flowCore: data structures package for flow cytometry data

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

May 5, 2007

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 the base for the development of many tools for the analysis of high throughput flowcytometry.

keywords Flow cytometry, high throughtput, 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 multiparameter 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 wich 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 1.0 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 "standardize" 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      Time
Min.    0.08309 0.01075 0.0000 0.0000 0.00000 0.00000 0.0000 0.0009775
1st Qu. 0.37630 0.13780 0.2278 0.2708 0.08798 0.00000 0.2053 0.1193000
Median  0.43110 0.18480 0.5332 0.3382 0.18870 0.02542 0.2727 0.2815000
Mean    0.48090 0.27170 0.4292 0.3579 0.17570 0.03331 0.3163 0.2881000
3rd Qu. 0.50640 0.26390 0.5963 0.4272 0.25810 0.04985 0.3812 0.4472000
Max.    1.00000 1.00000 0.8915 1.0000 0.87980 1.00000 0.9990 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 with 10000 cells and 8 observables:
<FSC.H> FSC-H <SSC.H> SSC-H <FL1.H> <FL2.H> <FL3.H> <FL1.A> <FL4.H> <Time> Time (51.20 sec.)
slot 'description' has 147 elements
```

Finally, 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 the Height parameters:

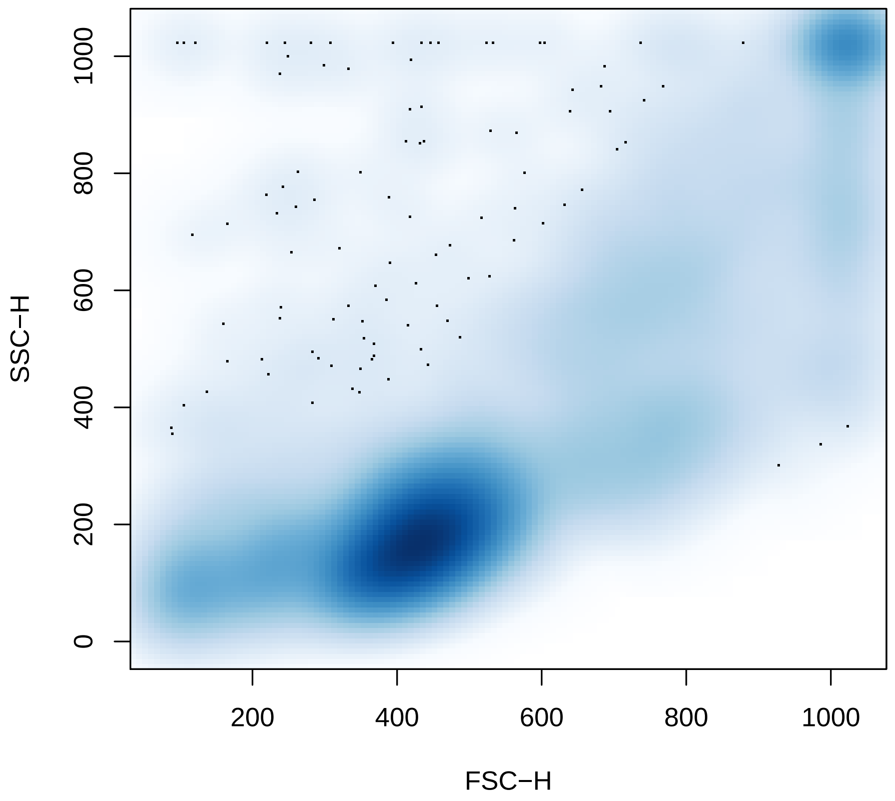
```
> x <- read.FCS(file.name, column.pattern = "-H")
> x

flowFrame object with 10000 cells and 6 observables:
<FSC-H> FSC-H <SSC-H> SSC-H <FL1-H> <FL2-H> <FL3-H> <FL4-H>
slot 'description' has 147 elements
```

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

2.1.3 Visualizing a *flowFrame*

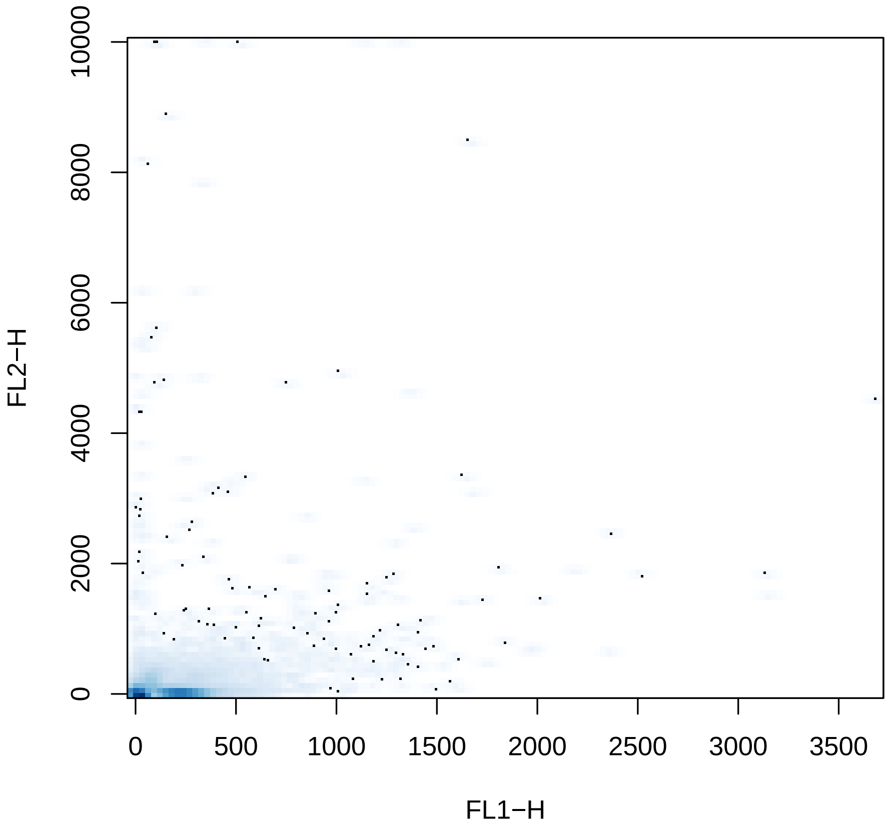
Much of the more sophisticated visualization of *flowFrame* and *flowSet* objects, including an interface to the lattice graphics system is implemented by the flowViz package, also included as part of Bioconductor. However, flowCore does implement some basic plotting facilities using the standard `plot` function. The basic plot provides a simple bivariate density plot of the first two parameters:

```
> plot(x)
```



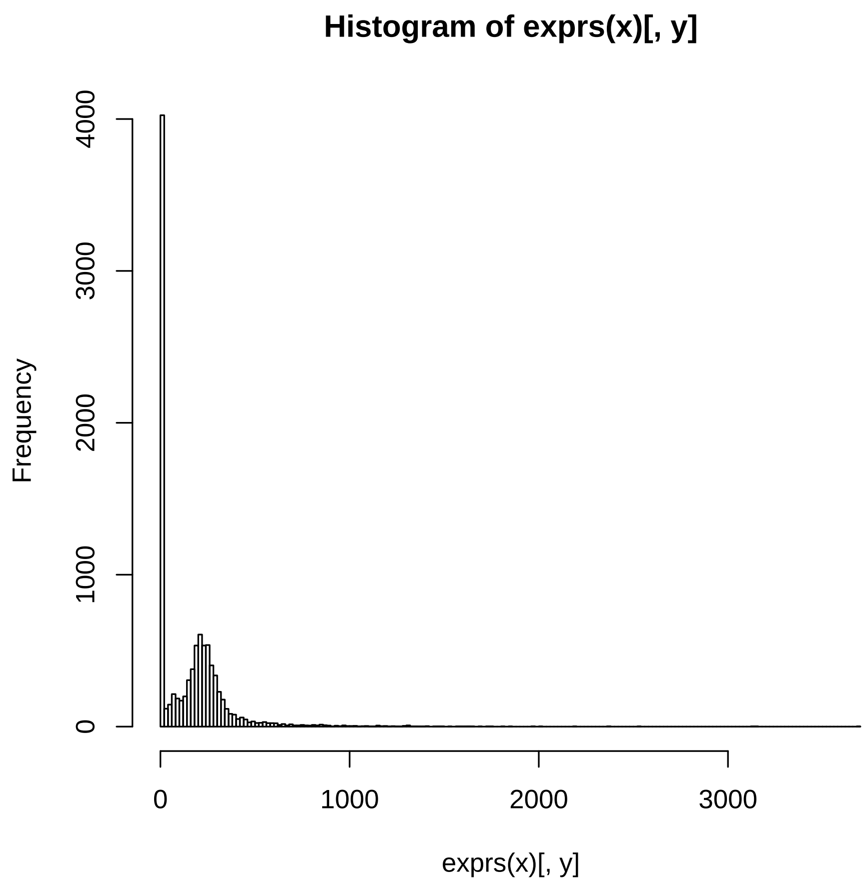
To control the parameters being plotted we can supply a y value in the form of a character vector:

```
> 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.

  rowNames: V1, V2, ..., V5 (5 total)
  varLabels and varMetadata:
    name: Name

  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.

  rowNames: 7AAD, apc, ..., pe (5 total)
  varLabels and varMetadata:
    name: Name

  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
7AAD 7AAD 060909.005
apc   apc 060909.004
fitc fitc 060909.002
NULL NULL 060909.001
pe     pe 060909.003
```

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.

  rowNames: 060909.001, 060909.002, ..., 060909.005 (5 total)
  varLabels and varMetadata:
    name: Name

  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.

  rowNames: 7AAD, apc, ..., pe (5 total)
  varLabels and varMetadata:
    name: NA
    Filename: NA
  varMetadata: labelDescriptions, labelDescription

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

```
> pData(phenoData(fs))
```

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

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 he `[` extraction operator returns a new *flowSet* by **copying** the environment. However, simply assigning the *flowFrame* to a new variable will **not** copying 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
7AAD   429   133   5.010744  15.029018  63.466061     0  20.970227
apc    441   129   4.377753   4.877217   4.790181     0 360.732067
fitc   436   128 936.811048 229.975372  33.490890   217   8.295949
NULL   423   128   4.110368   4.538282   3.656368     0   7.247948
pe     438   120  10.204639 796.655892 114.975700     0   9.326033
```

which is equivalent to the less readable

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

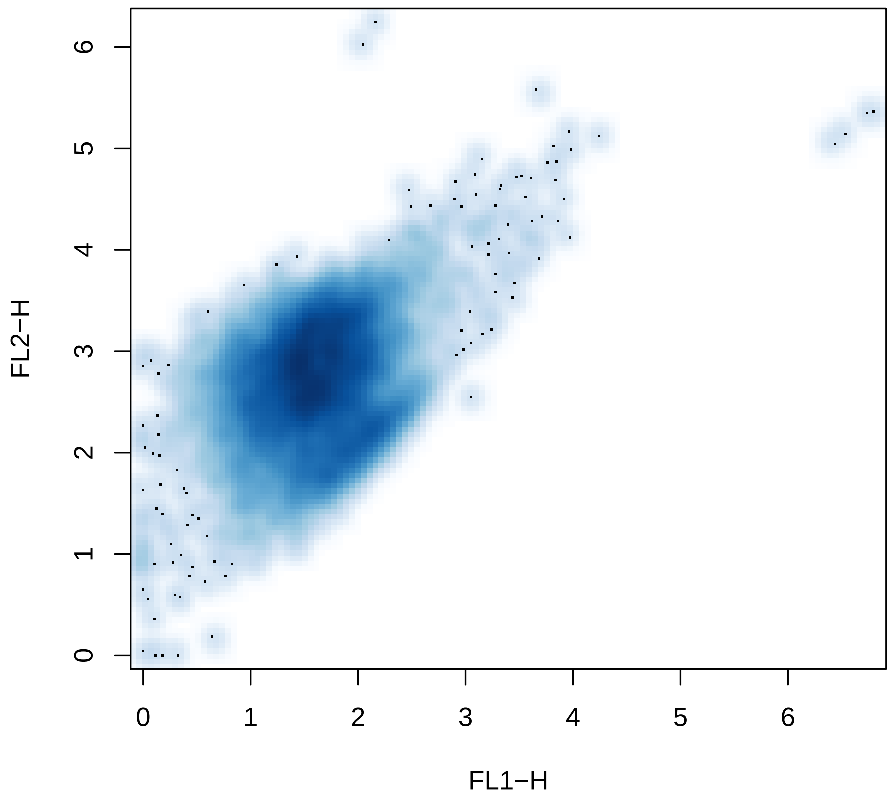
```
     FSC-H SSC-H      FL1-H      FL2-H      FL3-H FL1-A      FL4-H
7AAD   429   133   5.010744  15.029018  63.466061     0  20.970227
apc    441   129   4.377753   4.877217   4.790181     0 360.732067
fitc   436   128 936.811048 229.975372  33.490890   217   8.295949
NULL   423   128   4.110368   4.538282   3.656368     0   7.247948
pe     438   120  10.204639 796.655892 114.975700     0   9.326033
```

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

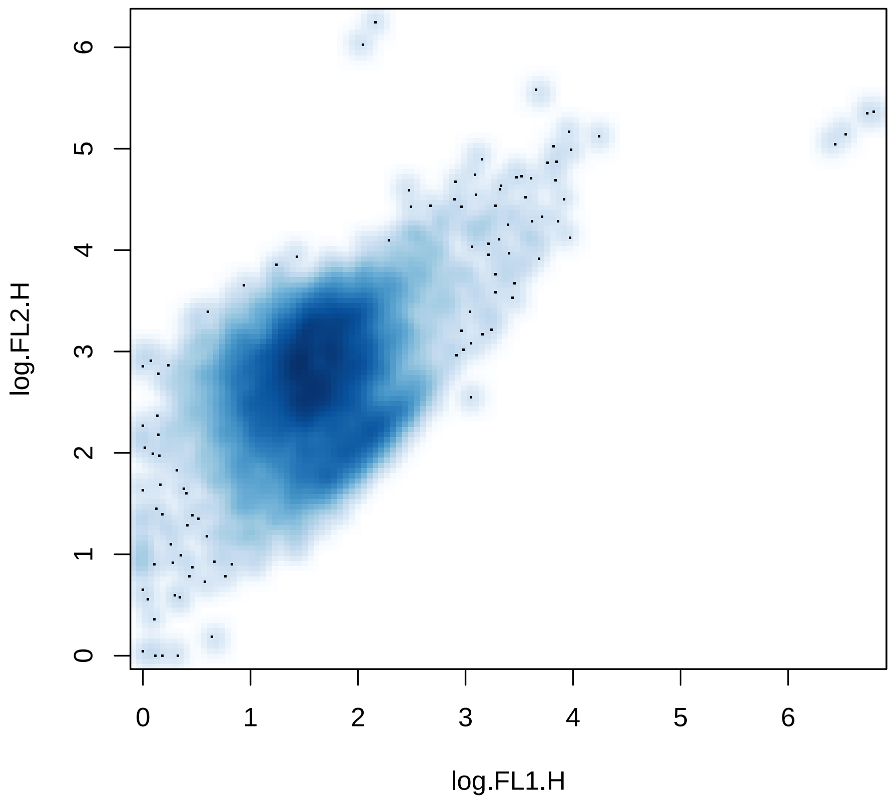
flowCore features two methods of transforming parameters within a *flowFrame*: inline and out-of-line. The inline method, discussed in the next section has been developed primarily to support filtering features and is strictly more limited than the out-of-line transformation method, which uses R's **transform**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

An object of class "transform"
function (x)
{
    x[x < a] <- a
    x
}
<environment: 0x3a46588>
Slot "transformationId":
character(0)
```

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.

  rowNames: 7AAD, apc, ..., pe (5 total)
  varLabels and varMetadata:
    name: NA
    Filename: NA
  varMetadata: labelDescriptions, labelDescription

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

4 Filtering

The most common task in the analysis of flow cytometry data is usual some form of filtering operation, 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.

4.1 Standard Filters

Most filtering operations are a composition of one or more common filtering operations. Like transformations, flowCore includes a number of built-in common flow cytometry filters.

The simplest of these filters are the geometric filters, 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(50,
+     100), "FL2-H" = c(50, 100))
```

We also include some data-driven filters not usually found in flow cytometry software:

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 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: 2 of 10000 (0.02%)
```

```
> summary(result)$n
```

```
[1] 10000
```

```
> summary(result)$true
```

```
[1] 2
```

```
> summary(result)$p
```

```
[1] 2e-04
```

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

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

```
Low: 2552 of 10000 (25.52%)
Medium: 5188 of 10000 (51.88%)
High: 2260 of 10000 (22.60%)
```

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

```
> filter(fs, rectGate)
```

```
$`7AAD`
A filter named 'Fluorescence Region'

$apc
A filter named 'Fluorescence Region'

$fitc
A filter named 'Fluorescence Region'

$`NULL`
A filter named 'Fluorescence Region'

$pe
A 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(filterId = "MorphologyGate", "FSC-H",
+     "SSC-H", scale = 2)
> smaller <- Subset(fs, morphGate)
> fs[[1]]

flowFrame object with 10000 cells and 7 observables:
<FSC-H> Forward Scatter <SSC-H> Side Scatter <FL1-H> <FL2-H> <FL3-H> 7AAD <FL1-A> <FL4-H>
slot 'description' has 125 elements

> smaller[[1]]

flowFrame object with 8406 cells and 7 observables:
<FSC-H> Forward Scatter <SSC-H> Side Scatter <FL1-H> <FL2-H> <FL3-H> 7AAD <FL1-A> <FL4-H>
slot 'description' has 125 elements
```

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("myKMeans", "FSC-H" = c("Low",
+     "Medium", "High")))

$Low
flowFrame object with 2375 cells and 7 observables:
<FSC-H> Forward Scatter <SSC-H> Side Scatter <FL1-H> <FL2-H> <FL3-H> 7AAD <FL1-A> <FL4-H>
slot 'description' has 125 elements

$Medium
flowFrame object with 3660 cells and 7 observables:
<FSC-H> Forward Scatter <SSC-H> Side Scatter <FL1-H> <FL2-H> <FL3-H> 7AAD <FL1-A> <FL4-H>
slot 'description' has 125 elements

$High
flowFrame object with 2371 cells and 7 observables:
<FSC-H> Forward Scatter <SSC-H> Side Scatter <FL1-H> <FL2-H> <FL3-H> 7AAD <FL1-A> <FL4-H>
slot 'description' has 125 elements
```

or for an entire *flowSet*

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

$`7AAD`
$`7AAD`$`Low in 7AAD`
flowFrame object with 2375 cells and 7 observables:
<FSC-H> Forward Scatter <SSC-H> Side Scatter <FL1-H> <FL2-H> <FL3-H> 7AAD <FL1-A> <FL4-H>
slot 'description' has 125 elements

$`7AAD`$`Medium in 7AAD`
flowFrame object with 3660 cells and 7 observables:
<FSC-H> Forward Scatter <SSC-H> Side Scatter <FL1-H> <FL2-H> <FL3-H> 7AAD <FL1-A> <FL4-H>
slot 'description' has 125 elements

$`7AAD`$`High in 7AAD`
flowFrame object with 2371 cells and 7 observables:
<FSC-H> Forward Scatter <SSC-H> Side Scatter <FL1-H> <FL2-H> <FL3-H> 7AAD <FL1-A> <FL4-H>
slot 'description' has 125 elements


$apc
$apc$`Low in apc`
flowFrame object with 2319 cells and 7 observables:
<FSC-H> Forward Scatter <SSC-H> Side Scatter <FL1-H> <FL2-H> <FL3-H> <FL1-A> <FL4-H> APC
slot 'description' has 129 elements

$apc$`Medium in apc`
flowFrame object with 3600 cells and 7 observables:
<FSC-H> Forward Scatter <SSC-H> Side Scatter <FL1-H> <FL2-H> <FL3-H> <FL1-A> <FL4-H> APC
slot 'description' has 129 elements

$apc$`High in apc`
flowFrame object with 2325 cells and 7 observables:
<FSC-H> Forward Scatter <SSC-H> Side Scatter <FL1-H> <FL2-H> <FL3-H> <FL1-A> <FL4-H> APC
slot 'description' has 129 elements


$fitc
$fitc$`Low in fitc`
flowFrame object with 2031 cells and 7 observables:
<FSC-H> Forward Scatter <SSC-H> Side Scatter <FL1-H> FITC <FL2-H> <FL3-H> <FL1-A> <FL4-H>
slot 'description' has 129 elements

$fitc$`Medium in fitc`
flowFrame object with 3220 cells and 7 observables:
```

```
<FSC-H> Forward Scatter <SSC-H> Side Scatter <FL1-H> FITC <FL2-H> <FL3-H> <FL1-A> <FL4-H>
slot 'description' has 129 elements

$fitc$`High in fitc`
flowFrame object with 2038 cells and 7 observables:
<FSC-H> Forward Scatter <SSC-H> Side Scatter <FL1-H> FITC <FL2-H> <FL3-H> <FL1-A> <FL4-H>
slot 'description' has 129 elements


$`NULL`
$`NULL`$`Low in NULL`
flowFrame object with 2420 cells and 7 observables:
<FSC-H> FSC-Height <SSC-H> SSC-Height <FL1-H> <FL2-H> <FL3-H> <FL1-A> <FL4-H>
slot 'description' has 123 elements

$`NULL`$`Medium in NULL`
flowFrame object with 3563 cells and 7 observables:
<FSC-H> FSC-Height <SSC-H> SSC-Height <FL1-H> <FL2-H> <FL3-H> <FL1-A> <FL4-H>
slot 'description' has 123 elements

$`NULL`$`High in NULL`
flowFrame object with 2331 cells and 7 observables:
<FSC-H> FSC-Height <SSC-H> SSC-Height <FL1-H> <FL2-H> <FL3-H> <FL1-A> <FL4-H>
slot 'description' has 123 elements


$pe
$pe$`Low in pe`
flowFrame object with 1813 cells and 7 observables:
<FSC-H> Forward Scatter <SSC-H> Side Scatter <FL1-H> <FL2-H> PE <FL3-H> <FL1-A> <FL4-H>
slot 'description' has 129 elements

$pe$`Medium in pe`
flowFrame object with 2734 cells and 7 observables:
<FSC-H> Forward Scatter <SSC-H> Side Scatter <FL1-H> <FL2-H> PE <FL3-H> <FL1-A> <FL4-H>
slot 'description' has 129 elements

$pe$`High in pe`
flowFrame object with 1627 cells and 7 observables:
<FSC-H> Forward Scatter <SSC-H> Side Scatter <FL1-H> <FL2-H> PE <FL3-H> <FL1-A> <FL4-H>
slot 'description' has 129 elements
```

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

A filter named 'Fluorescence Region and MorphologyGate'

> rectGate | morphGate

A filter named 'Fluorescence Region or MorphologyGate'

> !morphGate

A filter named 'not MorphologyGate'
```

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: 0 of 8406 (0.00%)
```

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]]
An object of class "transformMap"
Slot "output":
[1] "FL1-H"

Slot "input":
[1] "FL1-H"
```

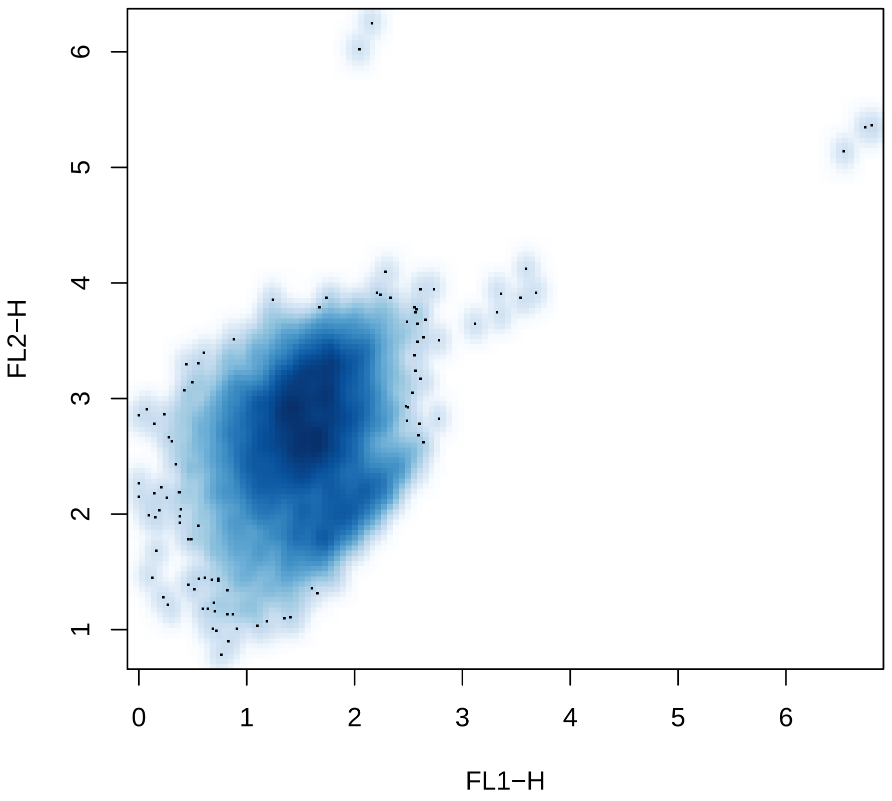
```
Slot "f":
function (x, base = exp(1))
if (missing(base)) .Internal(log(x)) else .Internal(log(x, base))
<environment: namespace:base>


[[2]]
An object of class "transformMap"
Slot "output":
[1] "FL2-H"

Slot "input":
[1] "FL2-H"

Slot "f":
function (x, base = exp(1))
if (missing(base)) .Internal(log(x)) else .Internal(log(x, base))
<environment: namespace:base>
```

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

A filter named '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.

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.

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.