* object itself. This allows functions to operate directly on the intensity information without first having to extract it.

As an aid to this sort of operation we also introduce the `each_row` and `each_col` convenience functions that take the place of `apply` in the `fsApply` call. For example, if we wanted the median value of each parameter of each *flowFrame* we might write:

```
fsApply(fs, each_col, median)
```

```
##       FSC-H SSC-H      FL1-H      FL2-H      FL3-H FL1-A      FL4-H
## NA      423   128   4.104698   4.531584   3.651741     0   7.233942
## fitc    436   128 930.572041 228.757320  33.376247   217   8.278826
## pe      438   120  10.181517 791.475544 114.444190     0   9.305720
## apc     441   129   4.371445   4.869675   4.782858     0 358.663762
## 7AAD    429   133   5.002865  14.989296  63.209339     0  20.908000
```

which is equivalent to the less readable

```
fsApply(fs,function(x) apply(x, 2, median), use.exprs=TRUE)
```

```
##       FSC-H SSC-H      FL1-H      FL2-H      FL3-H FL1-A      FL4-H
## NA      423   128   4.104698   4.531584   3.651741     0   7.233942
## fitc    436   128 930.572041 228.757320  33.376247   217   8.278826
## pe      438   120  10.181517 791.475544 114.444190     0   9.305720
## apc     441   129   4.371445   4.869675   4.782858     0 358.663762
## 7AAD    429   133   5.002865  14.989296  63.209339     0  20.908000
```

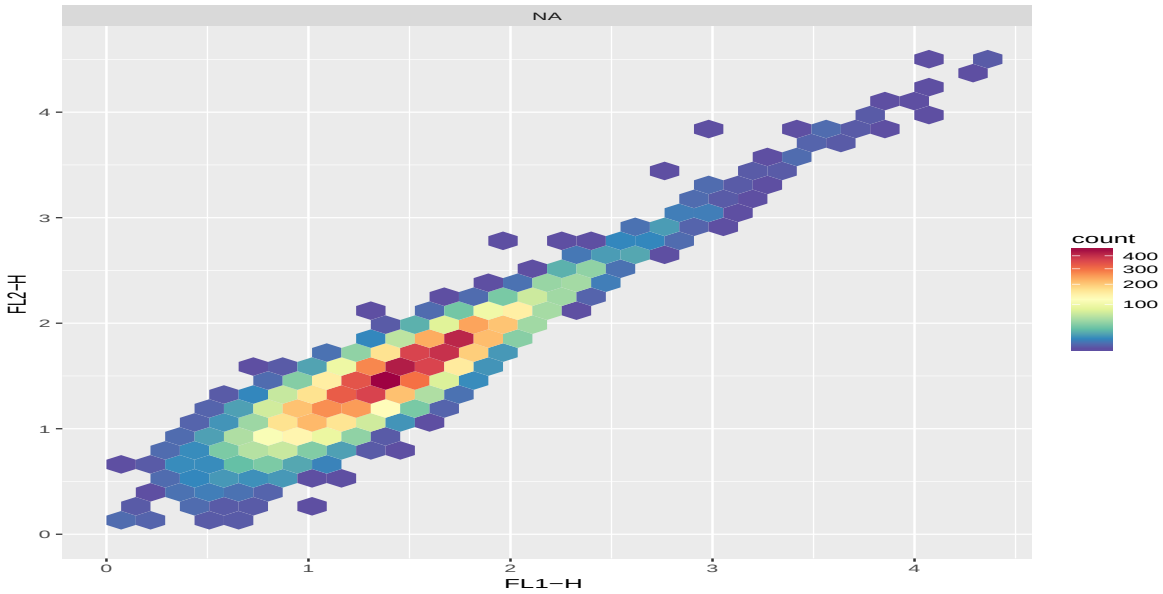
In this case, the `use.exprs` argument is not required in the first case because `each_col` and `each_row` are methods and have been defined to work on *flowFrame* objects by first extracting the intensity data.

3 Transformation

flowCore features two methods of transforming parameters within a *flowFrame*: inline and out-of-line. The inline method, discussed in the next section has been developed primarily to support filtering features and is strictly more limited than the out-of-line transformation method, which uses

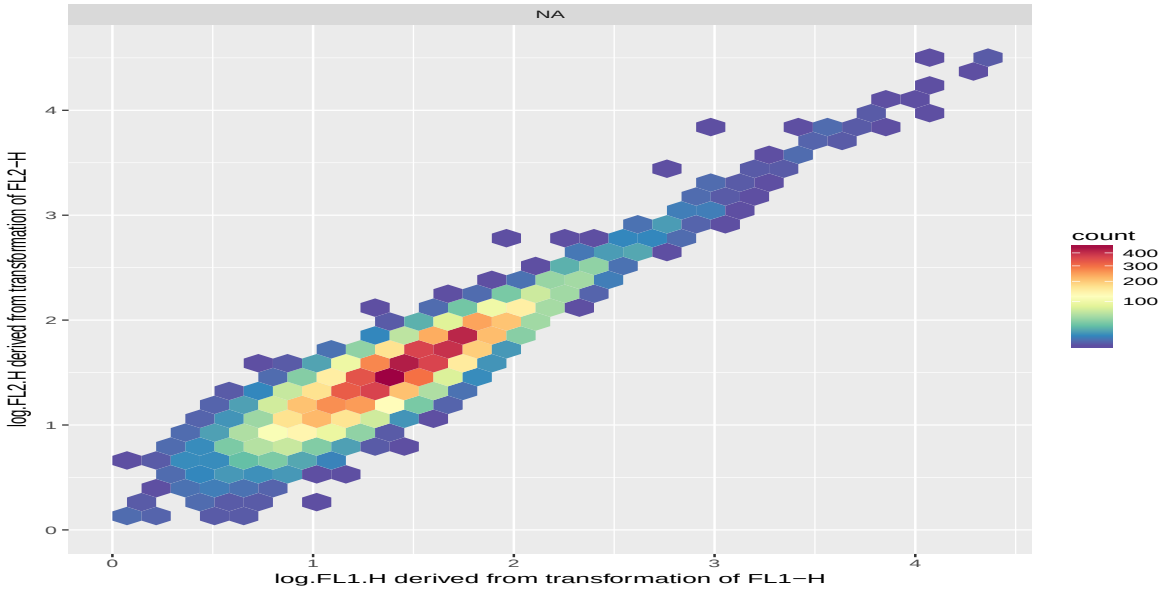
R's `transform` function to accomplish the filtering. Like the normal `transform` function, the *flowFrame*is considered to be a data frame with columns named for parameters of the FCS file. For example, if we wished to plot our first *flowFrame*'s first two fluorescence parameters on the log scale we might write:

```
autoplot(transform(fs[[1]]
                   , `FL1-H`=log(`FL1-H`)
                   , `FL2-H`=log(`FL2-H`)
                   )
         , "FL1-H","FL2-H")
```



Like the usual `transform` function, we can also create new parameters based on the old parameters, without destroying the old

```
autoplot(transform(fs[[1]]
                   , log.FL1.H=log(`FL1-H`)
                   , log.FL2.H=log(`FL2-H`)
                   )
         , "log.FL1.H", "log.FL2.H")
```



3.1 Standard Transforms

Though any function can be used as a transform in both the out-of-line and inline transformation techniques, flowCore provides a number of parameterized transform generators that correspond to the transforms commonly found in flow cytometry and defined in the Transformation Markup Language (Transformation-ML, see `http://www.ficcs.org/` and **?** for more details). Briefly, the predefined transforms are:

truncateTransform $y = \begin{cases} a & x < a \\ x & x \geq a \end{cases}$

scaleTransform $f(x) = \frac{x-a}{b-a}$

linearTransform $f(x) = a + bx$

quadraticTransform $f(x) = ax^2 + bx + c$

lnTransform $f(x) = \log(x)\,\frac{r}{d}$

logTransform $f(x) = \log_b(x)\,\frac{r}{d}$

biexponentialTransform $f^{-1}(x) = ae^{bx} - ce^{dx} + f$

logicleTransform A special form of the biexponential transform with parameters selected by the data.

arcsinhTransform $f(x) = asinh(a + bx) + c$

To use a standard transform, first we create a transform function via the constructors supplied by flowCore:

```
aTrans <- truncateTransform("truncate at 1", a=1)
aTrans

## transform object 'truncate at 1'
```

which we can then use on the parameter of interest in the usual way

```
transform(fs,`FL1-H`=aTrans(`FL1-H`))

## A flowSet with 5 experiments.
##
## An object of class 'AnnotatedDataFrame'
##   rowNames: NA fitc ... 7AAD (5 total)
##   varLabels: name Filename
##   varMetadata: labelDescription
##
##   column names:
##   FSC-H SSC-H FL1-H FL2-H FL3-H FL1-A FL4-H
```

However this form of transform call is not intended to used in the programmatic context because locally defined transform function (e.g. 'aTrans') may not be always visible to the non-standard evaluation environment. .e.g

```
f1 <- function(fs,...){
  transform(fs, ...)[,'FL1-H']
}

f2 <- function(fs){
  aTrans <- truncateTransform("truncate at 1", a=1)
  f1(fs, `FL1-H` = aTrans(`FL1-H`))
}
res <- try(f2(fs), silent = TRUE)
res

## A flowSet with 5 experiments.
##
## An object of class 'AnnotatedDataFrame'
##   rowNames: NA fitc ... 7AAD (5 total)
##   varLabels: name Filename
##   varMetadata: labelDescription
##
##   column names:
##   FL1-H
```

So this form of usage of 'transform' method is only useful for the interactive exploratary. we highly recommend the usage of `transformList` instead for the more robust and reproducible code.

```
myTrans <- transformList('FL1-H', aTrans)
transform(fs, myTrans)

## A flowSet with 5 experiments.
##
## An object of class 'AnnotatedDataFrame'
##   rowNames: NA fitc ... 7AAD (5 total)
##   varLabels: name Filename
##   varMetadata: labelDescription
##
##   column names:
##   FSC-H SSC-H FL1-H FL2-H FL3-H FL1-A FL4-H
```

4 Gating

The most common task in the analysis of flow cytometry data is some form of filtering operation, also known as gating, either to obtain summary statistics about the number of events that meet a certain criteria or to perform further analysis on a subset of the data. Most filtering operations are a composition of one or more common filtering operations. The definition of gates in flowCore follows the Gating Markup Language Candidate Recommendation **?**, thus any flowCore gating strategy can be reproduced by any other software that also adheres to the standard and *vice versa*.

4.1 Standard gates and filters

Like transformations, flowCore includes a number of built-in common flow cytometry gates. The simplest of these gates are the geometric gates, which correspond to those typically found in interactive flow cytometry software:

rectangleGate Describes a cubic shape in one or more dimensions–a rectangle in one dimension is simply an interval gate.

polygonGate Describes an arbitrary two dimensional polygonal gate.

polytopeGate Describes a region that is the convex hull of the given points. This gate can exist in dimensions higher than 2, unlike the `polygonGate`.

ellipsoidGate Describes an ellipsoidal region in two or more dimensions

These gates are all described in more or less the same manner (see man pages for more details):

```
rectGate <- rectangleGate(filterId="Fluorescence Region",
                          "FL1-H"=c(0, 12), "FL2-H"=c(0, 12))
```

In addition, we introduce the notion of data-driven gates, or filters, not usually found in flow cytometry software. In these approaches, the necessary parameters are computed based on the properties of the underlying data, for instance by modelling data distribution or by density estimation :

norm2Filter A robust method for finding a region that most resembles a bivariate Normal distribution.

kmeansFilter Identifies populations based on a one dimensional k-means clustering operation. Allows the specification of **multiple** populations.

4.2 Count Statistics

When we have constructed a filter, we can apply it in two basic ways. The first is to collect simple summary statistics on the number and proportion of events considered to be contained within the gate or filter. This is done using the `filter` method. The first step is to apply our filter to some data

```
result = filter(fs[[1]],rectGate)
result

## A filterResult produced by the filter named 'Fluorescence Region'
```

As we can see, we have returned a *filterResult* object, which is in turn a filter allowing for reuse in, for example, subsetting operations. To obtain count and proportion statistics, we take the summary of this *filterResult*, which returns a list of summary values:

```
summary(result)

## Fluorescence Region+: 9811 of 10000 events (98.11%)

summary(result)$n

## [1] 10000

summary(result)$true

## [1] 9811

summary(result)$p

## [1] 0.9811
```

A filter which contains multiple populations, such as the *kmeansFilter*, can return a list of summary lists:

```
summary(filter(fs[[1]], kmeansFilter("FSC-H"=c("Low", "Medium", "High"),
                                     filterId="myKMeans")))

## Low: 2518 of 10000 events (25.18%)
## Medium: 5109 of 10000 events (51.09%)
## High: 2373 of 10000 events (23.73%)
```

A filter may also be applied to an entire *flowSet*, in which case it returns a list of *filterResult* objects:

```
filter(fs,rectGate)

## A list of filterResults for a flowSet containing 5 frames
## produced by the filter named 'Fluorescence Region'
```

4.3 Subsetting

To subset or split a *flowFrame* or *flowSet*, we use the `Subset` and `split` methods respectively. The first, `Subset`, behaves similarly to the standard R `subset` function, which unfortunately could not used. For example, recall from our initial plots of this data that the morphology parameters, Forward Scatter and Side Scatter contain a large more-or-less ellipse shaped population. If we wished to deal only with that population, we might use `Subset` along with a *norm2Filter* object as follows:

```
morphGate <- norm2Filter("FSC-H", "SSC-H", filterId="MorphologyGate",
                         scale=2)
smaller <- Subset(fs, morphGate)
fs[[1]]

## flowFrame object 'NA'
## with 10000 cells and 7 observables:
##      name       desc range minRange maxRange
## $P1 FSC-H FSC-Height  1024        0     1023
## $P2 SSC-H SSC-Height  1024        0     1023
## $P3 FL1-H       <NA>  1024        1    10000
## $P4 FL2-H       <NA>  1024        1    10000
## $P5 FL3-H       <NA>  1024        1    10000
## $P6 FL1-A       <NA>  1024        0     1023
## $P7 FL4-H       <NA>  1024        1    10000
## 141 keywords are stored in the 'description' slot

smaller[[1]]

## flowFrame object 'NA'
## with 8312 cells and 7 observables:
##      name       desc range minRange maxRange
## $P1 FSC-H FSC-Height  1024        0     1023
## $P2 SSC-H SSC-Height  1024        0     1023
## $P3 FL1-H       <NA>  1024        1    10000
## $P4 FL2-H       <NA>  1024        1    10000
## $P5 FL3-H       <NA>  1024        1    10000
## $P6 FL1-A       <NA>  1024        0     1023
## $P7 FL4-H       <NA>  1024        1    10000
## 141 keywords are stored in the 'description' slot
```

Notice how the `smaller` *flowFrame* objects contain fewer events. Now imagine we wanted to use a *kmeansFilter* as before to split our first fluorescence parameter into three populations. To do this we employ the `split` function:

```
split(smaller[[1]], kmeansFilter("FSC-H"=c("Low","Medium","High"),
                                 filterId="myKMeans"))

## $Low
## flowFrame object 'NA (Low)'
## with 2422 cells and 7 observables:
##      name       desc range minRange maxRange
## $P1 FSC-H FSC-Height  1024        0     1023
## $P2 SSC-H SSC-Height  1024        0     1023
## $P3 FL1-H       <NA>  1024        1    10000
## $P4 FL2-H       <NA>  1024        1    10000
## $P5 FL3-H       <NA>  1024        1    10000
## $P6 FL1-A       <NA>  1024        0     1023
## $P7 FL4-H       <NA>  1024        1    10000
## 141 keywords are stored in the 'description' slot
##
## $Medium
## flowFrame object 'NA (Medium)'
## with 3563 cells and 7 observables:
##      name       desc range minRange maxRange
## $P1 FSC-H FSC-Height  1024        0     1023
## $P2 SSC-H SSC-Height  1024        0     1023
## $P3 FL1-H       <NA>  1024        1    10000
## $P4 FL2-H       <NA>  1024        1    10000
## $P5 FL3-H       <NA>  1024        1    10000
## $P6 FL1-A       <NA>  1024        0     1023
## $P7 FL4-H       <NA>  1024        1    10000
## 141 keywords are stored in the 'description' slot
##
## $High
## flowFrame object 'NA (High)'
## with 2327 cells and 7 observables:
##      name       desc range minRange maxRange
## $P1 FSC-H FSC-Height  1024        0     1023
## $P2 SSC-H SSC-Height  1024        0     1023
## $P3 FL1-H       <NA>  1024        1    10000
## $P4 FL2-H       <NA>  1024        1    10000
## $P5 FL3-H       <NA>  1024        1    10000
## $P6 FL1-A       <NA>  1024        0     1023
## $P7 FL4-H       <NA>  1024        1    10000
## 141 keywords are stored in the 'description' slot
```

or for an entire *flowSet*

```
split(smaller, kmeansFilter("FSC-H"=c("Low", "Medium", "High"),
                            filterId="myKMeans"))

## $Low
## A flowSet with 5 experiments.
##
## An object of class 'AnnotatedDataFrame'
##   rowNames: NA fitc ... 7AAD (5 total)
##   varLabels: name Filename population
##   varMetadata: labelDescription
##
##   column names:
##   FSC-H SSC-H FL1-H FL2-H FL3-H FL1-A FL4-H
##
## $Medium
## A flowSet with 5 experiments.
##
## An object of class 'AnnotatedDataFrame'
##   rowNames: NA fitc ... 7AAD (5 total)
##   varLabels: name Filename population
##   varMetadata: labelDescription
##
##   column names:
##   FSC-H SSC-H FL1-H FL2-H FL3-H FL1-A FL4-H
##
## $High
## A flowSet with 5 experiments.
##
## An object of class 'AnnotatedDataFrame'
##   rowNames: NA fitc ... 7AAD (5 total)
##   varLabels: name Filename population
##   varMetadata: labelDescription
##
##   column names:
##   FSC-H SSC-H FL1-H FL2-H FL3-H FL1-A FL4-H
```

4.4 Combining Filters

Of course, most filtering operations consist of more than one gate. To combine gates and filters we use the standard R Boolean operators: &, — and ! to construct an intersection, union and complement respectively:

```
rectGate & morphGate

## filter 'Fluorescence Region and MorphologyGate'
## the intersection between the 2 filters
##
## Rectangular gate 'Fluorescence Region' with dimensions:
##   FL1-H: (0,12)
##   FL2-H: (0,12)
##
## norm2Filter 'MorphologyGate' in dimensions FSC-H and SSC-H with parameters:
##   method: covMcd
##   scale.factor: 2
##   n: 50000

rectGate | morphGate

## filter 'Fluorescence Region or MorphologyGate'
## the union of the 2 filters
##
## Rectangular gate 'Fluorescence Region' with dimensions:
##   FL1-H: (0,12)
##   FL2-H: (0,12)
##
## norm2Filter 'MorphologyGate' in dimensions FSC-H and SSC-H with parameters:
##   method: covMcd
##   scale.factor: 2
##   n: 50000

!morphGate

## filter 'not MorphologyGate', the complement of
## norm2Filter 'MorphologyGate' in dimensions FSC-H and SSC-H with parameters:
##   method: covMcd
##   scale.factor: 2
##   n: 50000
```

we also introduce the notion of the subset operation, denoted by either %subset% or %&%. This combination of two gates first performs a subsetting operation on the input *flowFrame* using the right-hand filter and then applies the left-hand filter. For example,

```
summary(filter(smaller[[1]],rectGate %&% morphGate))

## Fluorescence Region in MorphologyGate+: 7187 of 8312 events (86.47%)
```

first calculates a subset based on the `morphGate` filter and then applies the `rectGate`.

4.5 Transformation Filters

Finally, it is sometimes desirable to construct a filter with respect to transformed parameters. To allow for this in our filtering constructs we introduce a special form of the `transform` method along with another filter combination operator %on%, which can be applied to both filters and *flowFrame* or *flowSet* objects. To specify our transform filter we must first construct a transform list using a simplified version of the `transform` function:

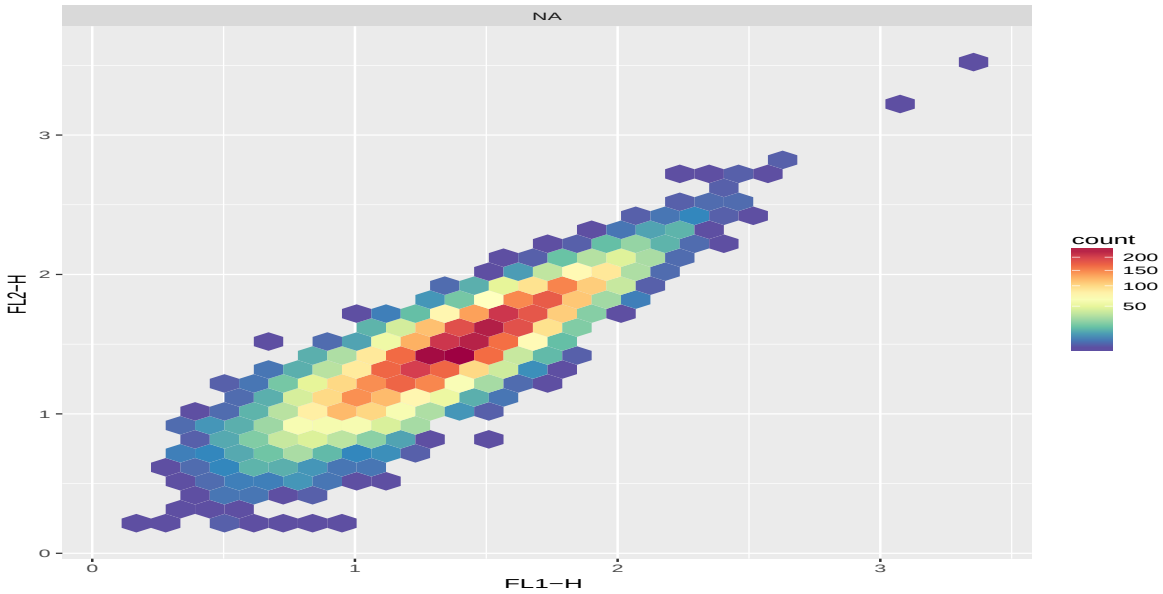
```
tFilter <- transform("FL1-H"=log,"FL2-H"=log)
tFilter

## An object of class "transformList"
## Slot "transforms":
## [[1]]
## transformMap for parameter 'FL1-H' mapping to 'FL1-H'
##
## [[2]]
## transformMap for parameter 'FL2-H' mapping to 'FL2-H'
##
##
## Slot "transformationId":
## [1] "defaultTransformation"
```

Note that this version of the transform filter does not take parameters on the right-hand side–the functions can only take a single vector that is specified by the parameter on the left-hand side. In this case those parameters are "FL1-H" and "FL2-H." The function also does not take a specific *flowFrame* or *flowSet* allowing us to use this with any appropriate data. We can then construct a filter with respect to the transform as follows:

```
rect2 <- rectangleGate(filterId="Another Rect", "FL1-H"=c(1,2),
"FL2-H"=c(2,3)) %on% tFilter
rect2

## transformed filter 'Another Rect on transformed values of FL1-H,FL2-H'
```

Additionally, we can use this construct directly on a *flowFrame* or *flowSet* by moving the transform to the left-hand side and placing the data on the right-hand side:

```
autoplot(tFilter %on% smaller[[1]], "FL1-H","FL2-H")
```



which has the same effect as the log transform used earlier.

5 GatingSet

filterSets are very limited in their use for complex analysis work flows. They are result-centric and it is hard to access intermediate results. flowWorkspace and openCyto framework (http://opencyto.org) offers much more versatile tools for such tasks though the *GatingSet* class (the old *workFlow* is now deprecated). The general idea is to let the software handle the organization of intermediate results, and operations and to provide a unified API to access and summarize these operations.

5.1 Abstraction of GatingSet

There are two classes in flowWorkspace that are used to abstract work flows: *GatingSet* objects are the basic container holding all the necessary bits and pieces and they are the main structure for user interaction. It is the container storing multiple *GatingHierarchy* objects which are associate with individual samples. One can think of *GatingSet* corresponds to (flowSet) and *GatingHierarchy* corresponds to (flowFrame).

It is important to know that *GatingSet* use 'exteranl pointer' to store the 'gating tree' and thus most of its accessors have a reference semantic instead of the pass-by-value semantic that is usually found in the R language. The main consequence on the user-level is the fact that direct assignments to a *GatingSet* object are usually not necessary; i.e., functions that operate on the *GatingSet* have the potential side-effect of modifying the object.

5.2 Creating *GatingSet* objects

Before creating a 'GatingSet', we need to have flow data loaded into R as a *flowSet* or *ncdfFlowSet*(the disk-based 'flowSet', to handle large data set that are too big for memory, see (ncdfFlow) package).

```
library(flowWorkspace)
fcsfiles <- list.files(pattern = "CytoTrol"
                       , system.file("extdata", package = "flowWorkspaceData")
                       , full = TRUE)
fs <- read.flowSet(fcsfiles)
```

Then *GatingSet* can be created using the constructor `GatingSet`.

```
gs <- GatingSet(fs)

## ..
## done!

gs

## A GatingSet with 2 samples
```

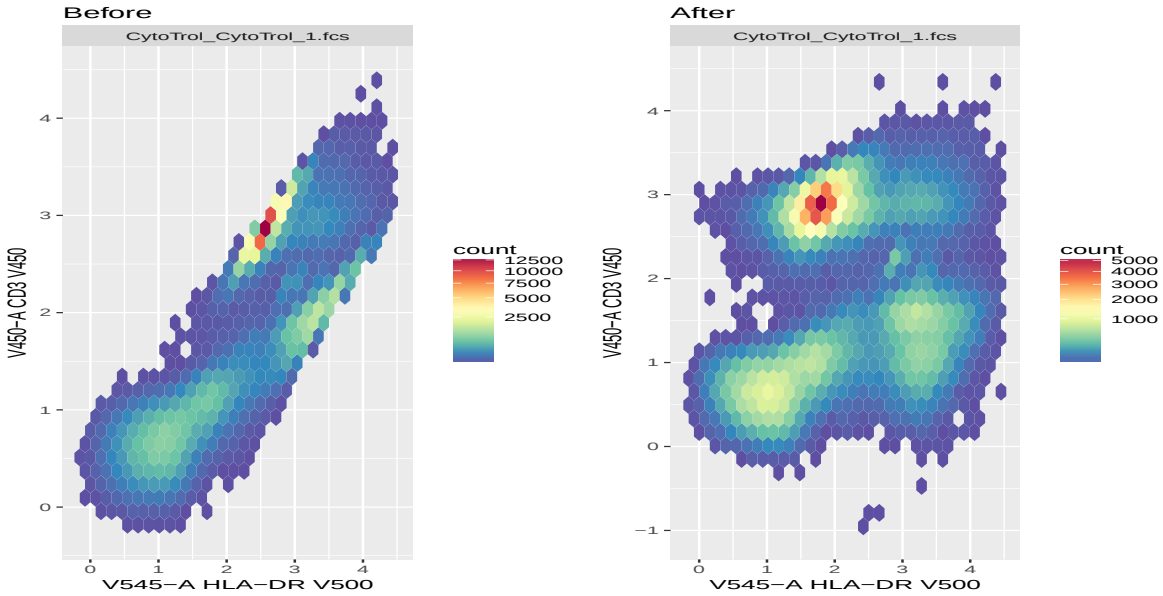
Normally, we want to compensate the data firstly by using a user supplied compensation matrix:

```
## loading R object...
## loading tree object...
## Done
```

```
comp

## Compensation object 'defaultCompensation':
##          B710-A    G560-A       G780-A     R660-A    R780-A   V450-A
## B710-A 1.000000 0.0009476  0.071170002  0.0362400 0.1800000 0.007104
## G560-A 0.115400 1.0000000  0.009097001  0.0018360 0.0000000 0.000000
## G780-A 0.014280 0.0380000  1.000000000  0.0006481 0.1500000 0.000000
## R660-A 0.005621 0.0000000  0.006604000  1.0000000 0.1786000 0.000000
## R780-A 0.000000 0.0000000  0.035340000  0.0102100 1.0000000 0.000000
## V450-A 0.000000 0.0000000 -0.059999999 -0.0400000 0.0000000 1.000000
## V545-A 0.002749 0.0000000  0.000000000  0.0000000 0.0006963 0.035000
##          V545-A
## B710-A 0.007608
## G560-A 0.000000
## G780-A 0.000000
## R660-A 0.000000
## R780-A 0.000000
## V450-A 0.410000
## V545-A 1.000000

gs <- compensate(gs, comp)
```

Here is the effect of compensation:

```
fs_comp <- getData(gs)
transList <- estimateLogicle(fs[[1]], c("V545-A","V450-A"))
library(gridExtra)
p1 <- autoplot(transform(fs[[1]], transList)
               , 'V545-A', 'V450-A') + ggtitle("Before")
p2 <- autoplot(transform(fs_comp[[1]], transList)
               , 'V545-A', 'V450-A') + ggtitle("After")
grid.arrange(as.ggplot(p1), as.ggplot(p2), ncol = 2)
```



We can query the available nodes in the *GatingSet* using the `getNodes` method:

```
getNodes(gs)

## [1] "root"
```

It shows the only node 'root' which corresponds to the raw 'flow data' just added.

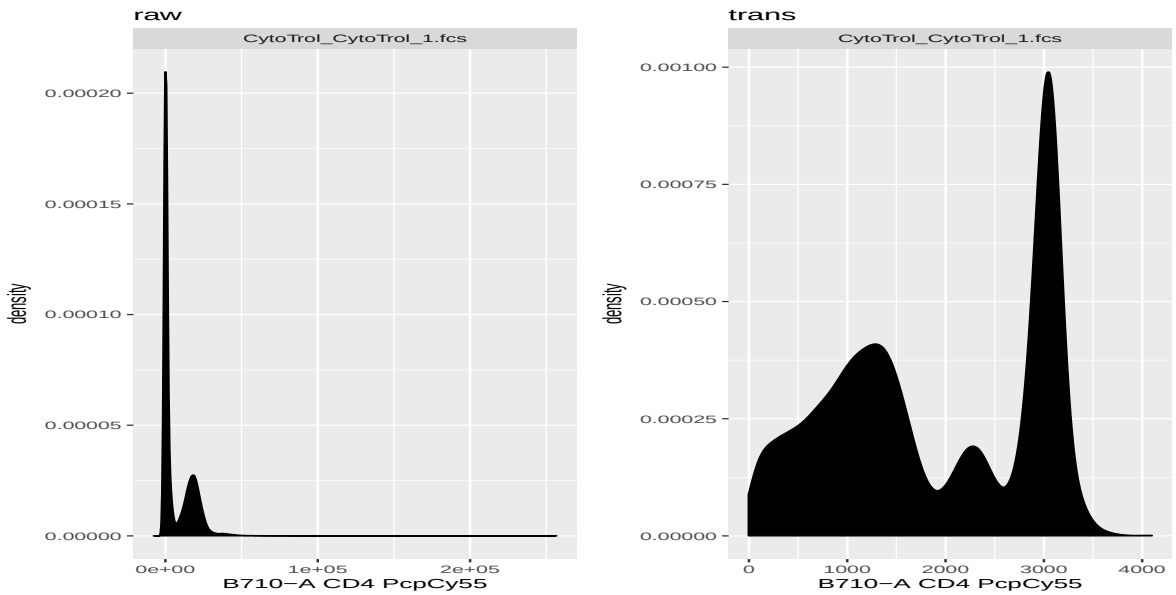
5.3 transform the data

Transformation can be either done on 'flowSet' before constructing 'GatingSet' or a *transformerList* can be directly added to a *GatingSet*:

```
biexpTrans <- flowJo_biexp_trans(channelRange=4096, maxValue=262144
                                 , pos=4.5,neg=0, widthBasis=-10)
chnls <- parameters(comp)
tf <- transformerList(chnls, biexpTrans)

#or use estimateLogicle directly on GatingHierarchy object to generate transformer
#tf <- estimateLogicle(gs[[1]], chnls)
```

```
gs <- transform(gs, tf)
```

```
p1 <- autoplot(fs_comp[[1]], "B710-A") + ggtitle("raw")
p2 <- autoplot(flowData(gs)[[1]], "B710-A") +
          ggtitle("trans") +
          ggcyto_par_set(limits = "instrument")
grid.arrange(as.ggplot(p1), as.ggplot(p2), ncol = 2)
```



Note that we did assign the return value of `transform` back to `gs`. This is because 'flow data' is stored as R object and thus transforming the data still follows the pass-by-value semantics.

5.4 Add the gates

Some basic flowCore *filter* can be added to a *GatingSet*:

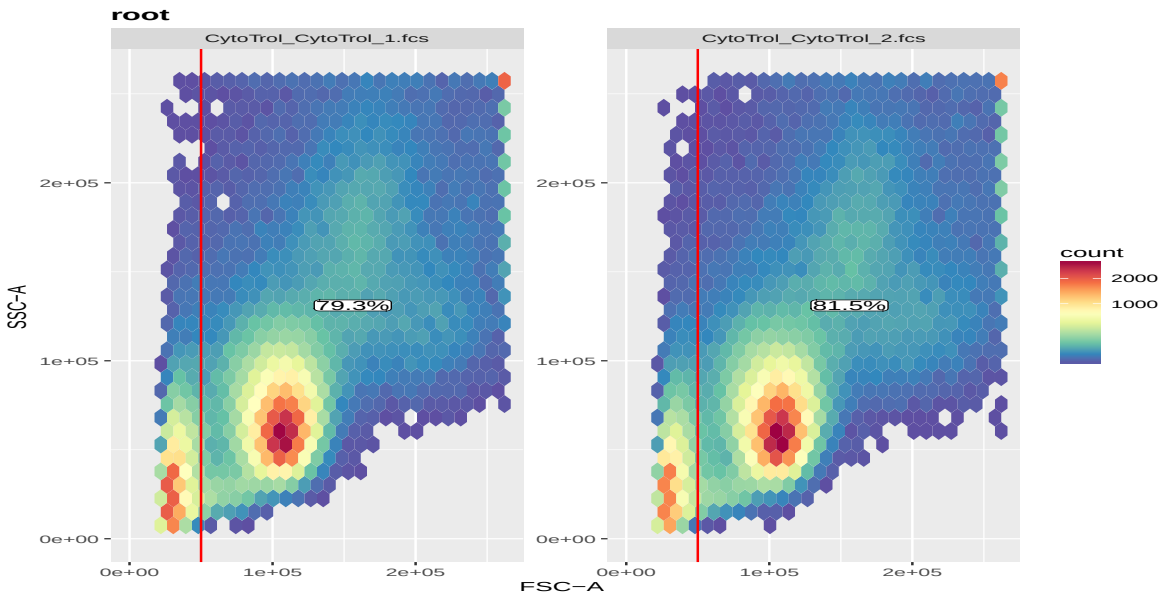
```
rg1 <- rectangleGate("FSC-A"=c(50000, Inf), filterId="NonDebris")
add(gs, rg1, parent = "root")

## replicating filter 'NonDebris' across samples!

## [1] 2

getNodes(gs)

## [1] "root"       "/NonDebris"

# gate the data
recompute(gs)
```

```
## .
## .
## done!
```

As we see, here we don't need to assign *GatingSet* back because all the modifications are made in place to the *external pointer* rather than the R object itself. And now there is one new population node under the 'root' node called 'NonDebris'. The node is named after the 'filterId' of the gate if not explictly supplied. After the gates are added, the actual gating process is done by explictly calling `recompute` method. Note that the numeric value it returns is the internal ID for the new population just added, which can be normally ignored since the gating path instead of numeric id is recommended way to refer to population nodes later.

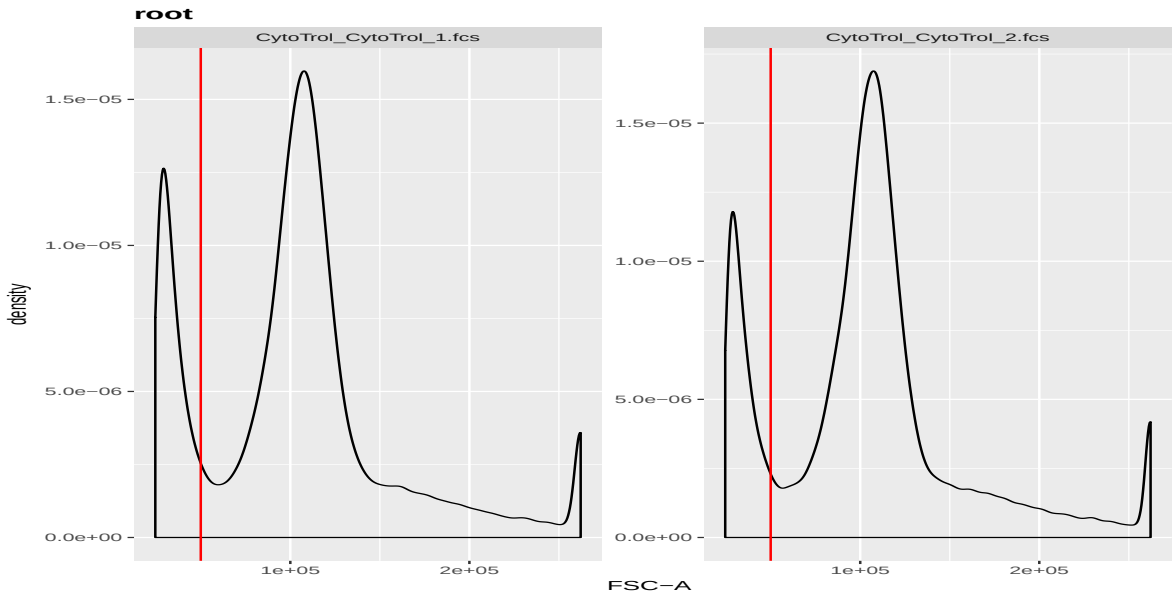
To view the gate we just added,

```
autoplot(gs, "NonDebris")
```



Since They are '1d' gates, we can also dispaly it in 'densityplot'

```
ggcyto(gs, aes(x = `FSC-A`)) + geom_density() + geom_gate("NonDebris")
```



To get population statistics for the given populatuion

```
getTotal(gs[[1]], "NonDebris")#counts

## [1] 94764

getProp(gs[[1]], "NonDebris")#proportion

## [1] 0.7927985
```

Now we add two more gates:

```
# add the second gate
mat <- matrix(c(54272,59392,259071.99382782
                ,255999.994277954,62464,43008,70656
                ,234495.997428894,169983.997344971,34816)
              , nrow = 5)
colnames(mat) <-c("FSC-A", "FSC-H")
mat

##       FSC-A  FSC-H
## [1,]  54272  43008
## [2,]  59392  70656
## [3,] 259072 234496
## [4,] 256000 169984
## [5,]  62464  34816

pg <- polygonGate(mat)
add(gs, pg, parent = "NonDebris", name = "singlets")
```

```
## replicating filter 'defaultPolygonGate' across samples!

## [1] 3

# add the third gate
rg2 <- rectangleGate("V450-A"=c(2000, Inf))
add(gs, rg2, parent = "singlets", name = "CD3")

## replicating filter 'defaultRectangleGate' across samples!

## [1] 4

getNodes(gs)

## [1] "root"                      "/NonDebris"
## [3] "/NonDebris/singlets"       "/NonDebris/singlets/CD3"
```

We see two more nodes are added to 'GatingSet' and the population names are explicitly specified during the adding this time.

quadrantGate that results in four sub-populationsis also supported.

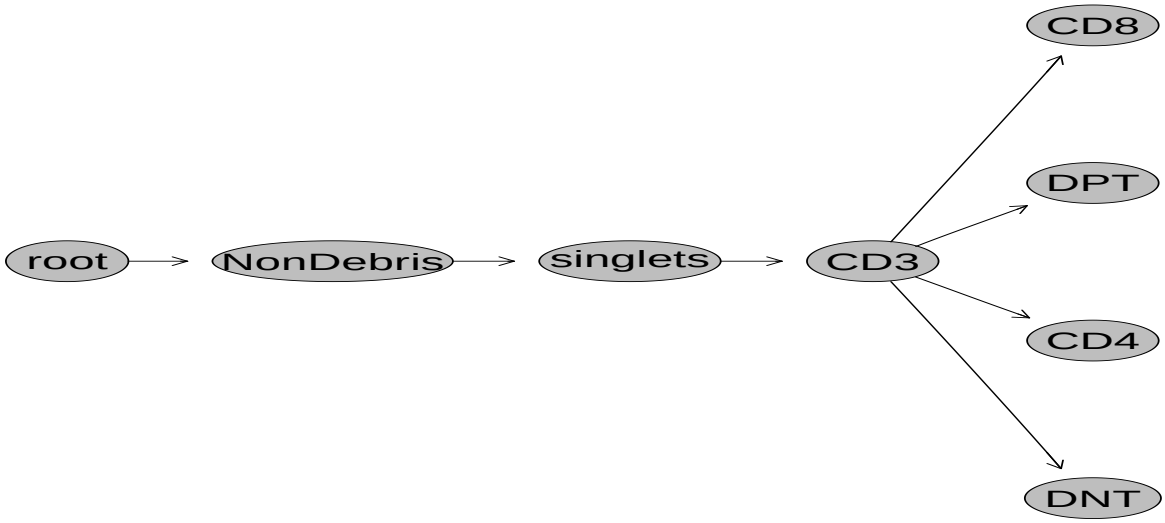
```
qg <- quadGate("B710-A" = 2000, "R780-A" = 3000)
add(gs, qg, parent="CD3", names = c("CD8", "DPT", "CD4", "DNT"))

## replicating filter 'defaultQuadGate' across samples!

## [1] 5 6 7 8

getChildren(gs[[1]], "CD3")

## [1] "/NonDebris/singlets/CD3/CD8" "/NonDebris/singlets/CD3/DPT"
## [3] "/NonDebris/singlets/CD3/CD4" "/NonDebris/singlets/CD3/DNT"

# gate the data from "singlets"
recompute(gs, "singlets")

## .
## .
## done!
```

Here we see four children nodes are added to 'CD3' parent node. Four quadrants are named explicitly through 'names' argument by clock-wise order (start from top-left quadrant). 'recomputing' only needs to be done once from the first ungated node, which will automatically compute all its descendants.

To plot the underlying tree

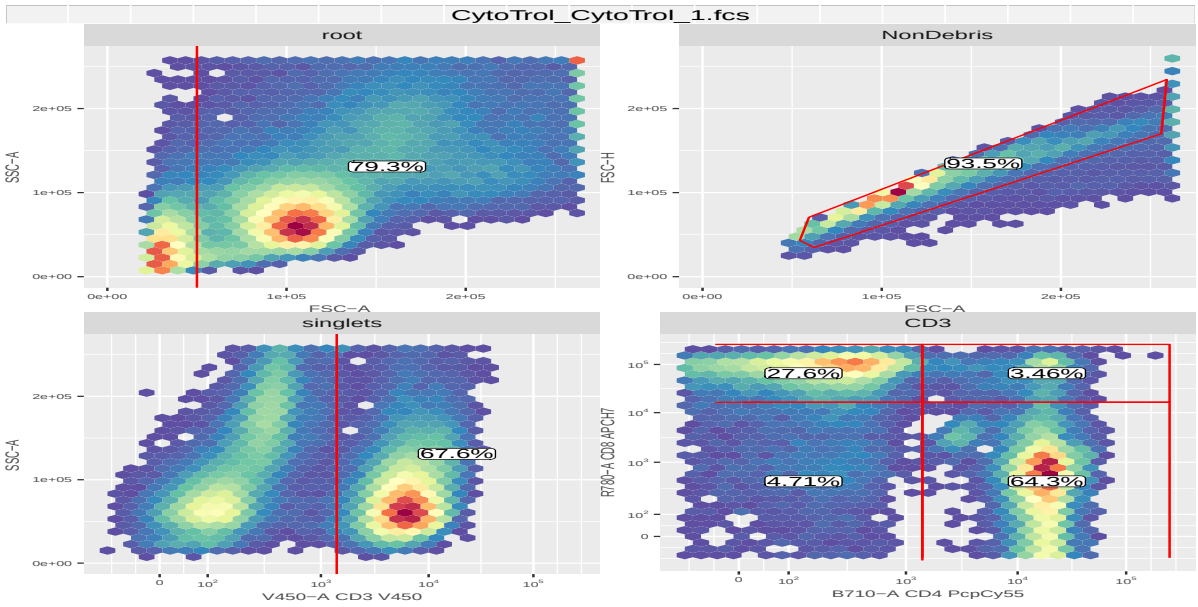
```r
plot(gs)
```

```
## Loading required package:  Rgraphviz
## Loading required package:  graph
## Loading required package:  BiocGenerics
## Loading required package:  parallel
##
## Attaching package:  'BiocGenerics'
## The following objects are masked from 'package:parallel':
##
##    clusterApply, clusterApplyLB, clusterCall, clusterEvalQ,
##    clusterExport, clusterMap, parApply, parCapply, parLapply,
##    parLapplyLB, parRapply, parSapply, parSapplyLB
## The following object is masked from 'package:gridExtra':
##
##    combine
## The following objects are masked from 'package:flowCore':
##
##    normalize, sort
## The following objects are masked from 'package:stats':
##
##    IQR, mad, xtabs
## The following objects are masked from 'package:base':
##
##    Filter, Find, Map, Position, Reduce, anyDuplicated, append,
##    as.data.frame, cbind, colnames, do.call, duplicated, eval,
##    evalq, get, grep, grepl, intersect, is.unsorted, lapply,
##    lengths, mapply, match, mget, order, paste, pmax, pmax.int,
##    pmin, pmin.int, rank, rbind, rownames, sapply, setdiff, sort,
##    table, tapply, union, unique, unsplit, which, which.max,
##    which.min
## Loading required package:  grid
```



To plot all gates for one sample

```
autoplot(gs[[1]])
```



To retreive the underlying flow data for a gated *population*

```
fs_nonDebris <- getData(gs, "NonDebris")
fs_nonDebris

## A flowSet with 2 experiments.
##
##   column names:
##   FSC-A FSC-H FSC-W SSC-A B710-A R660-A R780-A V450-A V545-A G560-A G780-A Time
```

```
nrow(fs_nonDebris[[1]])

## [1] 94764

nrow(fs[[1]])

## [1] 119531
```

To get all the population statistics

```
getPopStats(gs)

##                          name Population    Parent Count ParentCount
##  1: CytoTrol_CytoTrol_1.fcs  NonDebris      root 94764      119531
##  2: CytoTrol_CytoTrol_1.fcs   singlets NonDebris 88586       94764
##  3: CytoTrol_CytoTrol_1.fcs        CD3  singlets 59911       88586
##  4: CytoTrol_CytoTrol_1.fcs        CD8       CD3 16515       59911
##  5: CytoTrol_CytoTrol_1.fcs        DPT       CD3  2070       59911
##  6: CytoTrol_CytoTrol_1.fcs        CD4       CD3 38506       59911
##  7: CytoTrol_CytoTrol_1.fcs        DNT       CD3  2820       59911
##  8: CytoTrol_CytoTrol_2.fcs  NonDebris      root 94290      115728
##  9: CytoTrol_CytoTrol_2.fcs   singlets NonDebris 88334       94290
## 10: CytoTrol_CytoTrol_2.fcs        CD3  singlets 59845       88334
## 11: CytoTrol_CytoTrol_2.fcs        CD8       CD3 16774       59845
## 12: CytoTrol_CytoTrol_2.fcs        DPT       CD3  2200       59845
## 13: CytoTrol_CytoTrol_2.fcs        CD4       CD3 38127       59845
## 14: CytoTrol_CytoTrol_2.fcs        DNT       CD3  2744       59845
```

5.5 Removing nodes from *GatingSet* object

There are dependencies between Rclassnodes in the hierarchical structure of the *GatingSet* object. Thus, removing a particular node means also removing all of its associated child *nodes*.

```
Rm('CD3', gs)
getNodes(gs)

## [1] "root"                "/NonDebris"          "/NonDebris/singlets"

Rm('NonDebris', gs)
getNodes(gs)

## [1] "root"
```

Now for the larger data set, it would be either inaccurate to apply the same hard-coded gate to all samples or impractical to manually set the gate coordinates for each individual sample. openCyto((**?**)) provides some data-driven gating functions to automatically generate these gates.

For example, `mindensity` can be used for estimating 'nonDebris' gate for each sample.

```
library(openCyto)
thisData <- getData(gs)
nonDebris_gate <- fsApply(thisData
                          , function(fr)
                            openCyto:::.mindensity(fr, channels = "FSC-A"))
add(gs, nonDebris_gate, parent = "root", name = "nonDebris")

## [1] 2

recompute(gs)

## ..
## done!
```

`singeltGate` can be used for estimating 'singlets'

```
thisData <- getData(gs, "nonDebris") #get parent data
singlet_gate <- fsApply(thisData
                        , function(fr)
                          openCyto:::.singletGate(fr, channels =c("FSC-A", "FSC-H"
add(gs, singlet_gate, parent = "nonDebris", name = "singlets")

## [1] 3

recompute(gs)

## ..
## done!
```

and then use `mindensity` again for "CD3" gate

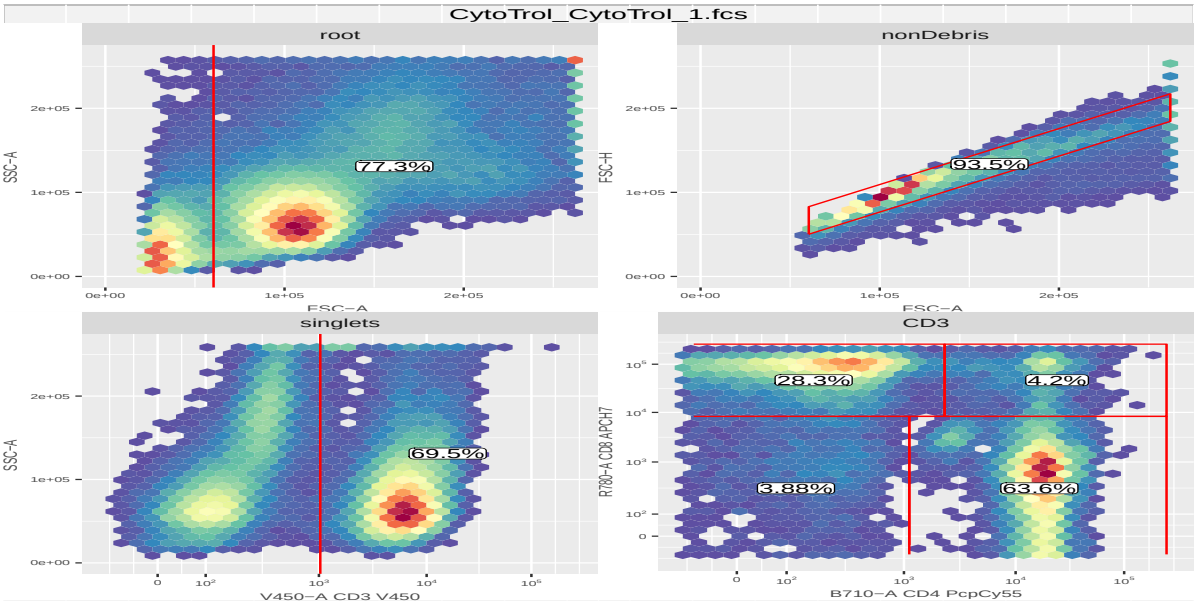
```
thisData <- getData(gs, "singlets") #get parent data
CD3_gate <- fsApply(thisData
                    , function(fr)
                      openCyto:::.mindensity(fr, channels ="V450-A"))
add(gs, CD3_gate, parent = "singlets", name = "CD3")

## [1] 4

recompute(gs)

## ..
## done!
```

and then use more adanced version of 'quadGate': `quadGate.seq` for gating "CD4" and "CD8" sequentially:

```
thisData <- getData(gs, "CD3") #get parent data
Tsub_gate <- fsApply(thisData
                     , function(fr)
                       openCyto::quadGate.seq(fr
                            , channels = c("B710-A", "R780-A")
                            , gFunc = 'mindensity'
                           )
                    )
add(gs, Tsub_gate, parent = "CD3", names = c("CD8", "DPT", "CD4", "DNT"))

## q1 q2 q3 q4
##  5  6  7  8

recompute(gs)

## ..
## done!
```

and then plot the gates

```
autoplot(gs[[1]])
```



Note that in order to get parent gated data by 'getData' we have to 'recompute' after adding each gate. And the result is very similar to the manual gates but the gating process is more data-driven and more consistent across samples.

The further automated the process, a gating pipeline can be established through OpenCyto((**?**)) that defines the hierarchical gating template in a text-based csv file. (See more details from http://openCyto.org)