BioC 2010 Automated Gating and Metaclustering for Flow Cytometry Data

Greg Finak, Raphael Gottardo

July 29, 2010

Purpose of this Document

This document describes how to use the *flowClust* and *flowMerge* BioConductor packages to analyze flow cytometry data. It provides examples of running *flowClust* and *flowMerge* in a fully automated and semi–automated manner, and describes how to run *flowMerge* and *flowClust* in a parallel computing environment using the *snow* and *snowfall* package, as well as a single processor setup. The document provdes worked examples that demonstrate the effects of different*flowClust* parameters on flow cytometry gating results. The document also covers downstream analysis of *flowClust* and *flowMerge* gated data for *metaclustering* of gated cell populations across multiple samples. *Metaclustering* is the matching of corresponding cell populations across replicated or similar samples.

Hardware and Software Requirements

The analysis described here was run on an Intel–based Macbook pro running Mac OS X 10.6.4, with a 2.93 GHz core 2 duo processor and 8GB of RAM. It was run on BioConductor version 2.7, R 2.12.0, flowMerge 1.3.7 (development), flowClust 2.7.0 (development), and flowCore 1.15.1. The development versions of these packages are available at http://bioconductor.org/checkResults/2.7/bioc-LATEST/. The metaclustering examples presented at the end of the document require the *flowMetaClust* packgage, which is currently in development and not yet available.

Goals

After working through the examples in this document, you should be able to:

- Run `flowClust` and `flowMerge` in a fully automated manner
- Run `flowClust` and `flowMerge` in a semi-automated manner

- Run high throughput `flowClust` model fitting in parallel using `snow` and `snowfall` packages
- Understand the role of different model parameters and how changing them impacts gated cell pouplations
- Be able to manipulate and evaluate `flowClust` and `flowMerge` model objects in R
- Perform metaclustering of cell populations across samples using `flowMeta-Cluster`

Overview

Analyzing flow cytometry data within R and BioConductor can be quite challenging for new users of the BioConductor flow cytometry tools. This is particularly true little prior experience programming or using R and BioConductor. Manual analysis of flow data is inherently suited to a graphical user interface, so R's and BioConductor's command line environment can be counter–intuitive.

However, the size and complexity of flow cytometry data sets is growing, and manual gating approaches can be inefficient and error-prone for such data. The tools available through BioConductor are extremely powerful and flexible, allowing automated and semi-automated analysis of many samples in a high-throughput manner. Many of the `flow` packages include flexible functionality for visualizing flow cytometry data in a multitude of different ways. Many of these will have been covered during the other practical sessions.

Data Set

We will use the same data set you have encountered in the previous tutorials on FCM data analysis. This is a data set comparing two treatment protocols (Drug A vs Drug B). We will be gating and measuring the fraction of T–helper cells and cytotoxic T cells that express HLADr activation marker following different treatments. We will work through the gating steps using different sets of parameters, and we will explore different modeling assumptions, comparing the results. Some of the data analysis may be time consuming if run on a laptop, so we have included pre-analyzed data that can be loaded using `data()`. We note these points in the document, and the data are included in the `data` subdirectory in the supplied package.

Loading flow libraries

BioConductor is modular, with different functionality coming from different user–submitted packages. In order to utilize the functionality provided by a package, that package must first be loaded into R. Therefore, in order to analyze

flow cytometry data in BioConductor, we need to load the appropriate packages and libraries.

```
require(flowMerge)
require(flowViz)
require(flowQ)
require(flowStats)
require(snowfall)
require(fpc)
require(flowTrack)
```

The `require()` function tells R that the package provided as an argument must be loaded in order to run the code which follows. R will therefore check whether the package is loaded, and if not, will load the packge. If the package is not installed or does not load correctly, the function returns `FALSE` which can be tested inside other functions. Similar behaviour can be obtained with the `library()` function, except that `library` will quit if the library is not installed or fails to load correctly.

Loading the Data

The raw flow data are is located individual files in FCS (flow cytometry data standard) format, with one sample per file. The data are stored in the `flowdata` subdirectory. Additional annotation information about each sample is in a user–generated text file, `annotation.txt`. BioConductor provides data structures for storing and working with metadata about biological samples. The class `AnnotatedDataFrame`, part of the `Biobase` core BioConductor package, is the object that handles storage and management of biological metadata for a variety of data types, including flow cytometry. More information about the `AnnotatedDataFrame` or any other R command can be found by typing `?` followed by the command name at the R command line (i.e. `?AnnotatedDataFrame`.

To load the data we run:

```
file.loc <- system.file("extdata",
            package = "flowTrack")
data <- read.flowSet(path = file.loc,
            pattern = "fcs", phenoData = "annotation.txt",
            transformation = FALSE)
```

The above command, `read.flowSet` is pretty self–explanatory. It reads in several fcs files into a `flowSet` object. The flowSet is an object that combines related flow cytometry samples. The samples must have the same number of

dimensions and the same labels in order to be combined into a flowSet. Another way to view a flowSet is as a combination of `flowFrames`. A flowFrame is the BioConductor object that represents an individual flow cytometry sample. The `read.flowSet()` function takes a number of parameters. The first is the path, or directory where the fcs data files are located. The second, `phenoData`, is the name of an annotation file containing additiona metadata about each sample, including sample names, file names, and any treatment information associated with each sample. The annotation information is used to internally construct an `AnnotatedDataFrame`, which is the BioConductor data structure that keeps track of sample metadata relating to an experiment. This annotation information is attached to the flowSet. An alternate way to specify the file names is via the `files` argument (omitted above). This should be a character vector of file names. If `files` and `phenoData` arguments are both omitted, all files in `path` will be loaded. The `transformation` argument specifies how the data should be transformed upon loading (see `?read.FCS` for more information).

Once the data are read into a `flowSet`, we can manipulate them. Each element of the flowSet can be extracted using R's list access notation (i.e. `data[[1]]` would extract the first flowFrame from the flowSet).

We can get additional information about the flowSet by entering the name of the flowSet at the R command line:

```
 data

A flowSet with 14 experiments.

An object of class "AnnotatedDataFrame"
  rowNames: 01126211_NB8_I025.fcs, 01247181
  _NB06_I025.fcs, ..., 12015151_NB8_I025.fc
  s  (14 total)
  varLabels and varMetadata description:
    PatientID:
    GroupID:
    ...: ...
    name: Filename
    (6 total)

  column names:
  FSC-A SSC-A FITC-A PE-A FL3-A PE-Cy7-A APC-A Time
```

We see it is composed of 14 sample (experiments), and we see some additional information about the AnnotatedDataFrame associated with the flowSet (we see file names, and that there are 6 pieces of metadata assocaited with each sample).

Similarly, by calling:

```
data[[1]]
```

```
flowFrame object '01126211_NB8_I025.fcs'
with 4000 cells and 8 observables:
        name         desc range minRange
$P1    FSC-A         <NA>  1024        0
$P2    SSC-A         <NA>  1024        0
$P3   FITC-A   CD8 FITC-A  1024        1
$P4     PE-A    CD69 PE-A  1024        1
$P5    FL3-A          CD4  1024        1
$P6 PE-Cy7-A CD3 PE-Cy7-A  1024        1
$P7    APC-A  HLADr APC-A  1024        1
$P8     Time         <NA>  1024        0
    maxRange
$P1     1023
$P2     1023
$P3     1023
$P4     1023
$P5     1023
$P6     1023
$P7     1023
$P8     1023
98 keywords are stored in the 'description' slot
```

we get some additional information about the first flowFrame. We see it has 4000 events and 8 channels. It is named using its associated FCS file name. The channels are named by their associated dye ("names" column). The assocaited markers are in the "desc" column.

If we examine the default sample and marker names associated with the flowSet, we see that they are not very informative:

```
colnames(data)
```

```
[1] "FSC-A"    "SSC-A"    "FITC-A"
[4] "PE-A"     "FL3-A"    "PE-Cy7-A"
[7] "APC-A"    "Time"
```

```
sampleNames(data)
```

```
[1] "01126211_NB8_I025.fcs"
[2] "01247181_NB06_I025.fcs"
```

```
 [3] "02276161_NB8_I025.fcs"
 [4] "03157141_NB06_I025.fcs"
 [5] "05107251_NB06_I025.fcs"
 [6] "06226101_NB8_I025.fcs"
 [7] "07115141_NB8_I025.fcs"
 [8] "07215281_NB8_I025.fcs"
 [9] "08045071_NB8_I025.fcs"
[10] "09225121_NB8_I025.fcs"
[11] "10175181_NB8_I025.fcs"
[12] "10276181_NB8_I025.fcs"
[13] "11145351_NB8_I025.fcs"
[14] "12015151_NB8_I025.fcs"
```

The default sample and channel names appear on any generated plots and graphics. We want to assign the more informative `"PatientID"` from the annotation metadata as sample names:

We can view the metadata via:

```
                      PatientID GroupID
01126211_NB8_I025.fcs     pid349  DRUG A
01247181_NB06_I025.fcs    pid300  DRUG B
02276161_NB8_I025.fcs     pid778  DRUG A
03157141_NB06_I025.fcs    pid291  DRUG B
05107251_NB06_I025.fcs    pid214  DRUG A
06226101_NB8_I025.fcs     pid867  DRUG B
07115141_NB8_I025.fcs     pid877  DRUG B
07215281_NB8_I025.fcs     pid409  DRUG A
08045071_NB8_I025.fcs     pid993  DRUG B
09225121_NB8_I025.fcs     pid847  DRUG A
10175181_NB8_I025.fcs     pid244  DRUG B
10276181_NB8_I025.fcs     pid149  DRUG B
11145351_NB8_I025.fcs     pid225  DRUG A
12015151_NB8_I025.fcs     pid333  DRUG A
                      Technician  Project
01126211_NB8_I025.fcs        Jill BIOC2009
01247181_NB06_I025.fcs       Jill BIOC2009
02276161_NB8_I025.fcs        Jill BIOC2009
03157141_NB06_I025.fcs      Peter BIOC2009
05107251_NB06_I025.fcs      Peter BIOC2009
06226101_NB8_I025.fcs        Jill BIOC2009
07115141_NB8_I025.fcs        Jill BIOC2009
07215281_NB8_I025.fcs        Jill BIOC2009
08045071_NB8_I025.fcs        Jill BIOC2009
09225121_NB8_I025.fcs        Mark BIOC2009
10175181_NB8_I025.fcs        Mark BIOC2009
10276181_NB8_I025.fcs        Mark BIOC2009
```

```
11145351_NB8_I025.fcs        Mark BIOC2009
12015151_NB8_I025.fcs        Mark BIOC2009
                                      Stain
01126211_NB8_I025.fcs  CD8/CD69/CD4/CD3/HLADR
01247181_NB06_I025.fcs CD8/CD69/CD4/CD3/HLADR
02276161_NB8_I025.fcs  CD8/CD69/CD4/CD3/HLADR
03157141_NB06_I025.fcs CD8/CD69/CD4/CD3/HLADR
05107251_NB06_I025.fcs CD8/CD69/CD4/CD3/HLADR
06226101_NB8_I025.fcs  CD8/CD69/CD4/CD3/HLADR
07115141_NB8_I025.fcs  CD8/CD69/CD4/CD3/HLADR
07215281_NB8_I025.fcs  CD8/CD69/CD4/CD3/HLADR
08045071_NB8_I025.fcs  CD8/CD69/CD4/CD3/HLADR
09225121_NB8_I025.fcs  CD8/CD69/CD4/CD3/HLADR
10175181_NB8_I025.fcs  CD8/CD69/CD4/CD3/HLADR
10276181_NB8_I025.fcs  CD8/CD69/CD4/CD3/HLADR
11145351_NB8_I025.fcs  CD8/CD69/CD4/CD3/HLADR
12015151_NB8_I025.fcs  CD8/CD69/CD4/CD3/HLADR
                                       name
01126211_NB8_I025.fcs   01126211_NB8_I025.fcs
01247181_NB06_I025.fcs 01247181_NB06_I025.fcs
02276161_NB8_I025.fcs   02276161_NB8_I025.fcs
03157141_NB06_I025.fcs 03157141_NB06_I025.fcs
05107251_NB06_I025.fcs 05107251_NB06_I025.fcs
06226101_NB8_I025.fcs   06226101_NB8_I025.fcs
07115141_NB8_I025.fcs   07115141_NB8_I025.fcs
07215281_NB8_I025.fcs   07215281_NB8_I025.fcs
08045071_NB8_I025.fcs   08045071_NB8_I025.fcs
09225121_NB8_I025.fcs   09225121_NB8_I025.fcs
10175181_NB8_I025.fcs   10175181_NB8_I025.fcs
10276181_NB8_I025.fcs   10276181_NB8_I025.fcs
11145351_NB8_I025.fcs   11145351_NB8_I025.fcs
12015151_NB8_I025.fcs   12015151_NB8_I025.fcs
```

Renaming Samples

The following code assigns the `"PatientID"` columns as the sample names:

```
sampleNames(data) <- as.character(pData(data)[,
          "PatientID"])
```

Renaming Channels

In order to use the more informative marker names as the channel names, rather than the names of the dyes, we run the following:

```
for (i in seq_len(length(data))) {
          pData(parameters(data[[i]]))[,
```

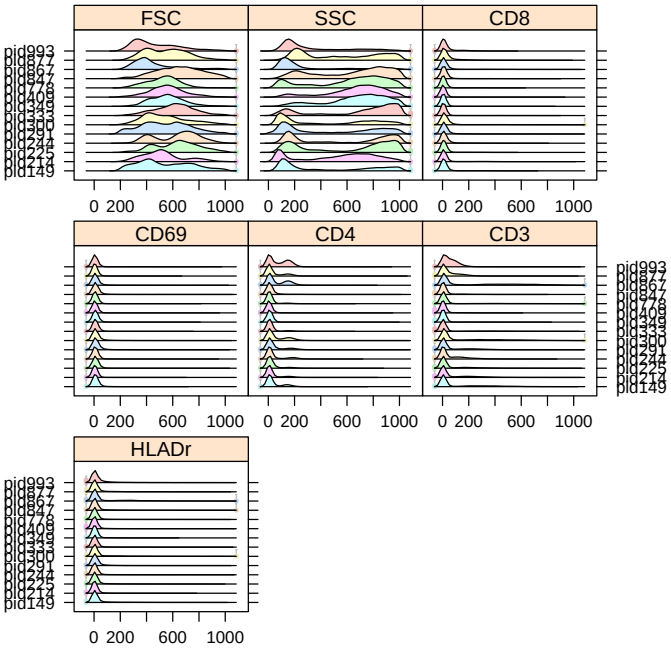
```
                "desc"] <- c("NA", "NA",
                "CD8", "CD69", "CD4", "CD3",
                "HLADr", "NA")
        }
colnames(data) <- c("FSC", "SSC",
            "CD8", "CD69", "CD4", "CD3",
            "HLADr", "Time")
```

The for–loop iterates over each flowFrame in the flowSet and assigns the vector of simpler marker names to the channels. These names are used to access each channel when performing data transformations, for example. The last line of the code snippet renames the channel names that are used for plotting.

Transformation and Preprocessing

The data are now in a form that we can work with more easily. Before we can begin gating we need to perform some simple preprocessing steps. Let's begin by visualizing the data.

```
densityplot(~., data)
```



These density plots show that although the forward and side scatter data are well represented on the linear scale, the fluorescence channels need to be transformed to a log–like scale via an appropriate transformation. flowCore gives many to choose from, including the arcsinh, biexponential, logicle, and logarithm. We will choose the commonly applied logicle transformation.

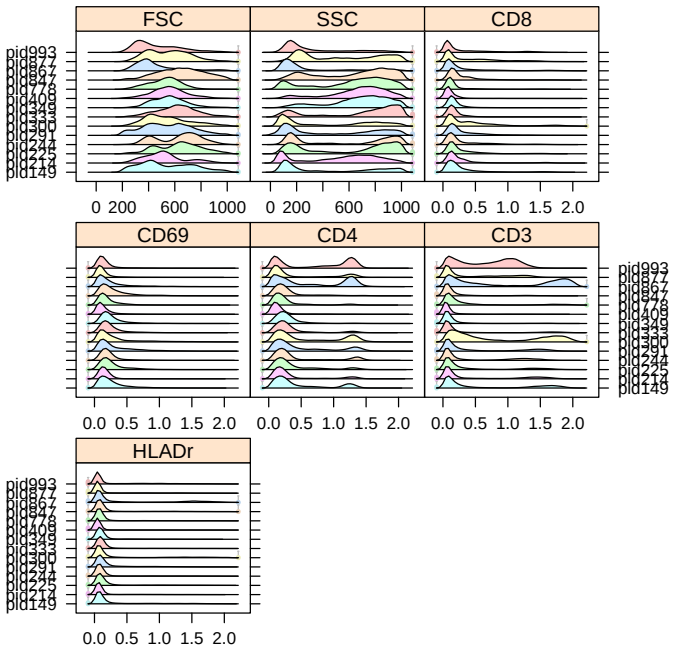
Data Transformation

```
tData <- transform(data, transformList(colnames(data)[3:7],
           logicleTransform()))
```

The above code transforms the data via the `transform()` function, which takes as input the flowSet to be transformed, and an object called a `transformList`, which takes a vector of dimensions to be transformed (in this case channels 3 through 7, excluding FSC and SSC), as well as a transformation function, (`logicleTransform()`). There are other ways to perform data transformation; please refer to the `transform` and `transformList` documentation for other examples.

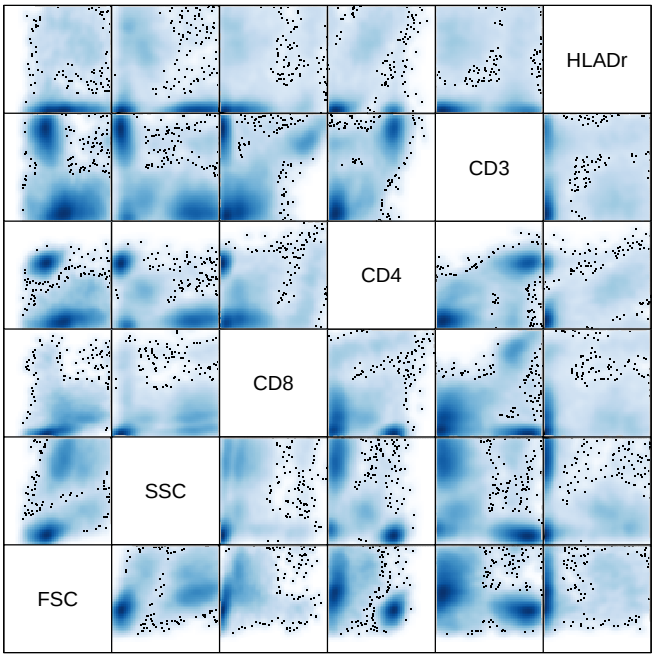
We can now plot the transformed data for comparison.

```
densityplot(~., tData)
```



Looking at the CD3 and CD4 dimensions we can see that the different populations are readily visible. Let's also take a look at some scatterplots.

```
splom(tData[[2]][, c(1, 2, 3, 5,
          6, 7)], smooth = TRUE)
```



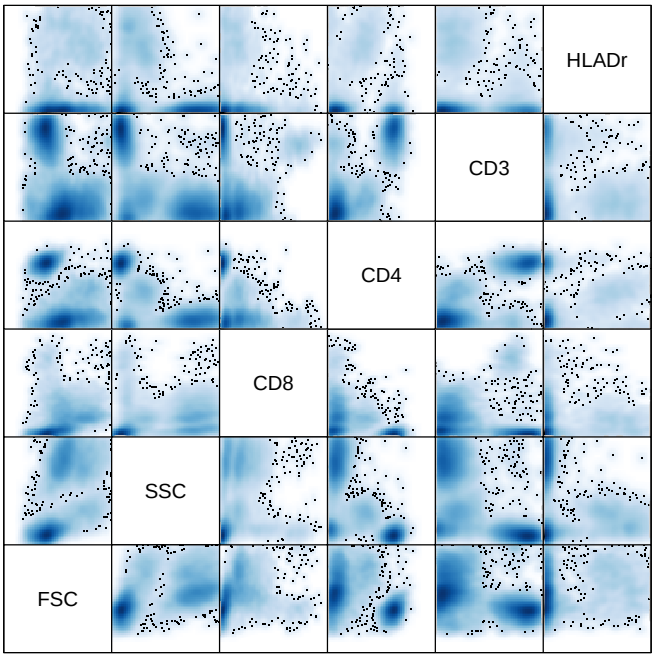
Scatter Plot Matrix

Again, we see the different populations are readily visualized in the different pairwise scatterplots following transformation. We need to also take care of boundary events in FSC and SSC by removing them. flowCore provides a variety of filtering functions for such purposes. Here we will be using boundaryFilter.

```
tData.f <- Subset(tData, filter(tData,
           boundaryFilter(c("FSC", "SSC"))))
```

The above code constructs a boundary filter object via `boundaryFilter()`, taking the names of the channels to be filtered. Next, via `filter()` the filtering operation is performed on `tData`. Finally, the filtered data are extracted into a new flowFrame object via `Subset()`. Let's visualize some of the filtered data:

```
splom(tData.f[[2]][, c(1, 2, 3, 5,
           6, 7)], smooth = TRUE)
```



Scatter Plot Matrix

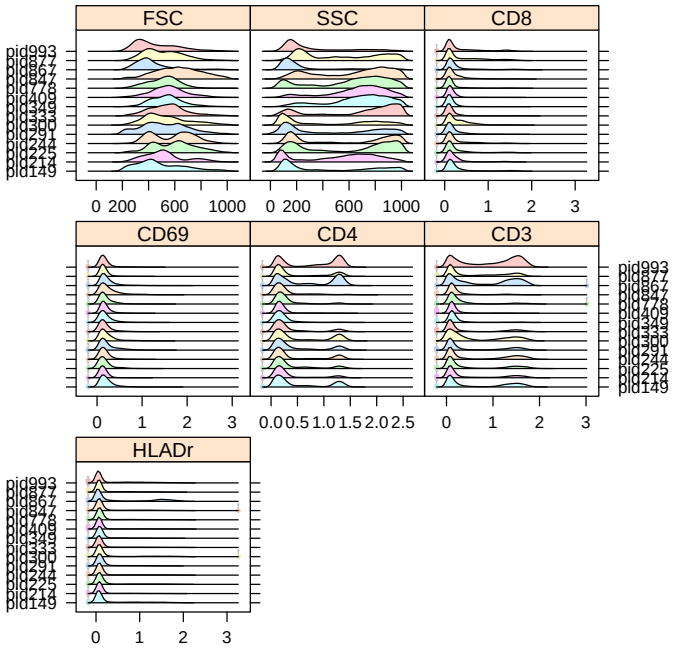
Finally, we'll perform some normalization of the fluorescence channels so that corresponding populations line up across multiple samples. These functions come from the `flowSet` package.

```
tData.n <- normalize(tData.f, normalization(parameters = colnames(tData.f)[3:7],
           normFun = function(x, parameters,
               ...) warpSet(x, parameters,
               ...)))
```

```
Estimating landmarks for channel CD8 ...
Estimating landmarks for channel CD69 ...
Estimating landmarks for channel CD4 ...
Estimating landmarks for channel CD3 ...
Estimating landmarks for channel HLADr ...
Registering curves for parameter CD8 ...
Registering curves for parameter CD69 ...
Registering curves for parameter CD4 ...
Registering curves for parameter CD3 ...
Registering curves for parameter HLADr ...
```

Let's plot the normalized data, just to check.

```
densityplot(~., tData.n)
```



With the preprocessing completed, we move on to automated gating with flowClust and flowMerge.

Gating and Gating Strategy

For this data set, the general gating strategy is to identify lymphocytes in FSC vs SSC that are CD3+/CD4+ or CD3+/CD8+, and express activation marker HLADr. There are a number of ways to attack the problem using flowClust and flowMerge. One tempting approach would be to run flowClust on the entire data set, using all channels to cluster the data simultaneously. Unfortunately, such an "all–at–once" gating approach can sometimes lead to problems, specifically if the scatter and fluorescence channels are not on the same scale.

The reason for this is that flowClust can optionally transform the data using a Box–Cox transformation. FlowClust estimates a Box–Cox transformation parameter from the data that can either be cluster specific or common across all clusters. However, if different dimensions of a cluster are on radically different scales, then, it becomes difficult to identify a transformation that is truly optimal

for that cluster. In practice, we can get around this issue by performing the gating in multiple stages, first gating the scatter dimensions, followed by gating the fluorescence dimensions. However, we will try both approaches and compare the results.

There are a number of other parameters associated with flowClust that can substantially impact the resulting model fit. The most important are: degrees of freedom, `nu`, how the degrees of freedom is estimated, `nu.est={0,1,2}`, and how the Box–Cox transformation parameter is estimated, `trans={0,1,2}`.

The degrees of freedom determines how robust the distribution used to model cell subpopulations will be to outliers. When the degrees of freedom is infinite (i.e. nu=Inf), then a Gaussian distribution is used to model the data. Gaussian distributions are not robust to outliers, and so the presence of these will affect the position and shape of the resulting clusters. This is in contrast to a t–distribution, which is used to model cell subpopulations when nu is between 1 and Infinity. The smaller the degrees of freedom, the more robust to outliers the t–distribution. However, a small degrees of freedom will make it more difficult for flowClust to detect and effectively model rare cell populations. Therefore, the choice of degrees of freedom is a tradeoff between robustness to outliers, and the ability to model rare cell populations (i.e. cell populations with small numbers of events).

The argument `nu.est` determines whether the degrees of freedom is be fixed (nu.est=0), or whether it is estimated during the model fitting procedure (nu.est=1 or nu.est=2). A value of nu.est=1, means that a common degrees of freedom will be estimated for all cell populations in the data. In contrast, a value of nu.est=2, tells flowClust to estimate the degrees of freedom independently for each cell subpopulation. The latter approach provides more flexibility to the model, and is useful if the data contains both rare and abundant cell subpopulations of interest.

The final parameter of interest to us is the transformation–related parameter `trans`, which determines whether, and how the Box–Cox transformation should be applied to the data. Setting `trans=0`, instructs flowClust not to transform the data, whereas setting `trans=1` instructs flowClust to estimate a common transformation parameter for all clusters. Again, more model flexibility can be introduced by setting `trans=2`, instructing flowClust to estimate a distinct transformation parameter for each cluster. Applying a transformation is generallly a good idea if the cell populations appear to be asymmetric, as they frequently are in the forward and side scatter dimensions. In the fluorescence channels, sometimes the populations of interest are relatively symmetric and spherical, in which case one may opt not to transform the data, in order to save time, and reduce the number of parameters in the model.

We will examine the effects of applying a common, distinct, and no transformation during flowClust fitting, as well as common and distinct degrees of freedom as we analyze the example data.

Fraction of CD3+/CD4+ and CD3+/CD8+ T-cells that express activation marker HLADr following treatment with two different drugs.

- All–at–once gating (scatter + fluorescence dimensions)
- Sequential gating: scatter channels first, followed by fluorescence channels.
- Gaussian vs t-mixture models.
- With and without transformation.

We'll begin by comparing sequential vs all–at–once gating.

Sequential Gating

We begin by comparing the sequential vs all–at–once gating strategies on the first sample in the flowSet, using the following code:

Since this may take some time to compute, you can load pre–computed data with:

Note: these results can be loaded with:

```
data(flowMerge1)
```

We'll look at just the first sample for now. Compare sequential vs all–at–once gating, with and without transformation..

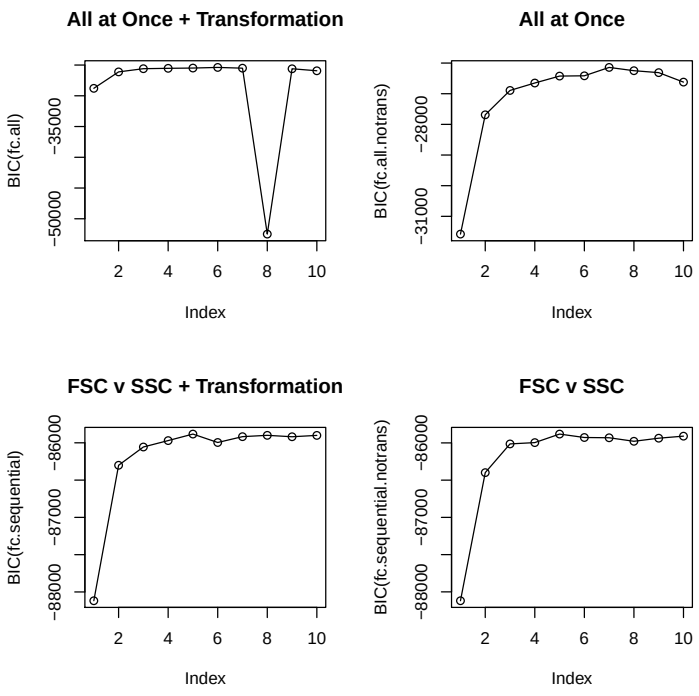
```
fc.all <- flowClust(tData.n[[1]],
           K = 1:10, varNames = colnames(tData.n[[1]])[1:7],
           trans = 1, nu = 4, nu.est = 1)
fc.all.notrans <- flowClust(tData.n[[1]],
           K = 1:10, varNames = colnames(tData.n[[1]])[1:7],
           trans = 0, nu = 4, nu.est = 1)
fc.sequential <- flowClust(tData.n[[1]],
           K = 1:10, varNames = colnames(tData.n[[1]])[c(1,
               2)], trans = 1, nu = 4,
           nu.est = 1)
fc.sequential.notrans <- flowClust(tData.n[[1]],
           K = 1:10, varNames = colnames(tData.n[[1]])[c(1,
               2)], trans = 0, nu = 4,
           nu.est = 1)
```

The first call to flowClust fits models containing from 1 through 10 clusters (specified by `K`) to all the dimensions of the data (specified by the `varNames` parameter). We applied the default parameter values for transformation and

degrees of freedom, specifically, common degrees of freedom, and common Box–Cox transformation parameter. The second call applies only to the scatter dimensions.

We can examine the BIC plots for each set of models and we can choose the model with the max BIC from each fitting procedure and then we can proceed to plot the best fitting models to get an idea how they differ.

```
par(mfrow = c(2, 2))
plot(BIC(fc.all), main = "All at Once + Transformation",
           type = "o")
plot(BIC(fc.all.notrans), main = "All at Once",
           type = "o")
plot(BIC(fc.sequential), main = "FSC v SSC + Transformation",
           type = "o")
plot(BIC(fc.sequential.notrans),
           main = "FSC v SSC", type = "o")
```



Model selection can be automated, without examining the BIC plots.

```
fc.all <- fc.all[[which.max(BIC(fc.all))]]
fc.all.notrans <- fc.all.notrans[[which.max(BIC(fc.all.notrans))]]
fc.sequential.notrans <- fc.sequential.notrans[[which.max(BIC(fc.sequential.notrans))]]
fc.sequential <- fc.sequential[[which.max(BIC(fc.sequential))]]
```

The above code identifies the model with the maximum bayesian information criterion (BIC) value and selects it from the list of flowClust fitted models. To plot the models, we need to pass the flowClust fit and the data frame to the plot function, and specify which dimensions we want to plot. This is cumbersome. The flowMerge package provides another option. We can construct a flowObj object using the data and fitted model. This can be passed directly to the plot function which will plot all projections of the data.

flowClust generates multiple models for the same set of data. In order to keep track of which data set corresponds to which model, we can combine them in a `flowObj` object. Plotting `flowClust` models is also simplified for this type of object.

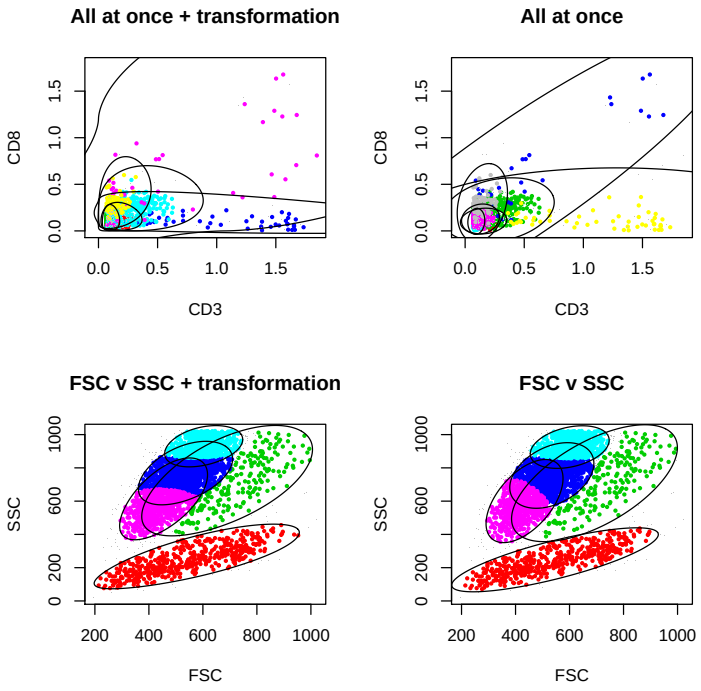
```
fc.all <- flowObj(fc.all, tData.n[[1]])
fc.all.notrans <- flowObj(fc.all.notrans,
           tData.n[[1]])
fc.sequential <- flowObj(fc.sequential,
           tData.n[[1]])
fc.sequential.notrans <- flowObj(fc.sequential.notrans,
           tData.n[[1]])
```

- flowObj combined model and data
- Memory efficient: keep only one copy of the data for multiple models
- Extract data via `getData(object)`

The `flowObj` function will combine the flowClust model and the flowFrame into a single object. This simplifies data manipulation later on since we won't have to keep track of which flowFrame goes with which model when we're analyzing multiple samples.

```
par(mfrow = c(2, 2))
plot(as(fc.all, "flowClust"), data = tData.n[[1]],
           subset = c("CD3", "CD8"), pch = 20,
           main = "All at once + transformation")
plot(as(fc.all.notrans, "flowClust"),
           data = tData.n[[1]], subset = c("CD3",
```

```
                "CD8"), pch = 20, main = "All at once")
plot(as(fc.sequential, "flowClust"),
           data = tData.n[[1]], pch = 20,
           main = "FSC v SSC + transformation")
plot(as(fc.sequential.notrans,
           "flowClust"), data = tData.n[[1]],
           pch = 20, main = "FSC v SSC")
```



- All–at–once clustering with or without transformation doesn't identify the CD8/CD3+ cell population correctly.
- Optimal parameters for scatter dimensions and fluorescence dimensions are different.
- Could use gaussian mixtures, but it would affect the clustering of scatter dimensions.
- Best option is sequential gating of scatter followed by fluorescence dimensions.
- Single cell population modeled by three clusters in the scatter channels.
- `flowMerge` can help model it as distinct cell population.

We see that the model in the first panel (clustering all dimensions simultaneously) doesn't identify the CD8+/CD3+ cell subpopulation properly. Similarly, the second model (clustering all dimensions, no transformation) also doesn't pick up the CD8+/CD3+ populations. It is possible that the degrees of freedom is too low (set to nu=4, and estimated from the data). However, if we were to use a Gaussian distribution, this would impact gating of the populations in the scatter dimensions. We see that the sequential gating approach (FSC vs SSC with and without transformation) does a good job of fitting the cell populations in the scatter dimensions. Furthermore, visually, we don't see any substantial difference between the two models fit to the scatter data in this case. The only issue is that three clusters are required to model what is clearly a single population in the upper left of the FSC vs SSC plot. This is where the flowMerge package comes into play. FlowMerge makes some specific assumptions about the parameters passed used to fit the flowClust model. Specifically , flowMerge requires that a common degrees of freedom and a common transformation parameter have been used during model fitting, (i.e. `(trans=1, nu.est=1)`). This was the case above, so we can continue.

flowMerge Objects

```
getSlots("flowMerge")
```

```
       merged       entropy
    "numeric"     "numeric"
        DATA       expName
"environment"   "character"
     varNames             K
  "character"     "numeric"
           w            mu
     "vector"      "matrix"
       sigma        lambda
      "array"     "numeric"
          nu             z
    "numeric"      "matrix"
           u         label
     "matrix"      "vector"
  uncertainty  ruleOutliers
     "vector"      "vector"
 flagOutliers        rm.min
     "vector"     "numeric"
       rm.max       logLike
    "numeric"     "numeric"
         BIC           ICL
    "numeric"     "numeric"
```

Merge overlapping clusters to model distinct cell populations.

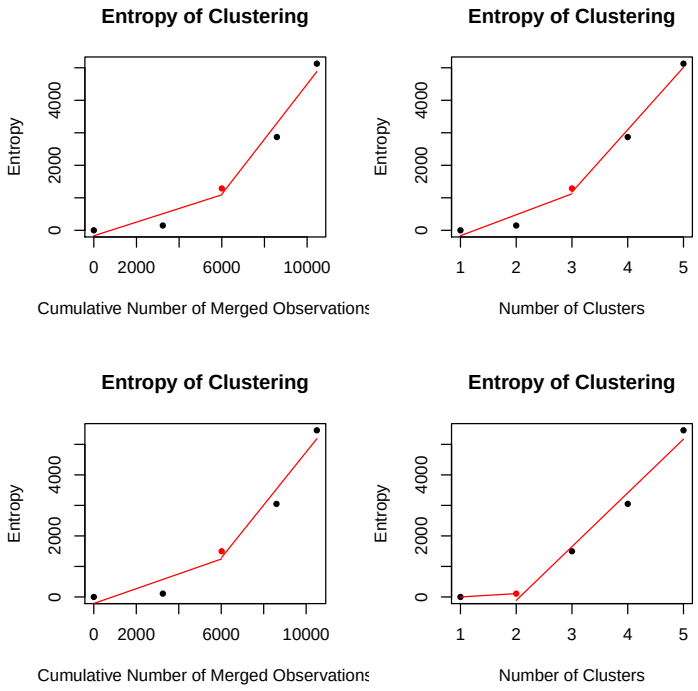
flowMerge requires that each cluster is modeled with a common degrees of freedom and a common transformation parameter. Merging clusters is trivial, given the `flowObj` model object:

```
m1 <- merge(fc.sequential.notrans)
m2 <- merge(fc.sequential)
```

The code above merges clusters in the sequentially gated model, with and without transformation. The output is a list of merged models. The next step is to identify the optimal merged model.

FlowMerge iteratively merges the clusters in the flowClust model to generate distinct cell populations that are represented by single model components. Each merged model is evaluated based on the entropy of clustering. The change in entropy can be plotted as a function of the number of clusters, or as a function of the cumulative number of merged observations. The optimal merged model is the one where the slope of the entropy vs number of clusters or entropy vs number of merged observations plot changes.

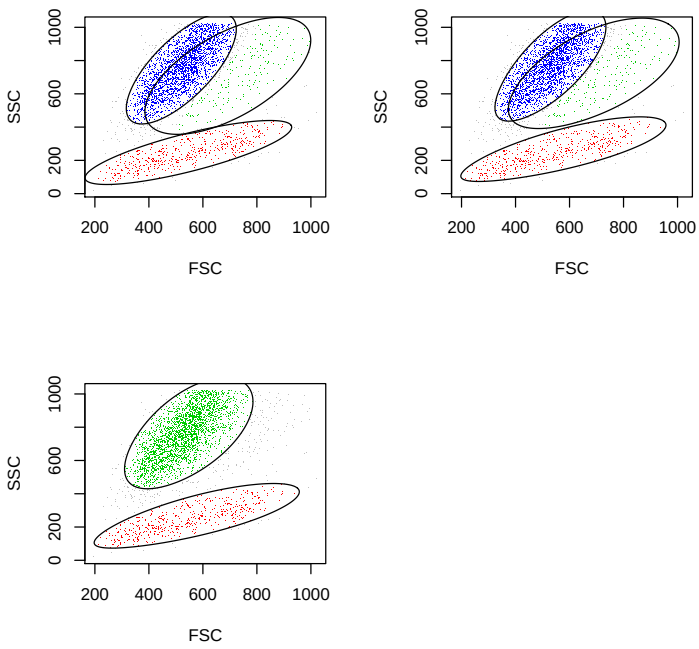
```
par(mfrow = c(2, 2))
fitPiecewiseLinreg(m1, plot = T,
           normalized = TRUE, )
fitPiecewiseLinreg(m1, plot = T,
           normalized = FALSE)
fitPiecewiseLinreg(m2, plot = T,
           normalized = TRUE)
fitPiecewiseLinreg(m2, plot = T,
           normalized = FALSE)
```



The optimal number of clusters is chosen to be either two or three, depending on which model and which model selection method we choose.

We can plot the different merged models to see how they differ.

```
par(mfrow = c(2, 2))
plot(m1[[3]])
plot(m2[[3]])
plot(m2[[2]])
```



In practice, any of these model fits seems to be acceptable. We'll choose to use the 3 cluster model based on sequential gating with transformation.

We can identify the optimal merged model from the model 2 series of fits using `fitPiecewiseLinreg` and then extract the gated populations:

```
m <- m2[[fitPiecewiseLinreg(m2,
          normalize = FALSE)]]
fl <- split(f = m)
```

The `split()` function extracts the cells within each of the gates that match the filtering criteria for outliers. By default the filtering criteria selects events that that are within the 90% quantile of the gate. We can change the filtering criteria using the `ruleOutliers()` function. Below, we change the quantile level for filtering events from 0.9 to 0.99. As a result, when we extract the data we have more events included in the gate.

We can modify the threshold for outlier detection when plotting or extracting a cell population. The `ruleOutliers` function modifies the outlier detection rule. Most commonly, we'll use a certain quantile of the distribution, beyond

which events are considered outliers. The default is the 90% quantile. When we extract the population, we see the number of included events changes based on the outlier threshold.

```
ruleOutliers(m)
```

```
Rule of identifying outliers: 90% quantile
```

```
dim(split(f = m)[[1]])
```

```
   events parameters
      481          8
```

```
ruleOutliers(m) <- list(level = 0.99)
```

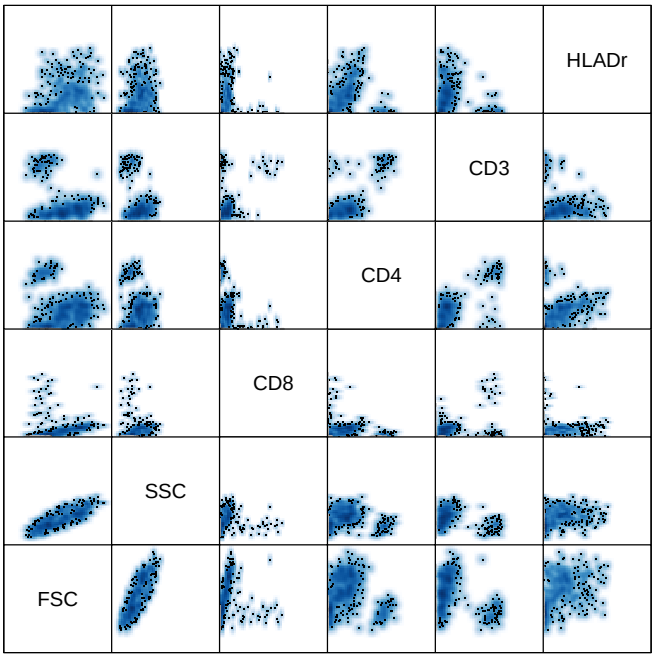
```
Rule of identifying outliers: 99% quantile
```

```
dim(split(f = m)[[1]])
```

```
   events parameters
      510          8
```

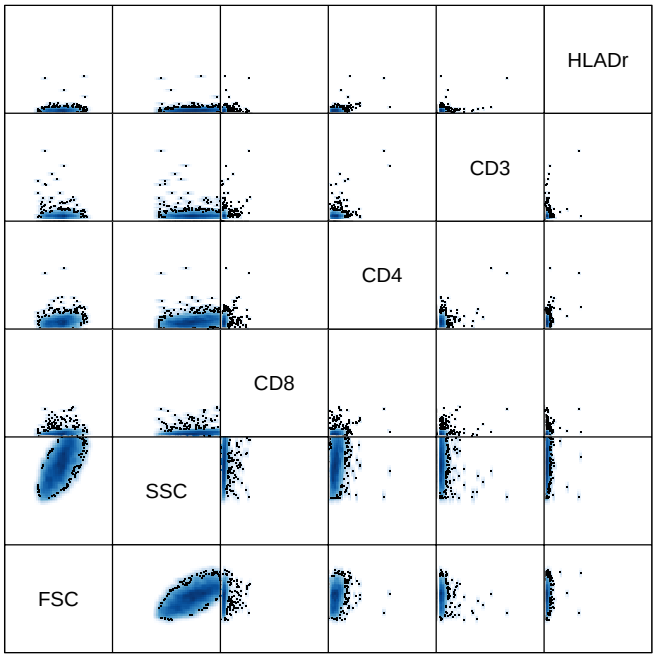
Once populations have been gated and extracted in the scatter dimensions, we can proceed to gate populations in the fluorescence dimensions. It is interesting to plot the gated populations to have a look.

```
splom(fl[[1]][, c(1, 2, 3, 5, 6,
           7)])
```



Scatter Plot Matrix

```
print(splom(fl[[2]][, c(1, 2, 3,
          5, 6, 7)]))
```



Scatter Plot Matrix

Examining the previous plots, we see that population 1 contains CD3+, CD4+, and CD8+ cells. This is the population we will focus on. We will use flowClust to gate the CD4,CD8, and CD3 populations. Looking at the scatter plots, we see the positive cell populations we're interested appear to be somewhat rare. We will use Gaussian, rather than t-distributions to fit the data. We will also try fitting the data with and without transformation.

```
fl.trans <- flowClust(fl[[1]],
            varNames = colnames(fl[[1]])[c(3,
                5, 6)], K = 1:10, trans = 1,
            nu = Inf, nu.est = 0)
fl.notrans <- flowClust(fl[[1]],
            varNames = colnames(fl[[1]])[c(3,
                5, 6)], K = 1:10, trans = 0,
            nu = Inf, nu.est = 0)
```

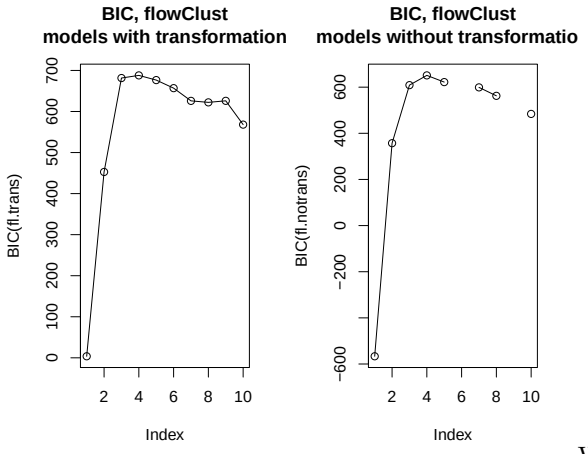
We run the second stage clustering on the CD3/CD4/CD8 fluorescence channels, using gaussian mixtures and t–mixtures. Populations are relatively symmetric in the fluorescence channels, so we opt for no transformation.

Again, the data has been pre-computed to save time. You can load it with:

```
data(flowMerge2)
```

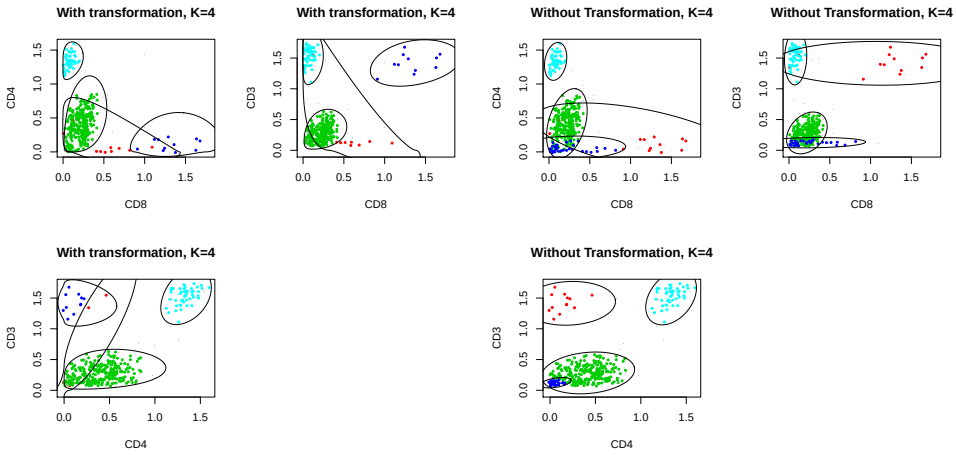
Next we plot and choose the best fitting model, again by examining the BIC plots.

```
par(mfrow = c(1, 2))
plot(BIC(fl.trans), type = "o",
           main = "BIC, flowClust\nmodels with transformation")
plot(BIC(fl.notrans), type = "o",
           main = "BIC, flowClust\nmodels without transformation")
```



We see that the best fitting model with or without transformation has four clusters. Also note that some of the models with no transformation failed to converge, returning a BIC value of NA. These are dealt with apporpriately within flowClust. How do the max BIC models compare?

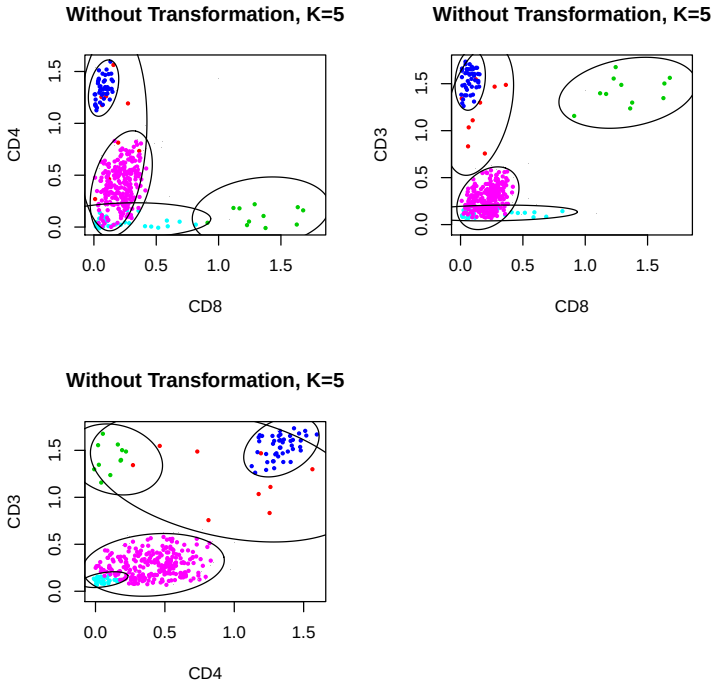
```
plot(flowObj(fl.trans[[4]], fl[[1]]),
           pch = 20, new = T, main = "With transformation, K=4")

plot(flowObj(fl.notrans[[4]], fl[[1]]),
           pch = 20, new = T, main = "Without Transformation, K=4")
```



It's interesting to note that the CD3+/CD8+ cell population identified in the model without transformation, and K=4 clusters picks up a couple of CD3+/CD8- events. This is due to the low density nature of this cell population and our use of t–mixtures for modeling. We could have a look at the K=5 solution (even though it doesn't have the highest BIC value) to see if we get further resolution of this cell population.

```
plot(flowObj(fl.notrans[[5]], fl[[1]]),
           pch = 20, new = T, main = "Without Transformation, K=5")
```

```
NULL
```



The CD3+/CD8+ and CD3+/CD4+ cell populations are now well defined. There are two additional populations that pick up some dim events in the CD4 and CD8 dimensions. This is acceptable. We modify the filter to pull in a few extra events when we look for HLADr+ cells in the CD3+/CD4+ and CD8+/CD3+ populations. In the end, we're using Gaussian distributions here, so it's acceptable. We can use the `order()` function to pull out the populations of interest, based on their mean fluorescence intensities.

We want the CD8+/CD3+ and CD4+/CD3+ subpopulations. We can make use of model parameters estimated by `flowClust` to select these programmatically.

```
fl.2 <- flowObj(fl.notrans[[5]],
           fl[[1]])
ruleOutliers(fl.2) <- list(level = 0.99)
```

```
Rule of identifying outliers: 99% quantile
```

```
order(fl.2@mu[, 1], fl.2@mu[, 3],
           decreasing = TRUE)[1]
```

```
[1] 2
```

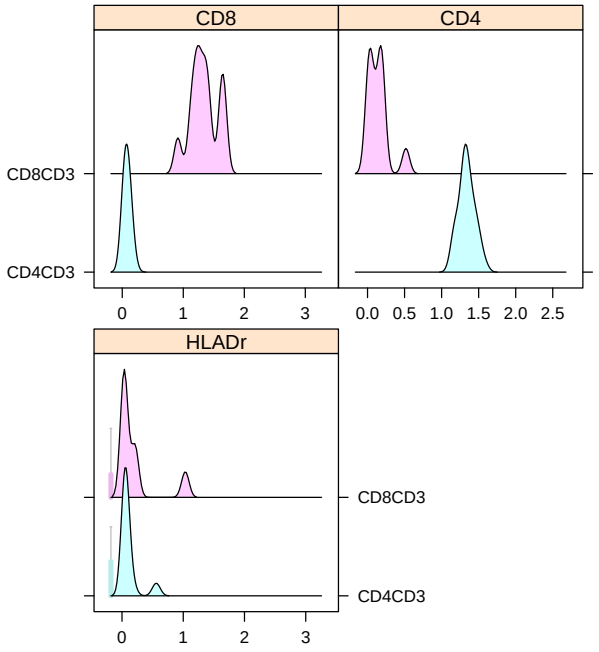
```
order(fl.2@mu[, 2], fl.2@mu[, 3],
            decreasing = TRUE)[1]
```

```
[1] 3
```

- Raise the outlier threshold.
- Use `order` to sort populations on their CD8/CD3, and CD4/CD3 columns.
- The CD8+/CD3+ and CD4+/CD3+ populations are the first in the list returned by the above code.

The above code orders the mean fluorescence intensity of each population, in increasing order, by the CD8/CD3 (columns 1 and 3) and CD4/CD3 markers (columns 2 and 3). The first element of each of these are the indices of the CD8+/CD3+ population, and CD4+/CD3+ populations, respectively. We can pull these out using the split function again.

```
hladr <- split(x = getData(fl.2),
            f = as(fl.2, "flowClust"),
            population = list(CD8CD3 = order(fl.2@mu[,
                1], fl.2@mu[, 3], decreasing = TRUE)[1],
                CD4CD3 = order(fl.2@mu[,
                    2], fl.2@mu[, 3], decreasing = TRUE)[1]))
print(densityplot(~CD8 + CD4 +
            HLADr, as(hladr, "flowSet")))
```



We see that the HLADr+ cells are really rare, so there's no point running flowClust on these. We'll use the `rangeGate()` function for one dimensional gating instead.

```
hladr.cd3.cd4 <- Subset(hladr[[1]],
           rangeGate(hladr[[1]], stain = "HLADr",
               plot = FALSE, alpha = 0.5,
               filterId = "CD3+CD4+HLAct"))
hladr.cd3.cd8 <- Subset(hladr[[2]],
           rangeGate(hladr[[2]], stain = "HLADr",
               plot = FALSE, alpha = 0.5,
               filterId = "CD3+CD8+HLAct"))
100 * dim(hladr.cd3.cd4)[1]/dim((data[[1]]))[1]

events
 0.025

100 * dim(hladr.cd3.cd8)[1]/dim((data[[1]]))[1]

events
 0.075
```

We find that 0.02 percent of the cells are CD3+/CD8+/HLADr activated, and 0.05 percent of the cells are CD4+/CD3+/HLADr activated. We have determined what the gating procedure should be for a single sample. Now we want to analyze the rest of the samples in the same manner. We would also like this to be analyzed quickly, and if we have multiple processors available, we can use the `snowfall` package to help us distribut the work.

Automated Gating with FlowClust in Parallel

In order to use the snow and snowfall package, you need to construct a `snow` cluster. Snowfall simplifies all this for you with a call to `sfInit()`.

`flowClust` has support for parallel processing via the `snow` and `snowfall` packages. We initialize `snowfall` by running:

```
sfInit(parallel = TRUE, cpus = 4)
```

`sfInit` constructs a snow cluster utilizing 2 cpus on the local machine. `flowClust` will detect that `snowfall` is in use and will utilize the snowfall package to distribute computations amongst the cluster nodes.

The following code performs the forward and side scatter gating on the full set of samples. `fsApply` cycles through the samples in the flowSet. We'll use the parameter settings we decided upon from the first run.

```
result <- fsApply(tData.n, function(x) flowClust(x,
           K = 1:10, trans = 1, nu = 4,
           nu.est = 1, varNames = colnames(x)[1:2]))
sfStop()
```

The data takes some time to process. A pre-computed version has been provided to save time:

```
data(flowMerge3)
```

Next we extract the max BIC solution for each sample, construct flowObj objects, and do the merging.

Essentially, we proceed as before, but wrapping calls in `lapply`, and `sapply` statements to loop through the different samples.

```
result <- lapply(result, function(x) x[[which.max(BIC(x))]])
result <- sapply(1:length(data),
           function(i) flowObj(result[[i]],
               tData.n[[i]]))
result <- lapply(result, function(x) {
           m <- merge(x)
           m[[fitPiecewiseLinreg(m)]]
       })
```

Semiautomated Gating

We now need to select the lymphocyte populations. These can vary from one instance of running the algorithm to another due to random initialization. The code below plots each FSC vs SSC gating and allows you to click with your mouse in the center of the lymphocyte cluster (usually the high density population around FSC 400, SSC 200).

```
indices <- unlist(lapply(result,
           function(x) {
               plot(x)
               z <- unlist(locator(n = 1))
               inc <- which.min(sapply(1:x@K,
                   function(i) mahalanobis(box(z,
                       lambda = x@lambda),
                       x@mu[i, 1:2], x@sigma[i,
                         , ])))
               plot(x, include = inc)
               inc
           }))
```

We load the pre-gated data for this analysis. We then proceed to exract the populations of interest (see vignette for details), and run the next stage of clustering.

The code above will compute the mahalanobis distance between the location you select and each cell population on the transformed scale. The `mahalanobis()` function is provided by the stats package. The Box–Cox transformation `box()` is provided by flowClust, and takes the transformation parameter specific to the sample under consideration as an argument, as well as the data being transformed.

Again, we extract the populations of interest using `split()`, and run the next stage of clustering on the fluorescence channels.

```
result.split <- sapply(1:length(result),
           function(i) split(f = result[[i]])[[indices[i]]])
```

Stage 2 Clustering

We'll run `flowClust` with cluster–specific transformation parameters. We won't be able to run `flowMerge` in this case, but it will give us more model flexibility.

```
sfInit(parallel = TRUE, cpus = 2)
result.fl <- lapply(result.split,
           function(x) flowClust(x, B.init = 100,
               K = 1:10, varNames = colnames(x)[c(3,
                   5, 6)], nu = Inf, nu.est = 0,
               trans = 2))
sfStop()
```

And, again, we can load the pre-computed data to save time here:

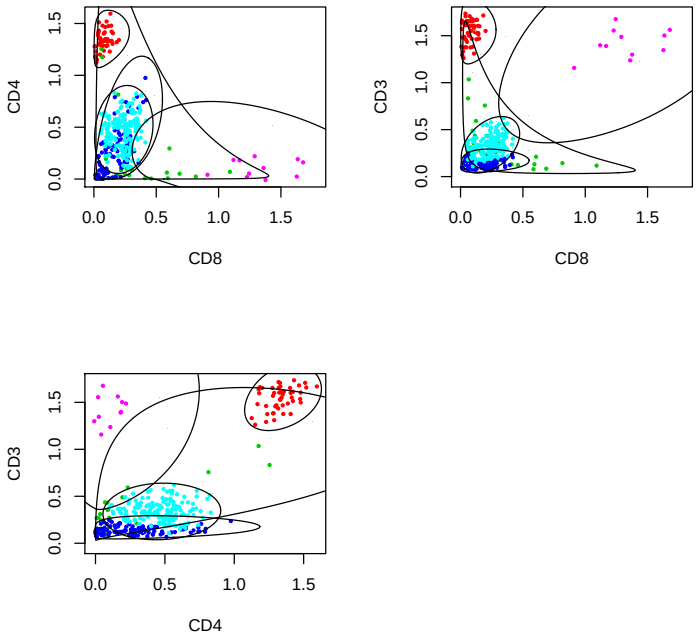
```
data(flowMerge4)
```

We will extract the max BIC model for the fluorescence clustering, construct the flowObj object, and plot it to determine whether the fit is good. We can manually explore models with fewer or more clusters if the max BIC fit is not appropriate. Extract the max BIC model

```
result.fl.bic <- sapply(1:length(result.fl),
           function(i) flowObj(result.fl[[i]][[which.max(BIC(result.fl[[i]]))]],
               result.split[[i]]))
```

Recall that the CD8+/CD3+ population is not well defined for sample 1. We manually select a model with five clusters rather than the model with 4 clusters.

```
plot(result.fl.bic[[1]], pch = 20)
result.fl.bic[[1]] <- flowObj(result.fl[[1]][[5]],
           result.split[[1]])
plot(result.fl.bic[[1]], pch = 20)
```



Extract the CD4+/CD3+ and CD8+/CD3+ populations and look at the HLADr markers.

```
fl.split.cd8 <- sapply(1:length(result.fl.bic),
           function(i) split(x = result.split[[i]],
               f = as(result.fl.bic[[i]],
                   "flowClust"), population = list(order(result.fl.bic[[i]]@mu[,
                   1], result.fl.bic[[i]]@mu[,
                   3], decreasing = TRUE)[1])))
fl.split.cd4 <- sapply(1:length(result.fl.bic),
           function(i) split(x = result.split[[i]],
               f = as(result.fl.bic[[i]],
                   "flowClust"), population = list(order(result.fl.bic[[i]]@mu[,
                   2], result.fl.bic[[i]]@mu[,
                   3], decreasing = TRUE)[1])))
```

Finally, we can gate the HLADr positive populations using the rangeGate function again.

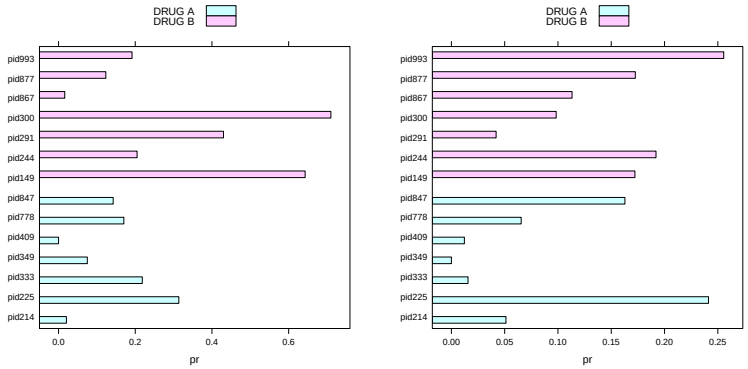
```
all.hladr.cd3.cd8 <- lapply(fl.split.cd8,
           function(x) Subset(x, rangeGate(x,
               stain = "HLADr", plot = FALSE,
               borderQuant = 0.5, alpha = 0.5,
               filterId = "CD3+CD8+HLAct")))
all.hladr.cd3.cd4 <- lapply(fl.split.cd4,
           function(x) Subset(x, rangeGate(x,
               stain = "HLADr", plot = FALSE,
               borderQuant = 0.5, alpha = 0.5,
               filterId = "CD3+CD4+HLAct")))
```

We can get the percentage of cells from the counts.

```
CD4.hladr <- sapply(1:length(all.hladr.cd3.cd4),
           function(i) 100 * dim(all.hladr.cd3.cd4[[i]])[1]/dim(data[[i]])[1])
CD8.hladr <- sapply(1:length(all.hladr.cd3.cd8),
           function(i) 100 * dim(all.hladr.cd3.cd8[[i]])[1]/dim(data[[i]])[1])
```

```
print(barchart(reorder(PatientID,
           as.numeric(factor(GroupID))) ~
           pr, data = data.frame(pr = as.vector(CD4.hladr),
           pData(data)[, c("GroupID",
               "PatientID")]), groups = GroupID,
           auto.key = T))
```

```
print(barchart(reorder(PatientID,
           as.numeric(factor(GroupID))) ~
           pr, data = data.frame(pr = as.vector(CD8.hladr),
           pData(data)[, c("GroupID",
               "PatientID")]), groups = GroupID,
           auto.key = T))
```



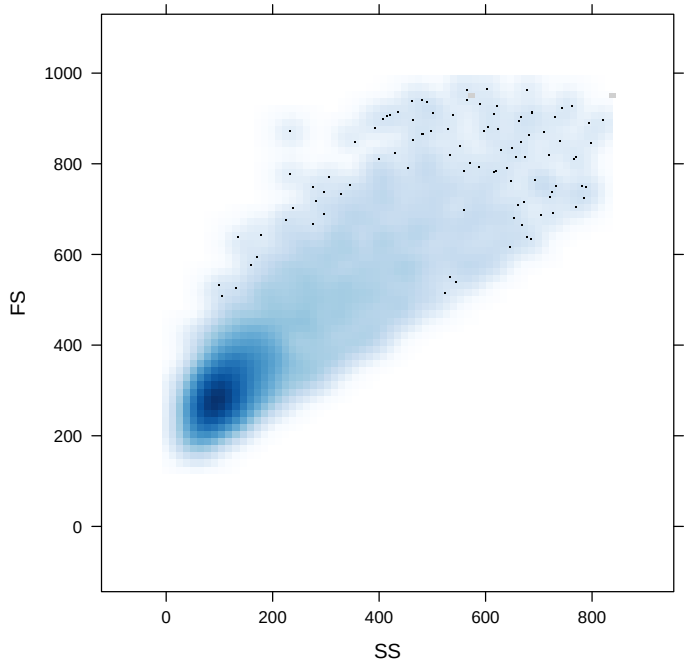
Metaclustering

Next we will examine how to match cell populations across multiple samples using the models output by flowClust and flowMerge. We will use a different data set for this example, consisting of nine lymph node samples from individuals with DLBCL, labelled with three different antibodies, in addition to forward and side-scatter.

```
require(flowMetaCluster)
require(snowfall)
data(lymphoma)
```

We load the lymphoma data above and run flowClust in parallel using the snowfall package. We'll run the clustering on the fluorescence dimensions only, since the forward and side scatter dimensions are pretty clean, as we can see below.

```
print(xyplot(FS ~ SS, lymph[[1]]))
```



The above plot is representative of the data distribution in the forward and side scatter dimensions of this data set. Therefore there is no need to run flowClust on FSC and SSC. We choose the default parameterization of an estimated transformation, and an estimated degrees of freedom, and fit models with one through ten components on the fluorescence channels.

We run `flowClust` on the data, followed by `flowMerge`.

```
result.lymph <- fsApply(lymph,
           function(x) flowClust(x, varNames = colnames(x)[-c(1:2)],
               K = 1:10, nu = 4, trans = 1,
               nu.est = 1))
```

```
sfInit(parallel = TRUE, cpus = 4)
clusterEvalQ(sfGetCluster(), library(flowMerge))
m <- sfLapply(result.lymph.opt,
           merge)
k <- sfLapply(m, fitPiecewiseLinreg)
sfStop()
m <- sapply(1:length(k), function(i) m[[i]][[k[[i]]]])
```

With the merged results we can now metacluster the samples. We have two possible approaches. The first uses the flowMerge methodology to merge cell populations across samples using the entropy measure. The second method uses the pairwise mahalanobis distance between cell populations and their centers, with the additional constraints of allowing only a single cell population from each sample to belong to each metacluster. We provide examples using these methods below. The package included, `flowMetaCluster` is a development release for this workshop.

```
#Mahalanobis distance metaclustering
meta<-metaCluster(m);
#Entropy-based metaclustering
meta2<-metaClusterByMerging(m);
```

Two different data structures, `meta` and `meta2` store information required to generate metacluster memberships for entropy–based metaclustering and metaclustering based on Mahalanobis distance between populations. The `flowMetaCluser` provides functionality for extracting and plotting the metaclustered cell populations using these structures. More information is provided in the Vignette.

flowMetaCluster Objects

Mahalanobis distance based metaclustering

```
getSlots("flowMetaCluster") #S4 object
```

```
         flow     Indicator
"environment"  "data.frame"
```

Entropy Based Metaclustering

```
names(meta2); #S3 object
```

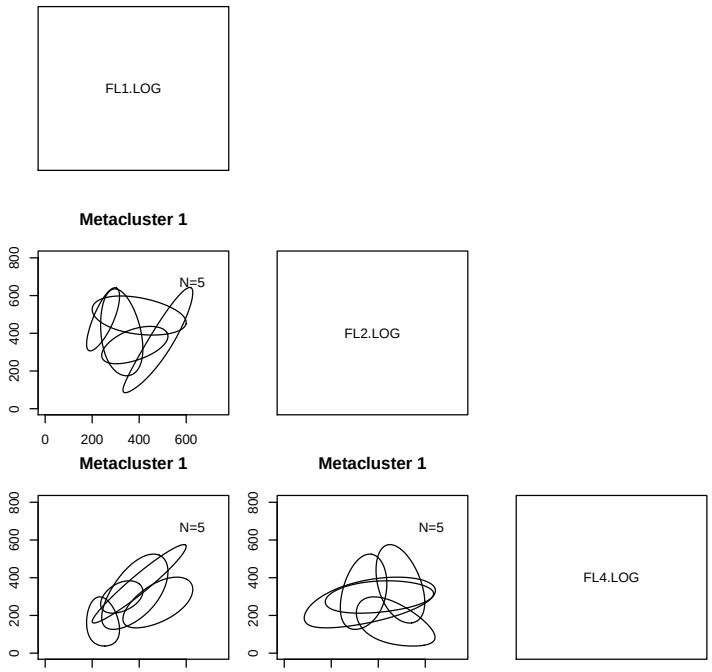
```
[1] "z"         "adjacency" "entropy"
```

The two function calls above perform metaclustering of the cell populations across samples using two different methods. The first `metaCluster` applies a Mahalanobis distance based approach to identify and cluster cell populations that are near each other. The algorithm applies an additional constraint that allows no more than one cell population from each sample to belong to a single metacluster (i.e. there is a one-to-one, rather than many-to-one mapping between samples and metaclusters). The second method makes use of cluster merging based on the entropy of overlapping clusters. We concatenate the individual flowFrames to be metaclustered, and evaluate each flowClust model against the

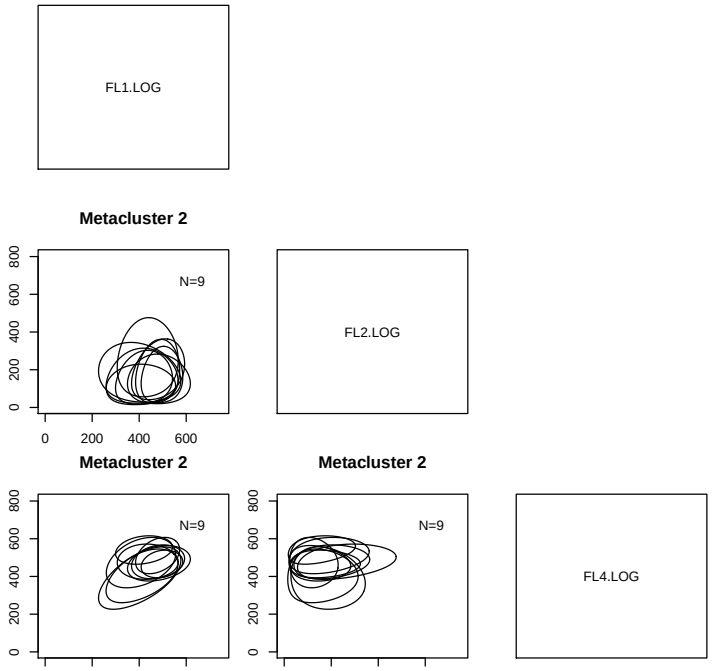
entire data set. We then apply the merging algorithm used in `flowMerge` to combine overlapping cell populations, and look for the changepoint in the entropy vs cumulative number of merged observations plot.

The `flowMetaCluster` package provides implementations of both of these methods, together with functions for visualization of the metaclustered data. The mahalanobis-distance based metaclustering can be visualized by calling the `plot()` method.

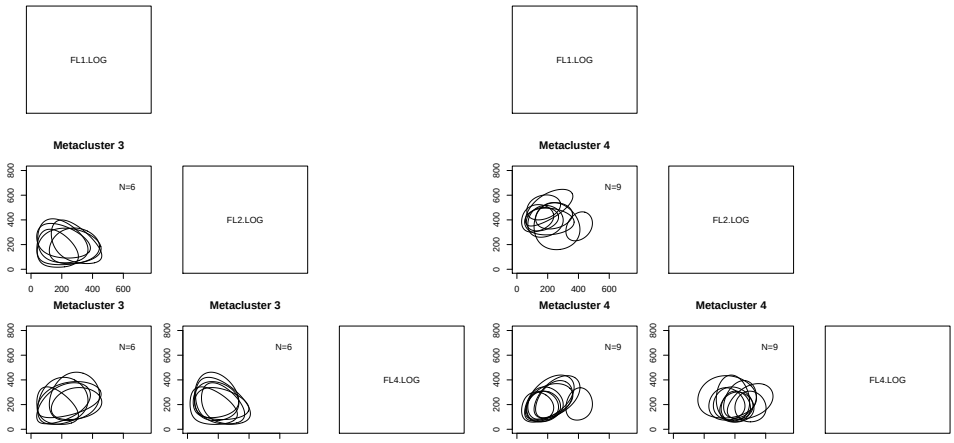
```
print(plot(meta, by = "metacluster",
           KK = 1, main = "Metacluster 1"))
```




```
print(plot(meta, by = "metacluster",
           KK = 2, main = "Metacluster 2"))
```




```
print(plot(meta, by = "metacluster",
           KK = 3, main = "Metacluster 3"))

print(plot(meta, by = "metacluster",
           KK = 4, main = "Metacluster 4"))
```



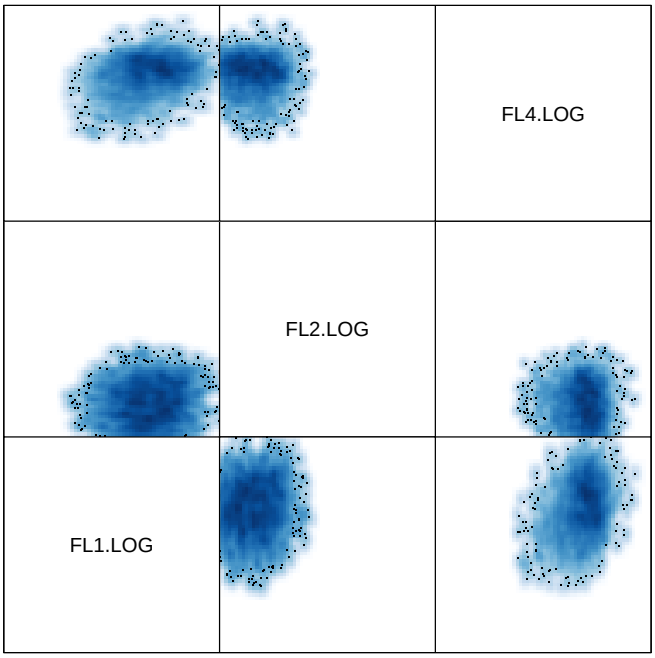
There are a number of helper methods for working with the metaclustered data.

```
# Return metacluster membership of each cell population.
metaClusterPopulations(meta)
# Return the transformed or untransformed means of the populations in each metacluster
getClassMeans(meta)
# Return the list of flowMerge objects being clustered
getFlow(meta)
#Return clustering statistics about each metacluster
getClusterStats(meta)
# Return the distance matrix used for clustering the data
getDist(meta)
```

These methods can be used to extract information about each metacluster for further analysis.

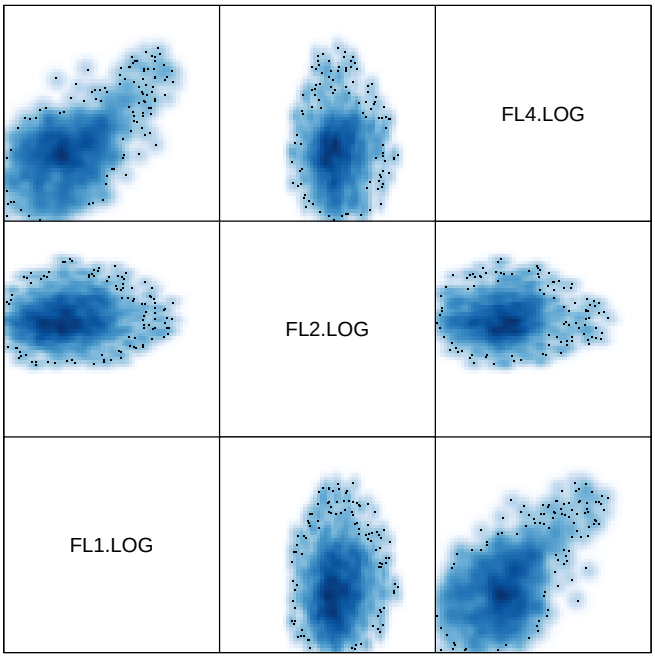
We can use the `split` method to extract individual cell populations within each metacluster. From the plots above, we we want to extract metaclusters two and four. `flowMetaCluster` implements the `split` method for this purpose.

```
metacluster2 <- split(meta, metaK = 2)
metacluster4 <- split(meta, metaK = 4)

print(splom(metacluster2[[1]][,
           c(3, 4, 5)]))
```



Scatter Plot Matrix

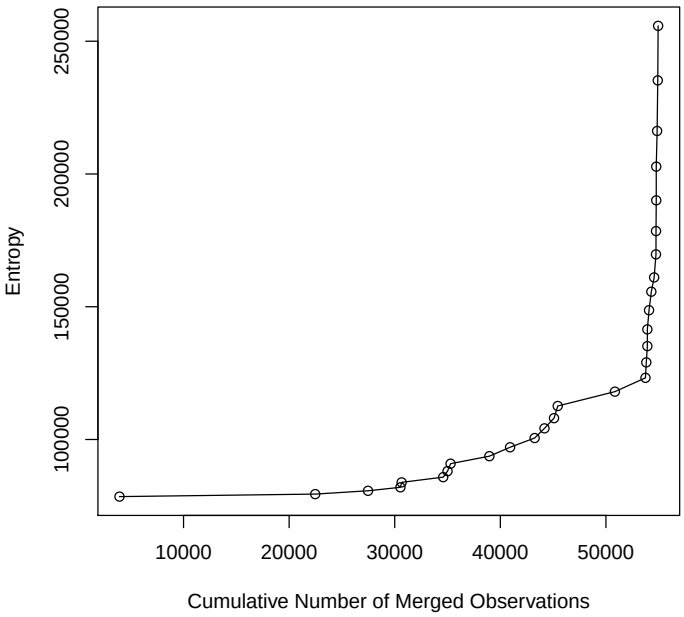
```
print(splom(metacluster4[[1]][,
          c(3, 4, 5)]))
```



Scatter Plot Matrix

We can also have a look at the entropy vs cumulative number of merged observations plot for the entropy based metaclustering.

```
screePlot(meta2)
```



- The entropy vs cumulative number of merged observations identifies when sufficient populations have been merged across samples.
- Localization of the changepoint can be done automatically, as in `flowMerge`.

```
cp <- findChangePoint(screePlot(meta2,
           main = "Entropy vs Cum. # Merged Obs."))
print(cp)
```

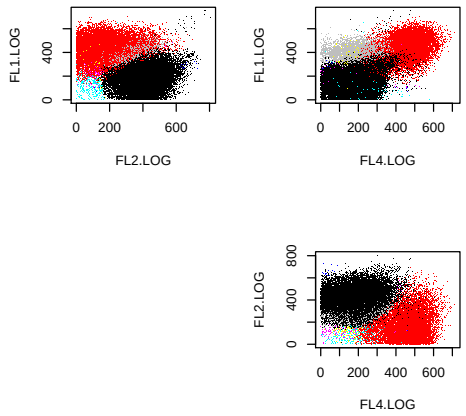
```
[1] 16
```

The `screePlot` function generates a plot of the entropy vs cumulative number of merged clusters, and returns the data that can be used as input to the `findChangePoint` function. The changepoint in this plot corresponds to the model where further merging of populations doesn't significantly decrease the entropy because the remaining populations do not significantly overlap. We can fit a two-component linear regression to this plot to automatically identify the location of the changpoint and plot the corresponding model.

The changepoint corresponds to the model with 16 populations. We can plot this model to see how it looks.

We can plot the populations identified by entropy based metaclustering.

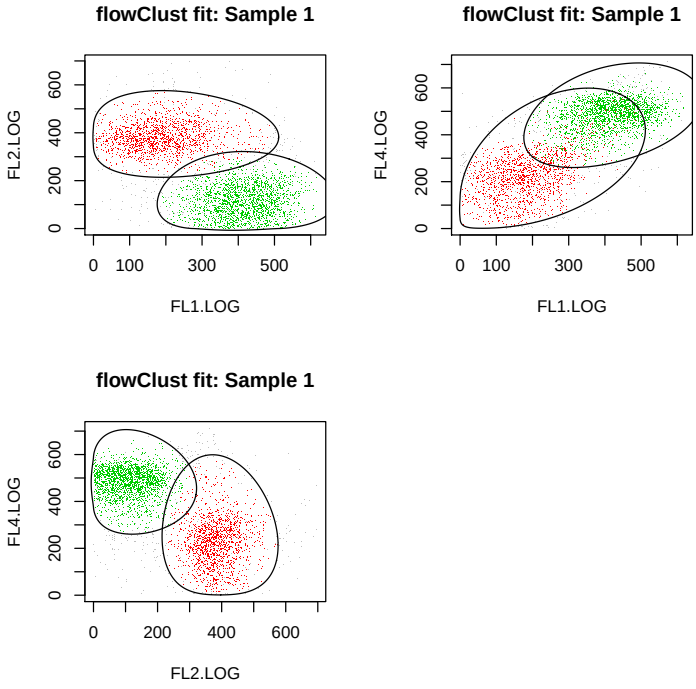
```
plot(meta2)
```



There are four major populations with the remaining 12 populations corresponding to various outlier points.

```
print(plot(m[[1]], pch = ".", new = T,
           main = "flowClust fit: Sample 1"))
```

```
NULL
```



Ongoing Work

Work on the metaclustering package is ongoing. We hope to have development version ready for submission in the near future. We are in the process of developing greater depth and functionality for the package.

Feature Requests

- Integrate `flowMerge` more closely with `workFlows`.
- Add explicit parallelization support
- Expand plotting and graphing capabilities
- ...

flowMetaCluster

- Add support for user provided distance matrix.
- Add more robust support for labeling of cell populatons.
- Entropy-based merging: map metacluster labelled events back to original populations.

- Implement bayesian priors for rare cell populations.
- ...

Selected References

- *Merging Mixture Components for Cell Population Identification in Flow Cytometry* Greg Finak, Ali Bashashati, Ryan Brinkman, Raphael Gottardo. Adv. In Bioinformatics.
- *Automated Gating of Flow Cytometry Data via Robust Model-Based Clustering* Kenneth Lo, Ryan Brinkman, Raphael Gottardo. Cytometry A.

R Code

```
###################################################
### chunk number 1: pre
###################################################
options(prompt=" ",continue="\t",width=43);


###################################################
### chunk number 2: loadpackages
###################################################
require(flowMerge)
require(flowViz)
require(flowQ)
require(flowStats)
require(snowfall)
require(fpc);
require(flowTrack)


###################################################
### chunk number 3: loaddata
###################################################
file.loc<-system.file("extdata",package="flowTrack");
data<-read.flowSet(path=file.loc,pattern="fcs",phenoData="annotation.txt",
    transformation=FALSE)


###################################################
### chunk number 4: summary
###################################################
data
```

```
#####################################################
### chunk number 5: summary2
#####################################################
data[[1]]


#####################################################
### chunk number 6: names
#####################################################
colnames(data);
sampleNames(data);


#####################################################
### chunk number 7: phenoData2
#####################################################
pData(data)


#####################################################
### chunk number 8: renamesamples
#####################################################
sampleNames(data)<-as.character(pData(data)[,"PatientID"])


#####################################################
### chunk number 9: renamechannels
#####################################################
for(i in seq_len(length(data))){
pData(parameters(data[[i]]))[,"desc"]<-c("NA","NA","CD8","CD69",
    "CD4","CD3","HLADr","NA")
}
colnames(data)<-c("FSC","SSC","CD8","CD69","CD4","CD3","HLADr","Time")


#####################################################
### chunk number 10: densityplotcode eval=FALSE
#####################################################
## densityplot(~.,data);


#####################################################
### chunk number 11: densityplot
#####################################################
```

```
print(densityplot(~.,data));


##################################################
### chunk number 12: transform
##################################################
tData<-transform(data,transformList(colnames(data)[3:7],logicleTransform()))


##################################################
### chunk number 13: densityplot2code eval=FALSE
##################################################
## densityplot(~.,tData);


##################################################
### chunk number 14: densityplot3
##################################################
print(densityplot(~.,tData));


##################################################
### chunk number 15: scatterplotcode eval=FALSE
##################################################
## splom(tData[[2]][,c(1,2,3,5,6,7)],smooth=TRUE)


##################################################
### chunk number 16: scatterplot
##################################################
print(splom(tData[[2]][,c(1,2,3,5,6,7)],smooth=TRUE));


##################################################
### chunk number 17: boundaryfilter
##################################################
tData.f<-Subset(tData,filter(tData,boundaryFilter(c("FSC","SSC"))))


##################################################
### chunk number 18: scatterplot2code eval=FALSE
##################################################
## splom(tData.f[[2]][,c(1,2,3,5,6,7)],smooth=TRUE)


##################################################
```

```
### chunk number 19: scatterplot2
###################################################
print(splom(tData.f[[2]][,c(1,2,3,5,6,7)],smooth=TRUE))


###################################################
### chunk number 20: opts
###################################################
options(width=40);


###################################################
### chunk number 21: normalization
###################################################
tData.n<-normalize(tData.f,normalization(parameters=colnames(tData.f)[3:7],
    normFun=function(x,parameters,...)warpSet(x,parameters,...)))


###################################################
### chunk number 22: normplotcode eval=FALSE
###################################################
## densityplot(~.,tData.n)


###################################################
### chunk number 23: normplot
###################################################
print(densityplot(~.,tData.n))


###################################################
### chunk number 24: loadsequentialGating
###################################################
data(flowMerge1);


###################################################
### chunk number 25: sequentialGating eval=FALSE
###################################################
## fc.all<-flowClust(tData.n[[1]],K=1:10,varNames=colnames(tData.n[[1]])[1:7],
    ##trans=1,nu=4,nu.est=1)
## fc.all.notrans<-flowClust(tData.n[[1]],K=1:10,varNames=colnames(tData.n[[1]])[1:7],
    ##trans=0,nu=4,nu.est=1)
## fc.sequential<-flowClust(tData.n[[1]],K=1:10,varNames=colnames(tData.n[[1]])[c(1,2)],
    ##trans=1,nu=4,nu.est=1)
## fc.sequential.notrans<-flowClust(tData.n[[1]],K=1:10,varNames=colnames(tData.n[[1]])
```

```
    ##[c(1,2)],trans=0,nu=4,nu.est=1)


#####################################################
### chunk number 26: bicplots
#####################################################
par(mfrow=c(2,2));
plot(BIC(fc.all),main="All at Once + Transformation",type="o");
plot(BIC(fc.all.notrans),main="All at Once",type="o");
plot(BIC(fc.sequential),main="FSC v SSC + Transformation",type="o");
plot(BIC(fc.sequential.notrans),main="FSC v SSC",type="o");


#####################################################
### chunk number 27: bestfit
#####################################################
fc.all<-fc.all[[which.max(BIC(fc.all))]]
fc.all.notrans<-fc.all.notrans[[which.max(BIC(fc.all.notrans))]]
fc.sequential.notrans<-fc.sequential.notrans[[which.max(BIC(fc.sequential.notrans))]]
fc.sequential<-fc.sequential[[which.max(BIC(fc.sequential))]]


#####################################################
### chunk number 28: mkflowobj
#####################################################
fc.all<-flowObj(fc.all,tData.n[[1]])
fc.all.notrans<-flowObj(fc.all.notrans,tData.n[[1]])
fc.sequential<-flowObj(fc.sequential,tData.n[[1]])
fc.sequential.notrans<-flowObj(fc.sequential.notrans,tData.n[[1]])


#####################################################
### chunk number 29: plot2 eval=FALSE
#####################################################
## par(mfrow=c(2,2));
## plot(as(fc.all,"flowClust"),data=tData.n[[1]],subset=
    ##c("CD3","CD8"),pch=20,main="All at once + transformation");
## plot(as(fc.all.notrans,"flowClust"),data=tData.n[[1]],
    ##subset=c("CD3","CD8"),pch=20,main="All at once");
## plot(as(fc.sequential,"flowClust"),data=tData.n[[1]],
    ##pch=20,main="FSC v SSC + transformation");
## plot(as(fc.sequential.notrans,"flowClust"),data=tData.n[[1]],
    ##pch=20,main="FSC v SSC");


#####################################################
```

```
### chunk number 30: plot1
###################################################
par(mfrow=c(2,2));
plot(as(fc.all,"flowClust"),data=tData.n[[1]],subset=c("CD3","CD8"),
    pch=20,main="All at once + transformation");
plot(as(fc.all.notrans,"flowClust"),data=tData.n[[1]],subset=c("CD3","CD8"),
    pch=20,main="All at once");
plot(as(fc.sequential,"flowClust"),data=tData.n[[1]],pch=20,
    main="FSC v SSC + transformation");
plot(as(fc.sequential.notrans,"flowClust"),data=tData.n[[1]],
    pch=20,main="FSC v SSC");


###################################################
### chunk number 31: flowMergeObjects
###################################################
getSlots("flowMerge")


###################################################
### chunk number 32: flowMerge
###################################################
m1<-merge(fc.sequential.notrans);
m2<-merge(fc.sequential);


###################################################
### chunk number 33: opts
###################################################
options(width=40);


###################################################
### chunk number 34: fitPiecewiseLinreg
###################################################
par(mfrow=c(2,2));
fitPiecewiseLinreg(m1,plot=T,normalized=TRUE,);
fitPiecewiseLinreg(m1,plot=T,normalized=FALSE);
fitPiecewiseLinreg(m2,plot=T,normalized=TRUE);
fitPiecewiseLinreg(m2,plot=T,normalized=FALSE);


###################################################
### chunk number 35: opts
###################################################
options(width=40);
```

```
#####################################################
### chunk number 36: plotmerged
#####################################################
par(mfrow=c(2,2));
plot(m1[[3]]);
plot(m2[[3]]);
plot(m2[[2]]);


#####################################################
### chunk number 37: rm2
#####################################################
rm(m1);


#####################################################
### chunk number 38: extract
#####################################################
m<-m2[[fitPiecewiseLinreg(m2,normalize=FALSE)]];
fl<-split(f=m);


#####################################################
### chunk number 39: ruleOutliers
#####################################################
ruleOutliers(m);
dim(split(f=m)[[1]])
ruleOutliers(m)<-list(level=0.99);
dim(split(f=m)[[1]]);


#####################################################
### chunk number 40: flplots3 eval=FALSE
#####################################################
## splom(fl[[1]][,c(1,2,3,5,6,7)]);


#####################################################
### chunk number 41: flplots4
#####################################################
print(splom(fl[[1]][,c(1,2,3,5,6,7)]));


#####################################################
```

```
### chunk number 42: flplotsbb eval=FALSE
##################################################
## print(splom(fl[[2]][,c(1,2,3,5,6,7)]));


##################################################
### chunk number 43: flplotsb
##################################################
print(splom(fl[[2]][,c(1,2,3,5,6,7)]));


##################################################
### chunk number 44: secondstage eval=FALSE
##################################################
## fl.trans<-flowClust(fl[[1]],varNames=colnames(fl[[1]])[c(3,5,6)],
    ##K=1:10,trans=1,nu=Inf,nu.est=0)
## fl.notrans<-flowClust(fl[[1]],varNames=colnames(fl[[1]])[c(3,5,6)],
    ##K=1:10,trans=0,nu=Inf,nu.est=0)


##################################################
### chunk number 45: loadstage2data
##################################################
data(flowMerge2)


##################################################
### chunk number 46: bicplot
##################################################
par(mfrow=c(1,2));
plot(BIC(fl.trans),type="o",main="BIC, flowClust\nmodels with transformation")
plot(BIC(fl.notrans),type="o",main="BIC, flowClust\nmodels without transformation")


##################################################
### chunk number 47: flplot
##################################################
plot(flowObj(fl.trans[[4]],
fl[[1]]),pch=20,new=T,
main="With transformation, K=4")


##################################################
### chunk number 48: flplot2
##################################################
plot(flowObj(fl.notrans[[4]],
```

```
fl[[1]]),pch=20,new=T,
main="Without Transformation, K=4")


#####################################################
### chunk number 49: flplot3
#####################################################
plot(flowObj(fl.notrans[[5]],fl[[1]]),pch=20,new=T,main="Without Transformation, K=5")


#####################################################
### chunk number 50: flowObj
#####################################################
fl.2<-flowObj(fl.notrans[[5]],fl[[1]])
ruleOutliers(fl.2)<-list(level=0.99);
order(fl.2@mu[,1],fl.2@mu[,3],decreasing=TRUE)[1]
order(fl.2@mu[,2],fl.2@mu[,3],decreasing=TRUE)[1]


#####################################################
### chunk number 51: split2
#####################################################
hladr<-split(x=getData(fl.2),f=as(fl.2,"flowClust"),
    population=list(CD8CD3=order(fl.2@mu[,1],fl.2@mu[,3],decreasing=TRUE)[1]
    ,CD4CD3=order(fl.2@mu[,2],fl.2@mu[,3],decreasing=TRUE)[1]))
print(densityplot(~CD8+CD4+HLADr,as(hladr,"flowSet")));


#####################################################
### chunk number 52: rangegate
#####################################################
hladr.cd3.cd4<-Subset(hladr[[1]],rangeGate(hladr[[1]],
    stain="HLADr", plot=FALSE,alpha=0.5, filterId="CD3+CD4+HLAct") )
hladr.cd3.cd8<-Subset(hladr[[2]],rangeGate(hladr[[2]],
    stain="HLADr", plot=FALSE,alpha=0.5, filterId="CD3+CD8+HLAct") )
100*dim(hladr.cd3.cd4)[1]/dim((data[[1]]))[1]
100*dim(hladr.cd3.cd8)[1]/dim((data[[1]]))[1]


#####################################################
### chunk number 53: loaddata
#####################################################
file.loc<-system.file("extdata",package="flowTrack");
data<-read.flowSet(path=file.loc,pattern="fcs",
    phenoData="annotation.txt",transformation=FALSE)
sampleNames(data)<-as.character(pData(data)[,"PatientID"])
```

```
for(i in seq_len(length(data))){
pData(parameters(data[[i]]))[,"desc"]<-c("NA","NA","CD8","CD69","CD4","CD3","HLADr","NA")
}
colnames(data)<-c("FSC","SSC","CD8","CD69","CD4","CD3","HLADr","Time")
tData<-transform(data,transformList(colnames(data)[3:7],logicleTransform()))
tData.f<-Subset(tData,filter(tData,boundaryFilter(c("FSC","SSC"))))
tData.n<-normalize(tData.f,normalization(parameters=colnames(tData.f)[3:7],
    normFun=function(x,parameters,...)warpSet(x,parameters,...)))


#######################################################
### chunk number 54: snow
#######################################################
sfInit(parallel=TRUE,cpus=4);


#######################################################
### chunk number 55: autogatingsnow eval=FALSE
#######################################################
## result<-fsApply(tData.n,function(x)flowClust(x,K=1:10,trans=1,nu=4,
    ##nu.est=1,varNames=colnames(x)[1:2]))
## sfStop();


#######################################################
### chunk number 56: loadautogating
#######################################################
data(flowMerge3)


#######################################################
### chunk number 57: maxbic
#######################################################
result<-lapply(result,function(x)x[[which.max(BIC(x))]]);
result<-sapply(1:length(data),function(i)flowObj(result[[i]],tData.n[[i]]))
result<-lapply(result,function(x){m<-merge(x);m[[fitPiecewiseLinreg(m)]]})


#######################################################
### chunk number 58: semiautogating eval=FALSE
#######################################################
## indices<-unlist(lapply(result,function(x){plot(x);
    ##z<-unlist(locator(n=1));inc<-which.min(sapply(1:x@K,
    ##function(i)mahalanobis(box(z,lambda=x@lambda),x@mu[i,1:2],
    ##x@sigma[i,,])));plot(x,include=inc);inc}))
```

```
#####################################################
### chunk number 59: loadindices
#####################################################
data(flowMergeIndices)


#####################################################
### chunk number 60: split
#####################################################
result.split<-sapply(1:length(result),function(i)split(f=result[[i]])[[indices[i]]])


#####################################################
### chunk number 61: parallelflowClust eval=FALSE
#####################################################
## sfInit(parallel=TRUE,cpus=2);
## result.fl<-lapply(result.split,function(x)flowClust(x,B.init=100,
    ##K=1:10,varNames=colnames(x)[c(3,5,6)],nu=Inf,nu.est=0,trans=2))
## sfStop();


#####################################################
### chunk number 62: loaddata4
#####################################################
data(flowMerge4)


#####################################################
### chunk number 63: flhighthrpt
#####################################################
result.fl.bic<-sapply(1:length(result.fl),function(i)
    flowObj(result.fl[[i]][[which.max(BIC(result.fl[[i]]))]],result.split[[i]]))


#####################################################
### chunk number 64: manual
#####################################################
plot(result.fl.bic[[1]],pch=20);
result.fl.bic[[1]]<-flowObj(result.fl[[1]][[5]],result.split[[1]]);
plot(result.fl.bic[[1]],pch=20);


#####################################################
### chunk number 65: extract2
#####################################################
```

```
fl.split.cd8<-sapply(1:length(result.fl.bic),function(i)
    split(x=result.split[[i]],f=as(result.fl.bic[[i]],"flowClust"),
    population=list(order(result.fl.bic[[i]]@mu[,1],result.fl.bic[[i]]@mu[,3],
    decreasing=TRUE)[1]))
fl.split.cd4<-sapply(1:length(result.fl.bic),function(i)
    split(x=result.split[[i]],f=as(result.fl.bic[[i]],"flowClust"),
    population=list(order(result.fl.bic[[i]]@mu[,2],result.fl.bic[[i]]@mu[,3],
    decreasing=TRUE)[1]))


#####################################################
### chunk number 66: rangeGate
#####################################################
all.hladr.cd3.cd8<-lapply(fl.split.cd8,function(x)
    Subset(x,rangeGate(x, stain="HLADr", plot=FALSE,borderQuant=0.5,
    alpha=0.5, filterId="CD3+CD8+HLAct") ))
all.hladr.cd3.cd4<-lapply(fl.split.cd4,function(x)
    Subset(x,rangeGate(x, stain="HLADr", plot=FALSE,borderQuant=0.5,
    alpha=0.5, filterId="CD3+CD4+HLAct") ))


#####################################################
### chunk number 67: hladrproportions
#####################################################
CD4.hladr<-sapply(1:length(all.hladr.cd3.cd4),
    function(i)100*dim(all.hladr.cd3.cd4[[i]])[1]/dim(data[[i]])[1])
CD8.hladr<-sapply(1:length(all.hladr.cd3.cd8),
    function(i)100*dim(all.hladr.cd3.cd8[[i]])[1]/dim(data[[i]])[1])


#####################################################
### chunk number 68: barchart
#####################################################
print(barchart(reorder(PatientID,as.numeric(factor(GroupID)))~pr,
    data=data.frame(pr=as.vector(CD4.hladr),pData(data)
    [,c("GroupID","PatientID")]),groups=GroupID,auto.key=T))


#####################################################
### chunk number 69: barchart2
#####################################################
print(barchart(reorder(PatientID,as.numeric(factor(GroupID)))~pr,data=data.frame(
pr=as.vector(CD8.hladr),pData(data)[,c("GroupID","PatientID")]),groups=GroupID,auto.key=T))


#####################################################
```

```
### chunk number 70: loadlymphdata
####################################################
require(flowMetaCluster)
require(snowfall)
data(lymphoma)
```