flowCore: data structures package for flow cytometry data

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

October 18, 2010

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 (Gasparetto et al., 2004). 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 (Brazma, 2001). Standardization is critical when developing new high throughput technologies and their associated information services (Brazma, 2001; Chicurel, 2002; Boguski and McIntosh, 2003). Standardization efforts have been made in clinical cell analysis by flow cytometry (Keeney et al., 2004), 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 (Bagwell, 2004; Braylan, 2004; Parks, 1997; Suni et al., 2003). 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 (LeMeur and Hahne, 2006) into one core package which is compliant with the data exchange standards that are currently developed in the community (Spidlen et al., 2006).

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 `flowFrame`s 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 "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    8.148    12.11    2.249    0.00    6.624 122.0
Median    441  189.0  135.800    22.54    5.684   26.00   12.330 288.0
Mean      492  277.9  158.700   106.60    8.488   34.08  141.300 294.8
3rd Qu.   518  270.0  242.700    51.13   10.770   51.00   33.490 457.5
Max.     1023 1023.0 3681.000 10000.00 3304.000 1023.00 9910.000 626.0
```

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 0.0007149 0.001111 0.0001249 0.00000 0.0005624
Median  0.43110 0.18480 0.0134800 0.002154 0.0004684 0.02542 0.0011330
Mean    0.48090 0.27170 0.0157700 0.010560 0.0007489 0.03331 0.0140300
3rd Qu. 0.50640 0.26390 0.0241800 0.005014 0.0009772 0.04985 0.0032490
Max.    1.00000 1.00000 0.3681000 1.000000 0.3303000 1.00000 0.9910000
            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                     1024        1    10000
$P4 FL2.H                     1024        1    10000
$P5 FL3.H                     1024        1    10000
$P6 FL1.A               <NA>  1024        0     1023
$P7 FL4.H                     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        1024        1    10000
$P4 FL2-H        1024        1    10000
$P5 FL3-H        1024        1    10000
$P7 FL4-H        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.

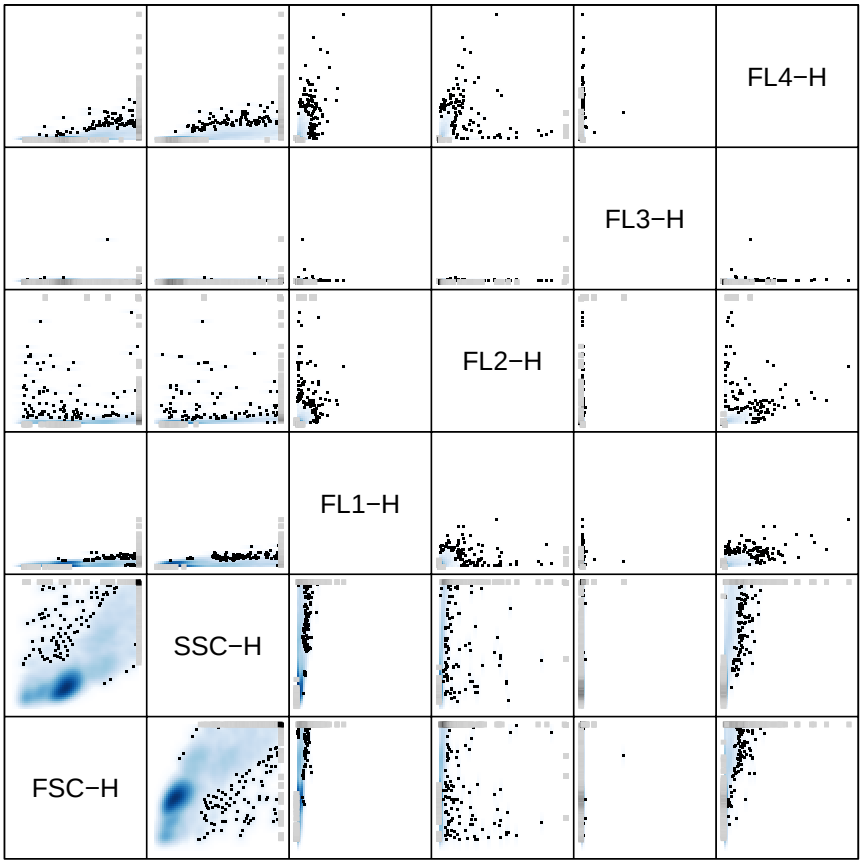
```
> lines <- sample(100:500, 50)
> y <- read.FCS(file.name, which.lines = lines)
> y

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                     1024        1    10000
$P4 FL2-H                     1024        1    10000
$P5 FL3-H                     1024        1    10000
$P6 FL1-A               <NA>  1024        0     1023
$P7 FL4-H                     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 lattice graphics system is implemented by the flowViz package, also included as part of Bioconductor. Here, we will only introduce the standard `plot` function. The basic plot provides a simple pairs plot of all parameters:

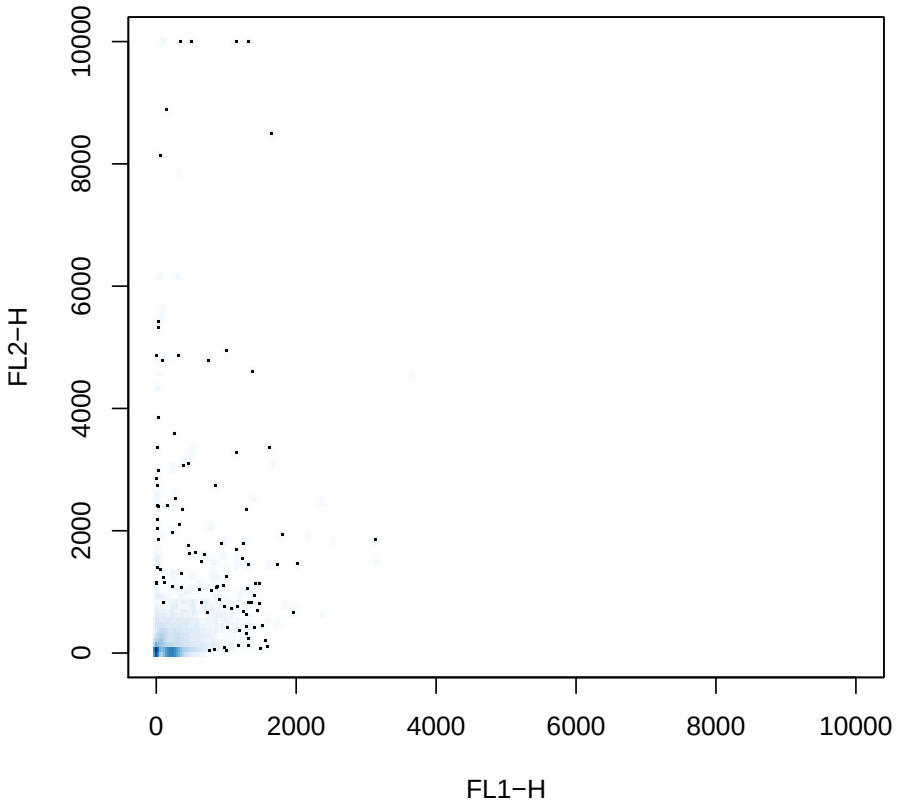
```
> library(flowViz)
> plot(x)
```



Scatter Plot Matrix

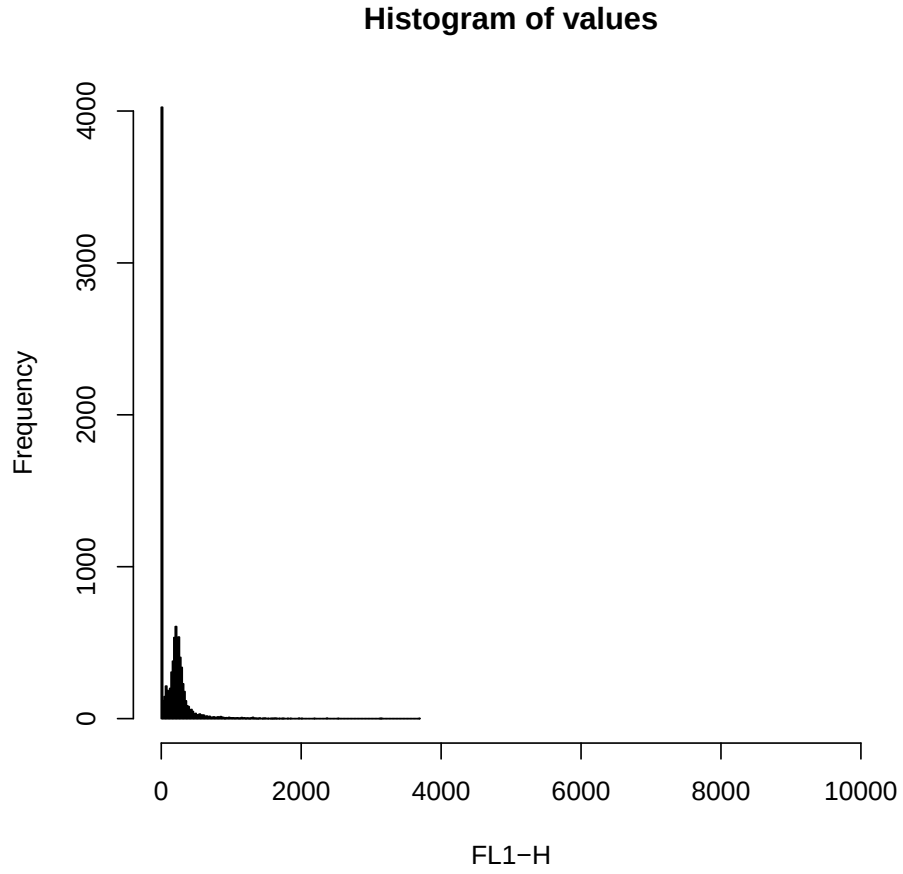
To control the parameters being plotted we can supply a `y` value in the form of a character vector. If we choose exactly two parameters this will create a bivariate density plot.

```
> plot(x, c("FL1-H", "FL2-H"))
```



However, if we only supply a single parameter we instead get a univariate histogram, which also accepts the usual histogram arguments:

```
> plot(x, "FL1-H", breaks = 256)
```



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.flowSet`s 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   4.110368   4.538282   3.656368     0   7.247948
fitc   436   128 936.811048 229.975372  33.490890   217   8.295949
pe     438   120  10.204639 796.655892 114.975700     0   9.326033
apc    441   129   4.377753   4.877217   4.790181     0 360.732067
7AAD   429   133   5.010744  15.029018  63.466061     0  20.970227
```

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   4.110368   4.538282   3.656368     0   7.247948
fitc   436   128 936.811048 229.975372  33.490890   217   8.295949
pe     438   120  10.204639 796.655892 114.975700     0   9.326033
apc    441   129   4.377753   4.877217   4.790181     0 360.732067
7AAD   429   133   5.010744  15.029018  63.466061     0  20.970227
```

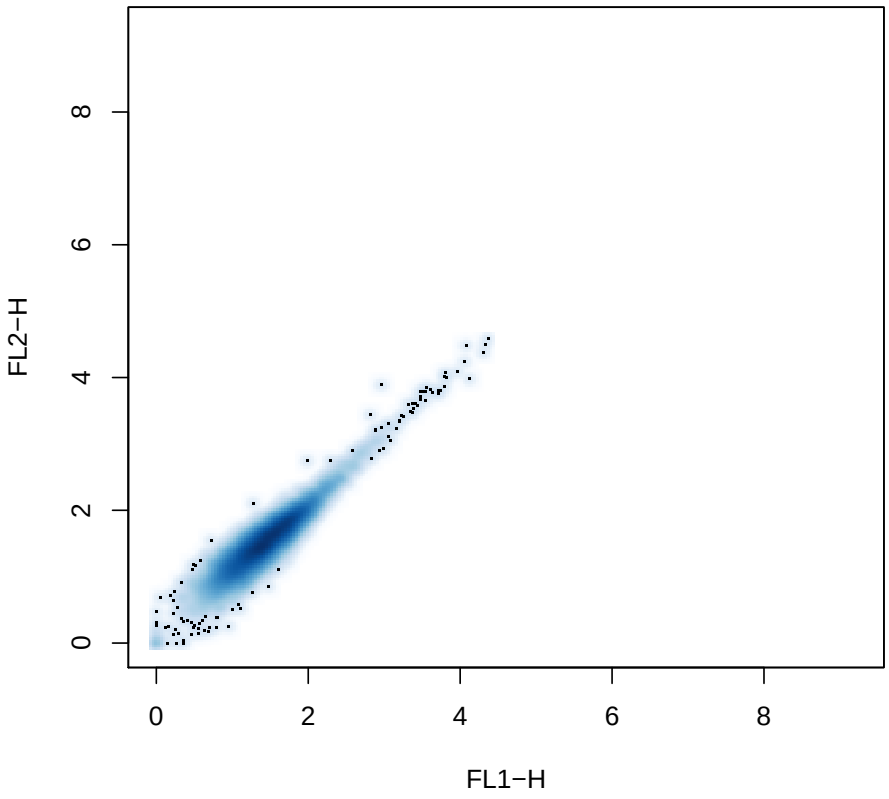
In this case, the `use.exprs` argument is not required in the first case because `each_col` and `each_row` are methods and have been defined to work on *flowFrame* objects by first extracting the intensity data.

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

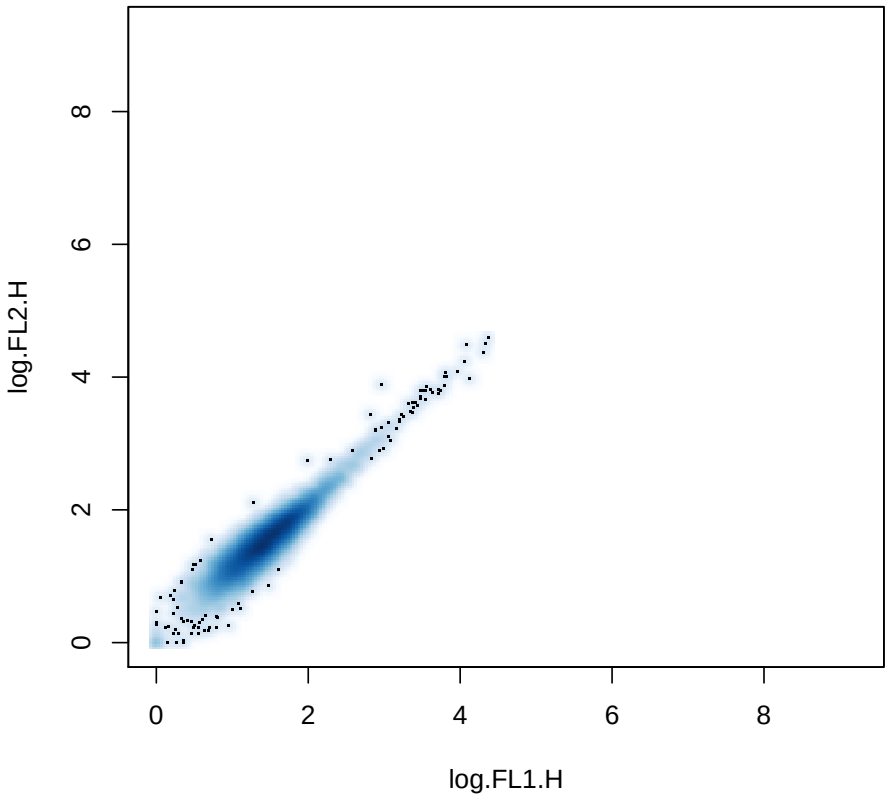
to support filtering features and is strictly more limited than the out-of-line transformation method, which uses R's `transform` function to accomplish the filtering. Like the normal `transform` function, the *flowFrame*is considered to be a data frame with columns named for parameters of the FCS file. For example, if we wished to plot our first *flowFrame*'s first two fluorescence parameters on the log scale we might write:

```
> plot(transform(fs[[1]], `FL1-H` = log(`FL1-H`), `FL2-H` = log(`FL2-H`)),
+     c("FL1-H", "FL2-H"))
```



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

```
> plot(transform(fs[[1]], log.FL1.H = log(`FL1-H`), log.FL2.H = log(`FL2-H`)),
+     c("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 Spidlen et al. (2006) 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
```

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 Spidlen et al. (2008), 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+: 7179 of 8312 events (86.37%)
```

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:

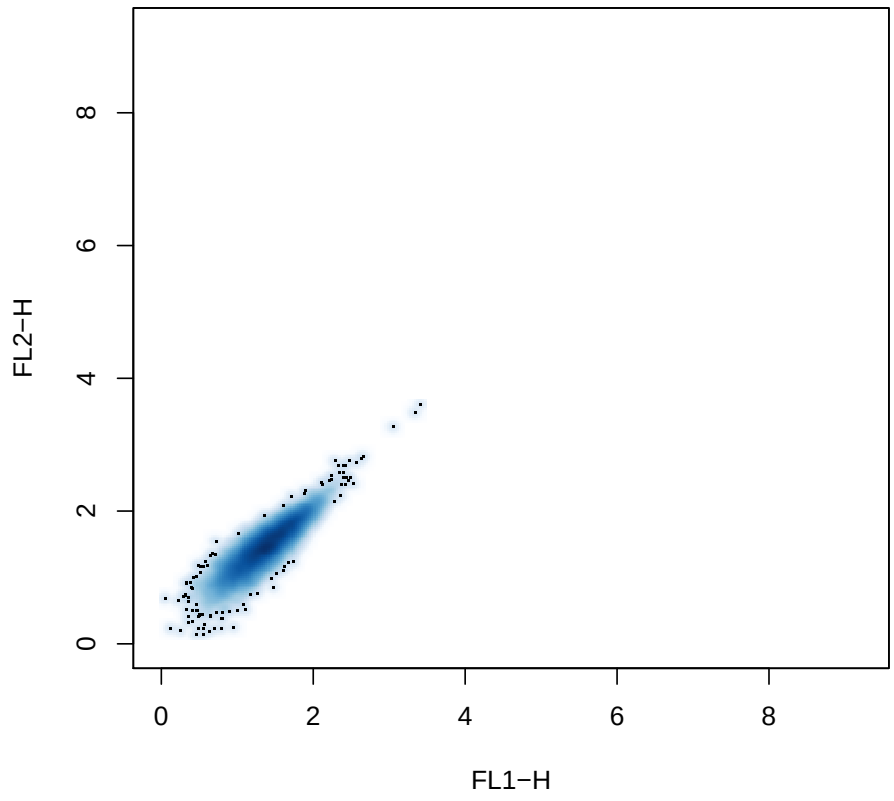
```
> tFilter <- transform(`FL1-H` = log, `FL2-H` = log)
> tFilter

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:

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



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

5 filterSet: Organizing Filtering Strategies

5.1 Building Filter Sets

In addition to allowing for sequential filtering, flowCore also provides a *filterSet* object that serves as an analogue to *flowSets* for filters. The primary use of this object is to organize and manipulate complex gating strategies. For example, recall from the filtering section we had two gates, a "morphologyGate" and a "Fluorescence Region" gate that we used to demonstrate the various logical operators available for gates. To recreate this example as a *filterSet*, we would do the following:

```
> fset1 = filterSet(rectangleGate(filterId = "Fluorescence Region",
+     `FL1-H` = c(50, 100), `FL2-H` = c(50, 100)), norm2Filter("FSC-H",
+     "SSC-H", filterId = "Morphology Gate", scale = 2), ~`Fluorescence Region` &
+     `Morphology Gate`, ~`Fluorescence Region` | `Morphology Gate`,
```

```
+     Debris ~ !`Morphology Gate`, ~`Fluorescence Region` %&% `Morphology Gate`)
> fset1

A set of filter objects:
Debris,Fluorescence Region,Fluorescence Region and Morphology Gate,Fluorescence Region in Mor
```

There are two features of note in `filterSet`, which can also take a single list argument for programmatic creation of *filterSet* objects. The first is that there is a formula interface for the creation of *filter* objects from operators. The formula interface can be used with or without a left-hand side, which specifies an optional filter identifier. If the filter identifier is not specified, as is the case with all but the "Debris" gate in the example above the usual default filter identifier is constructed. Non-formula gates can also have an optional name specified which overrides the filter identifier specified in the gate constructor. This is discouraged at this time as it leads to confusion when creating new filters.

5.2 Manipulating Filter Sets

Manipulating a *filterSet* is done using the normal list element replacement methods, though we do provide a special case where replacing the """" element or the "NULL" element causes the *filterSet* to use the "filterId" slot of the incoming filter to choose a name. Similarly, specifying an identifier will override any "filterId" slot in the original filter.

Additionally, there are several convenience functions and methods available for manipulating a *filterSet*. The `names` function provides a list of filter identifiers in an arbitrary order. The `sort` lists the filter identifiers according to a topological sort of the filter dependencies such that parent filters are always evaluated before their children. Filters without dependencies are provided in an arbitrary relative ordering. To obtain the adjacency matrix for this sorting, the option "dependencies=TRUE" supplied to the `sort` function will attach an "AdjM" attribute to the resulting character vector.

5.3 Using Filter Sets

Though they are not explicitly filters, meaning that they cannot play a role in the construction of sub filters via operators, *filterSet* objects do define `filter` and `split` methods that allow for the two most common use cases. Additionally, a filter can be selected from the *filterSet* using "[[" to perform a `Subset` operation if desired.

For standard filtering operations, the *filterSet* can yield better performance than a manual filtering operation because it ensures that each parent filter is only evaluated once per call to `filter` so that if many sub filters rely on a particular, likely expensive, filtering operation they use the original result rather than recalculating each time. The individual *filterResult* objects may be later extracted for further analysis of desire, making the *filterSet* the logical choice for most filtering operations. Additionally, though not presently implemented, it can allow visualization methods to make more intelligent choices about display when rendering gating strategies because the disposition of the entire filtering graph is known and the filter results are already available.

Like any other filtering operation, `summary` methods are also available to collect the usual sorts of count statistics for a filtering operation:

```
> f = filter(fs, fset1)
> f

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

When splitting a *flowFrame* using `split`, we have two options. The first is to include all of the resultant subsets in the usual way:

```
> split(fs[[1]], fset1, flowSet = TRUE)

A flowSet with 7 experiments.

An object of class "AnnotatedDataFrame"
  rowNames: rest Fluorescence Region ... Fluorescence Region or
    Morphology Gate (7 total)
  varLabels: name population
  varMetadata: labelDescription

  column names:
  FSC-H SSC-H FL1-H FL2-H FL3-H FL1-A FL4-H
```

While this provides access to all subsets, for automated analysis we will often only wish to work with the "leaf" gates (i.e. those without children of their own). In this case, we can specify "drop=TRUE" to obtain only the leaf gates:

```
> split(fs[[1]], fset1, drop = TRUE, flowSet = TRUE)

A flowSet with 7 experiments.

An object of class "AnnotatedDataFrame"
  rowNames: rest Fluorescence Region ... Fluorescence Region or
    Morphology Gate (7 total)
  varLabels: name population
  varMetadata: labelDescription

  column names:
  FSC-H SSC-H FL1-H FL2-H FL3-H FL1-A FL4-H
```

Note that in both cases, we use the "flowSet=TRUE" option that has been added to the `split` function to automatically create a *flowSet* object from the resulting *flowFrame* objects. For `filterSet` objects, the `split` function can take advantage of the added structure in this case to provide some information about the filtering operation itself in the *flowSet*'s metadata as well.

6 Work flows

filterSets are very limited in their use for complex analysis work flows. They are result-centric and it is hard to access intermediate results. flowCore offers much more versatile tools for such tasks though the *workFlow* class. The general idea is to let the software handle the organization of intermediate results, naming schemes and operations and to provide a unified API to access and summarize these operations.

6.1 Abstraction of work flows

There are three classes in flowCore that are used to abstract work flows: *workFlow* objects are the basic container holding all the necessary bits and pieces and they are the main structure for user interaction. *actionItem* objects are abstractions of data analysis operations like gating, transformation or normalization. There are sub-classes for different types of operations. Finally, *view* objects are created when applying *actionItems* to a *workFlow*. One can think of *views* as separate data sets, for instance a subset that is created by a gating operation or the modified data following a transformation. Again, different types of *views* are available through various sub-classes. This structure allows for a unified API in which the user interacts with either *views* (the most common application) or *actionItems*.

It is important to know that *workFlows* use a reference semantic instead of the pass-by-value semantic that is usually found in the R language. This design was chosen to minimize memory consumption and to keep the framework as flexible as possible. The only main consequence on the user-level is the fact that direct assignments to a *workFlow* object are usually not necessary; i.e., functions that operate on the *workFlow* have the potential side-effect of modifying the object.

6.2 Naming schemes

One major obstacle of using scripts as a work flow representation is the need to keep track of object symbols or names of list items. Choosing arbitrary names makes it hard to access objects and there is no way to easily query the structure of the work flow other than reading through the complete script. It is also very hard to appreciate the hierarchical structure of the work flow from a linear listing of R commands. Most objects in flowCore provide useful names or identifiers, and the work flow framework tries to utilize these names as much as possible. For example, a *rectangleGate* called "Lymphocytes" will create two subpopulations, namely the cells within the gate and their complement. Intuitive names for these subpopulation would therefore be "Lymphocyte+" and "Lymphocyte-". These names don't necessarily need to be unique, and in order to be able to unambiguously address each item in a work flow we need additional unique identifiers. Our software keeps an alias table that maps these internal unique identifiers to more human-readable and user friendly names, and unless there are naming collisions, those names will also be the primary identifiers that are exposed in the API. Only if and alias is not unique, objects need to be addressed by their internal unique identifier.

6.3 Creating *workFlow* objects

Objects of class *workFlow* have to be created using the constructor `workFlow`. The single mandatory argument is a flow data object, either a *flowFrame* or a *flowSet*.

```
> data(GvHD)
> wf <- workFlow(GvHD[1:5], name = "myWorkflow")
> wf

A flow cytometry workflow called 'myWorkflow'
The following data views are provided:

 Basic view 'base view'
 on a flowSet
 not associated to a particular action item
```

Printing the objects shows its structure: the work flow contains a single *view* "base view" which essentially is the *flowSet* `fs`. Since the data was added by the constructor and is not the result of a particular data analysis operation, no *actionItem* is associated to the *view*. We can query the available views in the *workFlow* using the `views` method:

```
> views(wf)

[1] "base view"
```

6.4 Adding *actionItems*

Currently there are four different types of *actionItems* that can be added to a *workFlow*:

- *compensateActionItem*: A *compensation* object
- *transformActionItem*: A *transformList* object
- *normalizeActionItem*: A *normalization* object
- *gateActionItem*: A *filter* object

The user doesn't have to bother with the details of these classes, they will be created internally when adding one of the above source objects to the *workFlow* using the `add` method.

```
> tf <- transformList(colnames(GvHD[[1]])[3:6], asinh, transformationId = "asinh")
> add(wf, tf)
> wf

A flow cytometry workflow called 'myWorkflow'
The following data views are provided:

 Basic view 'base view'
 on a flowSet
```

```
 not associated to a particular action item

        View 'asinh'
         on a flowSet linked to
         transform action item 'action_asinh'
```

```
> views(wf)

[1] "base view" "asinh"
```

There are several things to note here. First, we didn't assign the return value of `add` back to `wf`. This is the aforementioned feature of the reference semantic. Internally, *workFlows* are to the most part *environments*, and all modifications change the content of these *environments* rather than the object itself. We also used the `add` methods with two arguments only: the *workFlow* and the *transformList*. In a hierarchical work flow we usually want to control to which subset (or rather to which *view*, to stick to the work flow lingo) we want to add the *actionItem* to. The default is to use the base view, and alternative parent *views* have to be specified using the `parent` argument; `add(wf,tf,parent="base view")` would have been equivalent. Adding the transformation created a new *view* called "asinh" and we can see that this name was automatically taken from the *transformationList* object. If we want to control the name of the new *view*, we can do so using the `name` argument:

```
> add(wf, tf, name = "another asinh transform")
> wf

A flow cytometry workflow called 'myWorkflow'
The following data views are provided:

 Basic view 'base view'
 on a flowSet
 not associated to a particular action item

        View 'asinh'
         on a flowSet linked to
         transform action item 'action_asinh'
         (ID=transActionRef_n1yB93PeUc)

        View 'another asinh transform'
         on a flowSet linked to
         transform action item 'action_asinh'
         (ID=transActionRef_jRGAkXsKZG)
```

This operation created a warning since we didn't use a leaf node to add the transformation. The software can't be sure if any of the downstream leafs in the work flow hierarchy are affected by the transformation (as it would certainly be true for most gates). One solution would be to update all downstream leafs in the work flow, but this feature is not supported yet. Also

note that we used the same *transformList* object twice. While we controlled the name that was used for the second *view*, the alias for the *actionItem* is the same in both cases, and we would have to use the internal unique identifier to unambiguously access the respective object.

Transformation operations (and also compensations or normalizations) only create single *views*. This is not true for gating operations. Even the most simple gate creates two populations: those cells in the gate and the complement. Accordingly, two separate *views* have to be created as well.

```
> rg <- rectangleGate(`FSC-H` = c(200, 400), `SSC-H` = c(250, 400),
+     filterId = "rectangle")
> add(wf, rg, parent = "asinh")
> wf
```

```
A flow cytometry workflow called 'myWorkflow'
The following data views are provided:

 Basic view 'base view'
 on a flowSet
 not associated to a particular action item

        View 'asinh'
         on a flowSet linked to
         transform action item 'action_asinh'
         (ID=transActionRef_n1yB93PeUc)

             View 'rectangle+'
              on a flowSet linked to
              gate action item 'action_rectangle'

             View 'rectangle-'
              on a flowSet linked to
              gate action item 'action_rectangle'

        View 'another asinh transform'
         on a flowSet linked to
         transform action item 'action_asinh'
         (ID=transActionRef_jRGAkXsKZG)
```

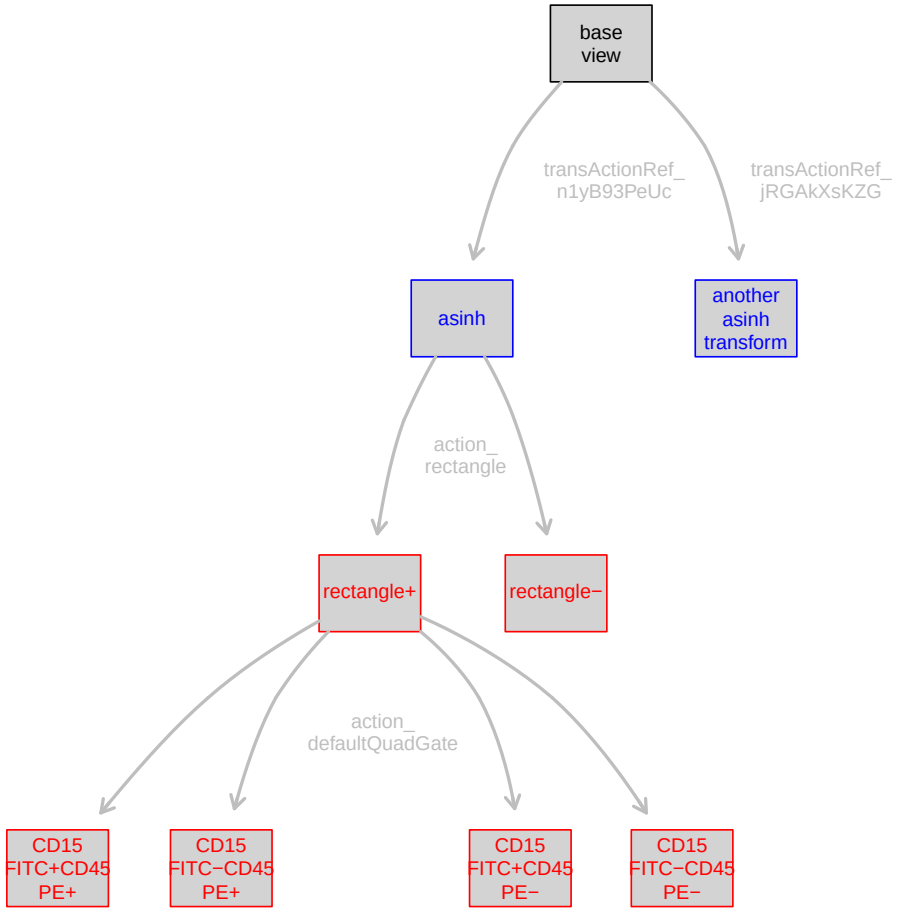
As we see, there are now two new *views* under the "asinh" node called "rectangle+" and "rectangle-". Other filter types may create more than two populations, for instance a *quadrantGate* always results in four sub-populations. There is no restriction on the number of populations that are allowed to be created by a single gating operation, however, when gating a *flowSet*, the number of populations have to be the same for each frame in the set.

```
> qg <- quadGate(`FL1-H` = 2, `FL2-H` = 4)
> add(wf, qg, parent = "rectangle+")
> wf
```

```
A flow cytometry workflow called 'myWorkflow'
The following data views are provided:

 Basic view 'base view'
 on a flowSet
 not associated to a particular action item

        View 'asinh'
         on a flowSet linked to
         transform action item 'action_asinh'
         (ID=transActionRef_n1yB93PeUc)

             View 'rectangle+'
              on a flowSet linked to
              gate action item 'action_rectangle'

                  View 'CD15 FITC+CD45 PE+'
                   on a flowSet linked to
                   gate action item 'action_defaultQuadGate'

                  View 'CD15 FITC-CD45 PE+'
                   on a flowSet linked to
                   gate action item 'action_defaultQuadGate'

                  View 'CD15 FITC+CD45 PE-'
                   on a flowSet linked to
                   gate action item 'action_defaultQuadGate'

                  View 'CD15 FITC-CD45 PE-'
                   on a flowSet linked to
                   gate action item 'action_defaultQuadGate'

             View 'rectangle-'
              on a flowSet linked to
              gate action item 'action_rectangle'

        View 'another asinh transform'
         on a flowSet linked to
         transform action item 'action_asinh'
         (ID=transActionRef_jRGAkXsKZG)
```

For complex work flows it becomes increasingly hard to convey all information by printing on the screen. There is a plotting function for *workFlow* objects which plots the underlying tree. The function depends on the Rgraphviz package.

```
> plot(wf)
```



6.5 Accessing items in the *workFlow* object

The main reason for having the *workFlow* class is to be able to easily access all elements of a potentially complex analysis work flow. The predominant use case here is to access individual *views*, either for plotting, to create summary statistics or to use the underlying data for subsequent analysis steps. You can do that using the familiar list subsetting syntax:

```
> wf[["rectangle+"]]

View 'rectangle+'
 on a flowSet linked to
 gate action item 'action_rectangle'
 applied to view 'asinh' (ID=transViewRef_H7OjixzVUE)

> wf$asinh

View 'asinh'
 on a flowSet linked to
 transform action item 'action_asinh'
```

```
 (ID=transActionRef_n1yB93PeUc)
 applied to view 'base view' (ID=viewRef_XakADLP44T)
```

This also works for *actionItems*:

```
> wf[["action_rectangle"]]

gate action item 'action_rectangle'
 applied to view 'asinh' (ID=transViewRef_H7OjixzVUE)
```

In order to retreive the underlying data for a *view* you can use the `Data` method.

```
> Data(wf[["rectangle-"]])

A flowSet with 5 experiments.

An object of class "AnnotatedDataFrame"
  rowNames: s5a01 s5a02 ... s5a05 (5 total)
  varLabels: Patient Visit ... name (5 total)
  varMetadata: labelDescription

  column names:
  FSC-H SSC-H FL1-H FL2-H FL3-H FL2-A FL4-H Time
```

There are `summary` methods defined for the different *view* subclasses, which basically create the respective summaries of the associated flowCore objects, e.g. a *filterResult* for a filtering operation.

```
> summary(wf[["action_rectangle"]])

   sample population    percent count  true false           p         q
1   s5a01 rectangle+  2.8070175  3420    96  3324 0.028070175 0.9719298
2   s5a02 rectangle+  1.5859031  3405    54  3351 0.015859031 0.9841410
3   s5a03 rectangle+  0.3202329  3435    11  3424 0.003202329 0.9967977
4   s5a04 rectangle+  0.3859649  8550    33  8517 0.003859649 0.9961404
5   s5a05 rectangle+  3.5542747 10410   370 10040 0.035542747 0.9644573
6   s5a01 rectangle- 97.1929825  3420  3324    96 0.028070175 0.9719298
7   s5a02 rectangle- 98.4140969  3405  3351    54 0.015859031 0.9841410
8   s5a03 rectangle- 99.6797671  3435  3424    11 0.003202329 0.9967977
9   s5a04 rectangle- 99.6140351  8550  8517    33 0.003859649 0.9961404
10  s5a05 rectangle- 96.4457253 10410 10040   370 0.035542747 0.9644573

> summary(wf[["CD15 FITC+CD45 PE+"]])

   sample         population  percent count true false         p          q
1   s5a01 CD15 FITC+CD45 PE+ 91.66667    96   88     8 0.9166667 0.08333333
5   s5a02 CD15 FITC+CD45 PE+ 85.18519    54   46     8 0.8518519 0.14814815
9   s5a03 CD15 FITC+CD45 PE+ 63.63636    11    7     4 0.6363636 0.36363636
13  s5a04 CD15 FITC+CD45 PE+ 93.93939    33   31     2 0.9393939 0.06060606
17  s5a05 CD15 FITC+CD45 PE+ 98.91892   370  366     4 0.9891892 0.01081081
```

The flowViz package also defines `xyplot` and `densityplot` methods for *view* objects. Sensible defaults are usually chosen automatically, for instance to add the gate boundaries to the plot of a *gateView*.

```
> densityplot(wf[["base view"]])
> xyplot(`FL1-H` ~ `FL2-H`, wf[["CD15 FITC+CD45 PE+"]])
```

6.6 Removing items from *workFlow* object

The hierarchical structur of the *workFlow* object introduces dependencies between *views* or between *views* and *actionItems*. Thus, removing a particular view means also removing all of its associated child *views* and *actionItems*. The easiest way to remove objects from a *workFlow* is through the `undo` mechanism. The function takes a *workFlow* object as first argument and an optional second integer argument giving the number of operartions that are supposed to be roled back.

```
> undo(wf)
> wf

A flow cytometry workflow called 'myWorkflow'
The following data views are provided:

 Basic view 'base view'
 on a flowSet
 not associated to a particular action item

        View 'asinh'
         on a flowSet linked to
         transform action item 'action_asinh'
         (ID=transActionRef_n1yB93PeUc)

             View 'rectangle+'
              on a flowSet linked to
              gate action item 'action_rectangle'

             View 'rectangle-'
              on a flowSet linked to
              gate action item 'action_rectangle'

        View 'another asinh transform'
         on a flowSet linked to
         transform action item 'action_asinh'
         (ID=transActionRef_jRGAkXsKZG)
```

Sometimes it is more convenient to specify a particular *view* or *action* item that is to be removed. This can be archived using the `Rm` methods for *views* or *actionItems*. As a side-effect

these methods will also remove all dependent items and the user should be well aware of this potentially dangerous behaviour.

```
> Rm("rectangle-", wf)
```

```
> Rm("asinh", wf)
```

References

C. Bruce Bagwell. DNA histogram analysis for node-negative breast cancer. *Cytometry A*, 58: 76–78, 2004.

Mark S Boguski and Martin W McIntosh. Biomedical informatics for proteomics. *Nature*, 422: 233–237, 2003.

Raul C Braylan. Impact of flow cytometry on the diagnosis and characterization of lymphomas, chronic lymphoproliferative disorders and plasma cell neoplasias. *Cytometry A*, 58:57–61, 2004.

A. Brazma. On the importance of standardisation in life sciences. *Bioinformatics*, 17:113–114, 2001.

M. Chicurel. Bioinformatics: bringing it all together. *Nature*, 419:751–755, 2002.

Maura Gasparetto, Tracy Gentry, Said Sebti, Erica O'Bryan, Ramadevi Nimmanapalli, Michelle A Blaskovich, Kapil Bhalla, David Rizzieri, Perry Haaland, Jack Dunne, and Clay Smith. Identification of compounds that enhance the anti-lymphoma activity of rituximab using flow cytometric high-content screening. *J Immunol Methods*, 292:59–71, 2004.

M. Keeney, D. Barnett, and J. W. Gratama. Impact of standardization on clinical cell analysis by flow cytometry. *J Biol Regul Homeost Agents*, 18:305–312, 2004.

N. LeMeur and F. Hahne. Analyzing flow cytometry data with bioconductor. *Rnews*, 6:27–32, 2006.

DR Parks. *Data Processing and Analysis: Data Management.*, volume 1 of *Current Protocols in Cytometry*. John Wiley & Sons, Inc, New York, 1997.

J. Spidlen, R.C. Gentleman, P.D. Haaland, M. Langille, N. Le Meur N, M.F. Ochs, C. Schmitt, C.A. Smith, A.S. Treister, and R.R. Brinkman. Data standards for flow cytometry. *OMICS*, 10(2):209–214, 2006.

J. Spidlen, R.C. Leif, W. Moore, M. Roederer, International Society for the Advancement of Cytometry Data Standards Task Force, and R.R. Brinkman. Gating-ml: Xml-based gating descriptions in flow cytometry. *Cytometry A*, 73A(12):1151–1157, 2008.

Maria A Suni, Holli S Dunn, Patricia L Orr, Rian de Laat, Elizabeth Sinclair, Smita A Ghanekar, Barry M Bredt, John F Dunne, Vernon C Maino, and Holden T Maecker. Performance of plate-based cytokine flow cytometry with automated data analysis. *BMC Immunol*, 4:9, 2003.