flowMatch: Cell population matching and meta-clustering in Flow Cytometry

Ariful Azad, Alex Pothen

October 24, 2025

aazad@purdue.edu

Contents

1 Licensing

Under the Artistic License, you are free to use and redistribute this software for academic and personal use.

2 Overview

The *flowMatch* package performs two major functions given a collection of flow cytometry (FC) samples:

1. Match cell populations across FC samples
2. Compute meta-clusters and templates from a collection of FC samples

2.1 FC sample

A flow cytometry sample measuring p features for n cells is represented with an $n \times p$ matrix A. The (i, j) entry of the matrix, $A(i, j)$, represents the measurement of the j^{th} feature in the i^{th} cell. We characterize a multi-parametric sample with a finite mixture model of multivariate normal distributions, where each component is a cluster of cells expressing similar phenotypes in the measured parameter space. Such a cluster of cells represents a particular cell type and is called a *cell population* in cytometry. In the mixture model, a cell population (cluster) is characterized by a multi-dimensional normal distribution and is represented by two parameters μ, the p dimensional mean vector, and Σ, the $p \times p$ covariance matrix [5].

2.2 Population matching

Registering cell populations and tracking their changes across samples often reveal the biological conditions the samples are subjected to. To study these cross-condition changes we first establish the correspondence among cell populations by matching clusters across FC samples. We used a robust variant of matching algorithm called the mixed edge cover (MEC) algorithm that allows cell cluster from one sample to get matched to zero or more clusters in another sample [2]. MEC algorithm covers possible circumstances when a cell population in one sample is absent from another sample, or when a cell population in one sample splits into two or more cell populations in a second sample, which can happen due to biological reasons or due to the limitations of clustering methods.

2.3 Meta-clustering and construction of templates

In high throughput flow cytometry, large cohorts of samples belonging to some representative classes are produced. Different classes usually represent multiple experiment conditions, disease status, time points etc. In this setting, samples belonging to same class can be summarized by a *template*, which is

a summary of the sample's expression pattern [1, 3, 6]. The concept of cell populations in a sample can be extended to *meta-clusters* in a collection of similar samples, representing generic cell populations that appear in each sample with some sample-specific variation. Each meta-cluster is formed by combining cell populations expressing similar phenotypes in different samples. Hence mathematically a meta-cluster is characterized by a normal distribution, with parameters computed from the distributions of the clusters included in it. Clusters in a meta-cluster represent the same type of cells and thus have overlapping distributions in the marker space.

A *template* is a collection of relatively homogeneous meta-clusters commonly shared across samples of a given class, thus describing the key immune-phenotypes of an overall class of samples in a formal, yet robust, manner. Mathematically a template is characterized by a finite mixture of normal distributions. We summarize these concepts in Table 1 and in Figure 1. Given the inter-sample variations, a few templates can together concisely represent a large cohort of samples by emphasizing all the major characteristics while hiding unnecessary details. Thereby, overall changes across multiple conditions can be determined rigorously by comparing just the cleaner class templates rather than the noisy samples themselves [1, 6].

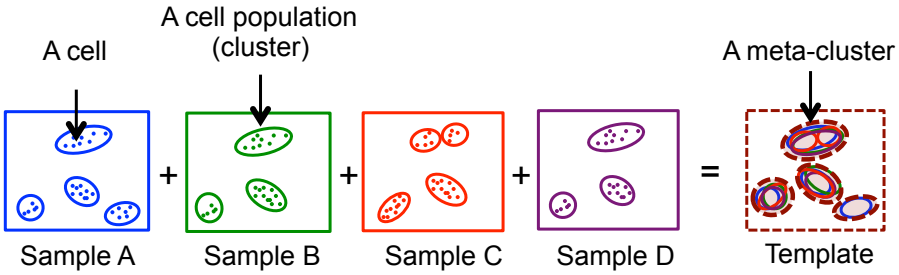


Figure 1: Summary of terminology used in this package.

We build templates from a collection of samples by a hierarchical algorithm that repeatedly merges the most similar pair of samples or partial templates obtained by the algorithm thus far. The algorithm builds a binary tree called the *template tree* denoting the hierarchical relationships among the samples. A leaf node of the template tree represents a sample and an internal (non-leaf) node represents a template created from the samples. Fig. 2 shows an example of a template tree created from four hypothetical samples, S_1, S_2, S_3, and S_4. An internal node in the template tree is created by matching similar cell clusters across the two children and merging the matched clusters into meta-clusters. For example, the internal node $T(S_1, S_2)$ in Fig. 2 denotes the template from samples S_1 and S_2. The mean vector and covariance matrix of a meta-cluster are computed from the means and covariance matrices of the clusters participating in the meta-cluster.

Terms	meaning
Cell population (cluster)	a group of cells expressing similar features, e.g., helper T cells, B cells
Sample	a collection of cell populations within a single biological sample
Meta-cluster	a set of biologically similar cell clusters from different samples
Template	a collection of meta-clusters from samples of same class

Table 1: Summary of terminology used in this package.

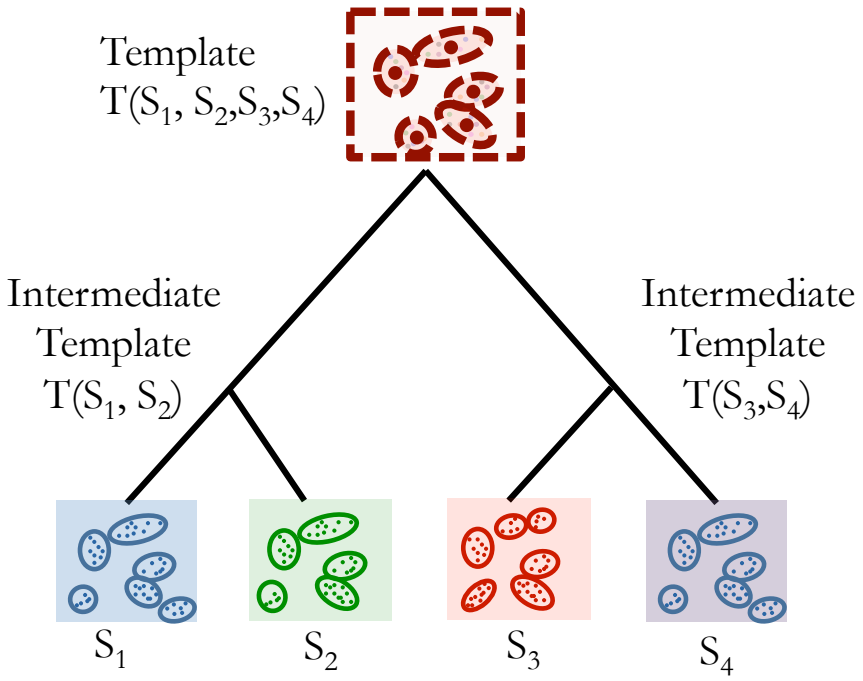


Figure 2: An example of a hierarchical template tree created from four hypothetical samples S_1, S_2, S_3 and S_4. A leaf node of the template tree represents a sample and an internal node represents a template created from its children.

2.4 Related packages in Bioconductor

Several packages are available in Bioconductor (http://www.bioconductor.org/) for analyzing flow cytometry data. The `flowCore` package provides basic structures for flow cytometry data. A number of packages are available for clustering or automated gating in a FC samples such as `flowClust`/`flowMerge` and `flowMeans`. Given an FC samples these packages identify cell populations (cell clusters) in the sample.

The `flowMatch` package starts working with the output of clustering/gating results. Given a pair of FC sample, `flowMatch` registers corresponding populations across the sample pair by using a combinatorial algorithm called the mixed edge cover [2]. In addition to registering populations, the `flowMatch` package merges corresponding clusters across samples to build meta-clusters. A meta-cluster represents the core pattern of a particular cell population across a large collection of samples. The collection of meta-clusters are then grouped together to build templates for a collection of similar samples. Thus, the `flowMatch` package works in a higher level than the other packages such as `flowClust`/`flowMerge`, `flowMeans`, `flowQ`, `flowTrans`, etc.

The only package related to this package is `flowMap` that also matches cell population across samples. However, `flowMap` uses the nonparametric Friedman-Rafsky (FR) multivariate run test to compute the mapping of clusters. By contrast, `flowMatch` uses Mahalanobis distance or Kullback-Leibler divergence to compute cluster dissimilarity and then applies a combinatorial algorithm to match clusters. Additionally, `flowMatch` performs meta-clustering and creates templates, which are not performed by `flowMap`. `FLAME` (not a Bioconductor package) by Pyne et al. provides funtionalities similar to `flowMatch`. The differences between these two approaches are discussed in [1].

2.5 Dataset for testing

In order to reduce the download size of `flowMatch`, I put an example dataset to a Bioconductor data package (`healthyFlowData`). The data package contains a dataset consisting of 20 FC samples where peripheral blood mononuclear cells (PBMC) were collected from four healthy individuals. Each sample was divided into five repplicates and each replicate was stained using labeled antibodies against CD45, CD3, CD4, CD8, and CD19 protein markers. Therefore we have total 20 samples from four healthy subjects. This is a part of a larger dataset of 65 samples.

The `healthyFlowData` package can be downloaded in the usual way.

```
> if (!requireNamespace("BiocManager", quietly=TRUE))
+     install.packages("BiocManager")
> BiocManager::install("healthyFlowData")
```

To use the examples included in this package, we must load the `flowMatch` and `healthyFlowData` packages:

```
> library(healthyFlowData)
> library(flowMatch)
```

3 Data structures

We summarized the concept of cluster, sample, meta-cluster and template in Table 1 and in Figure 1. In this package we represent these terms with four S4 classes. Additionally we represent matching of clusters across a pair of sample with another S4 class. We describe the classes in Table 2. Details about this classes will be discussed in their related sections.

Terms	S4 class
Cell population (cluster)	*Cluster*
Sample	*ClusteredSample*
Cluster matching	*ClusterMatch*
Meta-cluster	*MetaCluster*
Template	*Template*

Table 2: S4 classes used in this package.

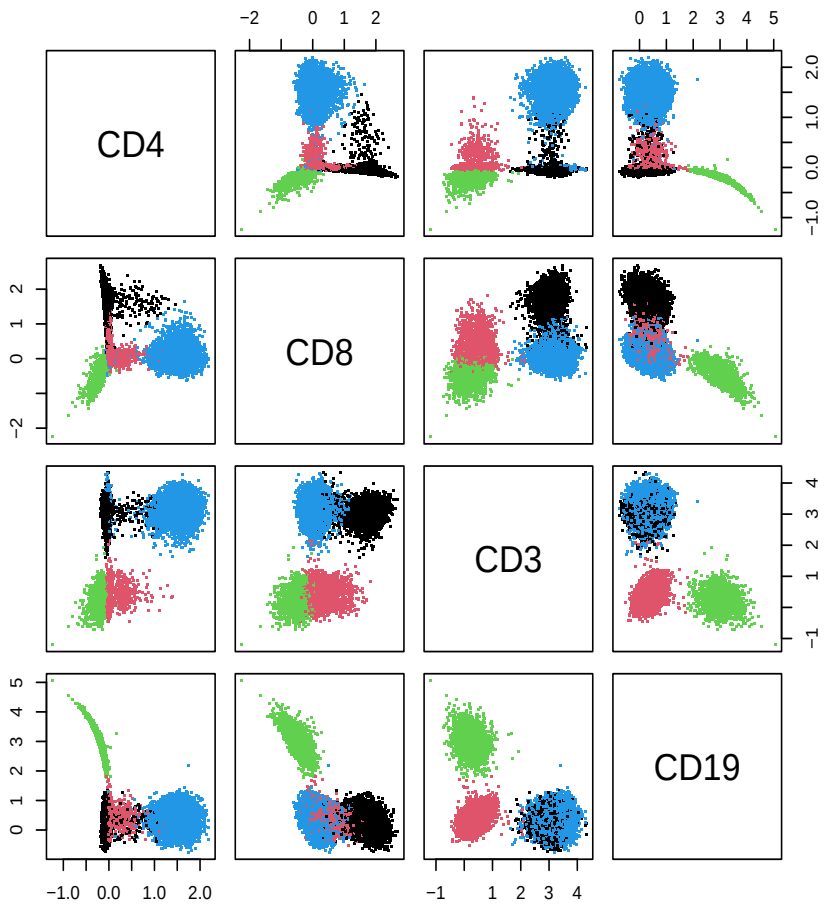
4 Population identification by using clustering algorithms

Since *flowMatch* package can work with any clustering algorithm, we did not include any clustering algorithm in this package.

We first identify cell populations in each sample by using any suitable clustering algorithm. We then create an object of class *ClusteredSample* to encapsulate all necessary information about cell populations in a sample. An object of class *ClusteredSample* stores a list of clusters (objects of class *Cluster*) and other necessary parameters. Since we characterize a sample with a finite mixture of normal distribution, the user can supply `centers` or `cov` of the clusters estimated by methods of their choice. When `centers` or `cov` of the clusters are not provided by user, they are estimated from the FC sample. The center of a cluster is estimated with the mean of points present in the cluster. An unbiased estimator of covariance is estimated using function `cov` from `stats` package.

```
> ## ------------------------------------------------
> ## load data and retrieve a sample
> ## ------------------------------------------------
>
> data(hd)
```

```
> sample = exprs(hd.flowSet[[1]])
> ## ------------------------------------------------
> ## cluster sample using kmeans algorithm
> ## ------------------------------------------------
> km = kmeans(sample, centers=4, nstart=20)
> cluster.labels = km$cluster
> ## ------------------------------------------------
> ## Create ClusteredSample object  (Option 1 )
> ## without specifying centers and covs
> ## we need to pass FC sample for paramter estimation
> ## ------------------------------------------------
>
> clustSample = ClusteredSample(labels=cluster.labels, sample=sample)
> ## ------------------------------------------------
> ## Create ClusteredSample object  (Option 2)
> ## specifying centers and covs
> ## no need to pass the sample
> ## ------------------------------------------------
>
> centers = list()
> covs = list()
> num.clusters = nrow(km$centers)
> for(i in 1:num.clusters)
+ {
+   centers[[i]] = km$centers[i,]
+   covs[[i]] = cov(sample[cluster.labels==i,])
+ }
> # Now we do not need to pass sample
> clustSample = ClusteredSample(labels=cluster.labels, centers=centers, covs=covs)
> ## ------------------------------------------------
> ## Show summary and plot a clustered sample
> ## ------------------------------------------------
>
> summary(clustSample)

An Object of class 'ClusteredSample'
Number of clusters: 4
Number of cells in cluster 1: 4399 [ 22.8 % ]
Number of cells in cluster 2: 2729 [ 14.1 % ]
Number of cells in cluster 3: 1658 [ 8.6 % ]
Number of cells in cluster 4: 10535 [ 54.5 % ]

> plot(sample, clustSample)
>
```



5 Computing distance between clusters

The mixed edge cover algorithm matches similar clusters based on a dissimilarity measure between a pair of clusters. In *flowMatch* package we included Euclidean distance, Mahalanobis distance and KL divergence for computing the dissimilarities. These distances are computed from a pair of *Cluster* objects by using their distribution parameters.

```
> ## ------------------------------------------------
> ## load data and retrieve a sample
> ## ------------------------------------------------
>
> data(hd)
```

```
> sample = exprs(hd.flowSet[[1]])
> ## ------------------------------------------------
> ## cluster sample using kmeans algorithm
> ## ------------------------------------------------
>
> km = kmeans(sample, centers=4, nstart=20)
> cluster.labels = km$cluster
> ## ------------------------------------------------
> ## Create ClusteredSample object
> ## and retrieve two clusters (cluster from different samples can be used as well)
> ## ------------------------------------------------
>
> clustSample = ClusteredSample(labels=cluster.labels, sample=sample)
> clust1 = get.clusters(clustSample)[[1]]
> clust2 = get.clusters(clustSample)[[2]]
> ## ------------------------------------------------
> ## compute dissimilarity between the clusters
> ## ------------------------------------------------
>
> dist.cluster(clust1, clust2, dist.type='Mahalanobis')

[1] 7.658469

> dist.cluster(clust1, clust2, dist.type='KL')

[1] 55.5178

> dist.cluster(clust1, clust2, dist.type='Euclidean')

[1] 2.711378

>
```

6 Matching cell clusters across a pair of samples

Given a pair of *ClusteredSample* objects we match clusters by using the MEC algorithm [2]. MEC algorithm allows a cluster to get matched to zero, one or more than one clusters from another sample. The penalty for leaving a cluster unmatched is empirically selected, see [2] for a discussion. When `unmatch.penalty` is set to a very large value every cluster get matched.

```
> ## ------------------------------------------------
> ## load data and retrieve two samples
> ## ------------------------------------------------
>
> data(hd)
> sample1 = exprs(hd.flowSet[[1]])
```

```
> sample2 = exprs(hd.flowSet[[2]])
> ## ------------------------------------------------
> ## cluster samples using kmeans algorithm
> ## ------------------------------------------------
>
> clust1 = kmeans(sample1, centers=4, nstart=20)
> clust2 = kmeans(sample2, centers=4, nstart=20)
> cluster.labels1 = clust1$cluster
> cluster.labels2 = clust2$cluster
> ## ------------------------------------------------
> ## Create ClusteredSample objects
> ## ------------------------------------------------
>
> clustSample1 = ClusteredSample(labels=cluster.labels1, sample=sample1)
> clustSample2 = ClusteredSample(labels=cluster.labels2, sample=sample2)
> ## ------------------------------------------------
> ## Computing matching of clusteres
> ## An object of class "ClusterMatch" is returned
> ## ------------------------------------------------
>
> mec = match.clusters(clustSample1, clustSample2, dist.type="Mahalanobis", unmatch.penalty=
> class(mec)

[1] "ClusterMatch"
attr(,"package")
[1] "flowMatch"

> summary(mec)

============================================================
clusters/meta-clusters      matched clusters/meta-clusters
from sample1/template1      sample2/template2
============================================================
           1                            2
           2                            4
           3                            3
           4                            1
============================================================
```

7 Computing template from a collection of samples

We now build a template by merging corresponding clusters from different samples of a class. A template is constructed by repeatedly matching clusters across a pair of samples and merging the matched clusters into meta-cluster. The algorithm is similar in spirit to the UPGMA algorithm from phylogenetics and

the hierarchy of the samples can be visualized by a dendrogram. Note that, the samples in the attached dataset are from four subjects each of them is replicated five times. The template tree preserves this structure by maintaining four well separated branches.

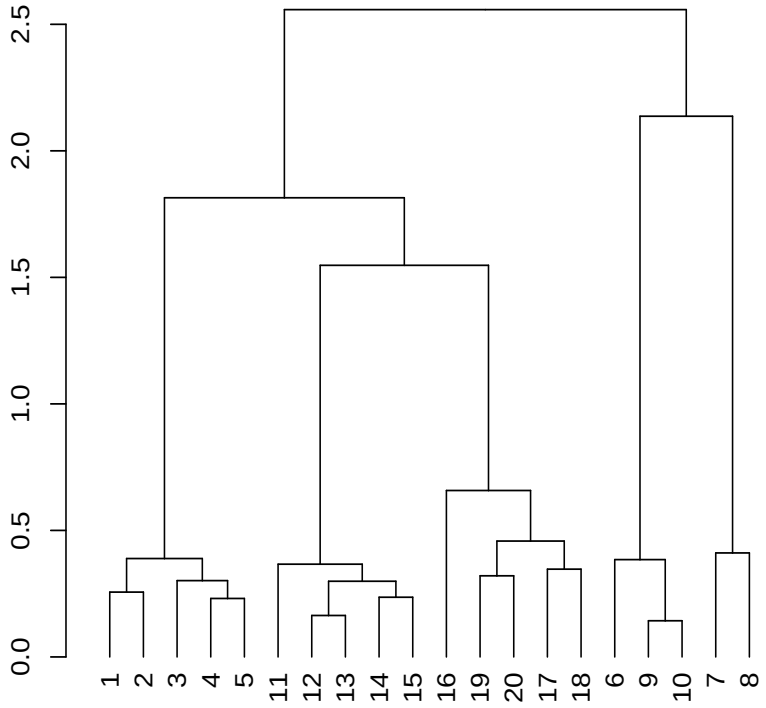
```
> ## load data (20 samples in total)
> ## ------------------------------------------------
>
> data(hd)
> ## ------------------------------------------------
> ## Retrieve each sample, clsuter it and store the
> ## clustered samples in a list
> ## ------------------------------------------------
> set.seed(1234) # for reproducable clustering
> cat('Clustering samples: ')

Clustering samples:

> clustSamples = list()
> for(i in 1:length(hd.flowSet))
+ {
+   cat(i, ' ')
+   sample1 = exprs(hd.flowSet[[i]])
+   clust1 = kmeans(sample1, centers=4, nstart=20)
+   cluster.labels1 = clust1$cluster
+   clustSample1 = ClusteredSample(labels=cluster.labels1, sample=sample1)
+   clustSamples = c(clustSamples, clustSample1)
+ }

1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20

> ## ------------------------------------------------
> ## Create a template from the list of clustered samples
> ## the function returns an object of class "Template"
> ## ------------------------------------------------
>
> template = create.template(clustSamples)

Number of Meta Cluster = 4
mc1 = [1 1; 2 1; 3 1; 4 3; 5 1; 6 2; 7 2; 8 4; 9 1; 10 3; 11 4; 12 1; 13 4; 14 3; 15 4; 16 1
mc2 = [1 2; 2 2; 3 3; 4 2; 5 2; 6 3; 7 1; 8 3; 9 3; 10 2; 11 1; 12 2; 13 1; 14 4; 15 3; 16 2
mc3 = [1 3; 2 4; 3 4; 4 1; 5 4; 6 1; 7 3; 8 2; 9 4; 10 4; 11 3; 12 4; 13 3; 14 2; 15 1; 16 3
mc4 = [1 4; 2 3; 3 2; 4 4; 5 3; 6 4; 7 4; 8 1; 9 2; 10 1; 11 2; 12 3; 13 2; 14 1; 15 2; 16 4

> summary(template)

An Object of class 'Template'
Number of metaclusters: 4
```

```
Number of cells in metacluster 1: 195568 [ 53.16 % ]
Number of cells in metacluster 2: 80884 [ 21.99 % ]
Number of cells in metacluster 3: 45748 [ 12.44 % ]
Number of cells in metacluster 4: 45682 [ 12.42 % ]
```

7.1 Plotting templates

All samples within a template are organized as binary tree. We can plot the hierarchy of samples established while creating a template-tree: Note that, the samples in the attached dataset are from four subjects each of them is replicated five times. The template tree preserves this structure by maintaining four well separated branches.

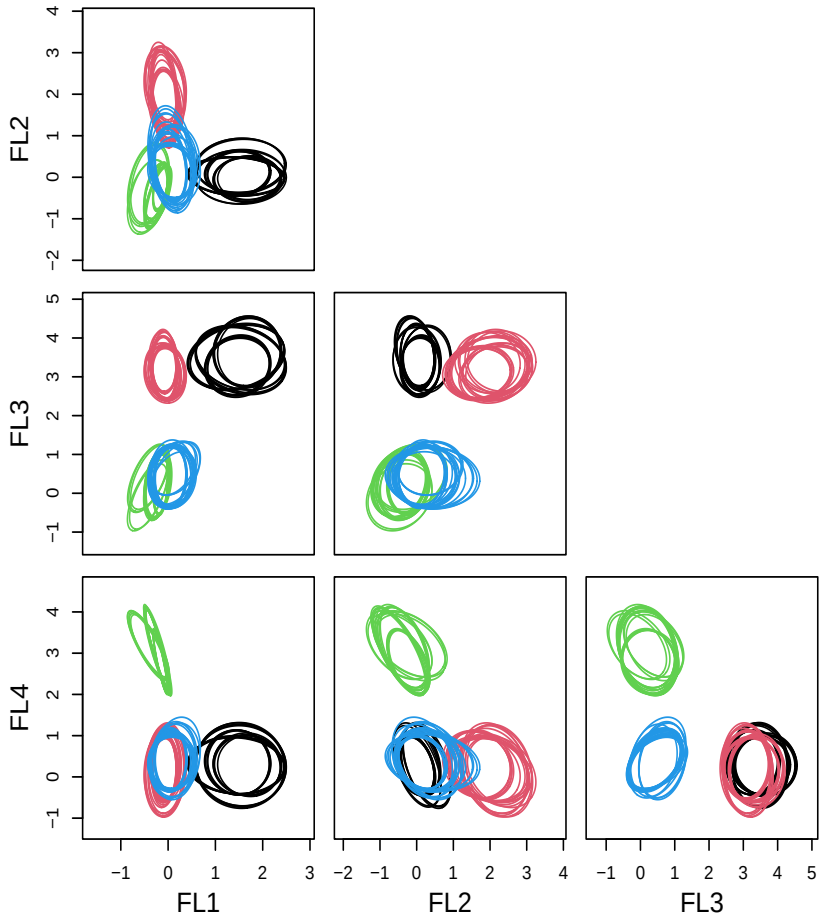
```
> template.tree(template)

Number of objects: 20
```



We plot a template as a collection of bivariate contour plots of its meta-clusters. To plot each meta-cluster we consider the clusters within the meta-cluster normally distributed and represent each cluster with an ellipsoid. The axes of an ellipsoid is estimated from the eigen values and eigen vectors of the covariance matrix of a cluster [4]. We then plot the bivariate projection of the ellipsoid as 2-D ellipses. There are several options to draw a template.

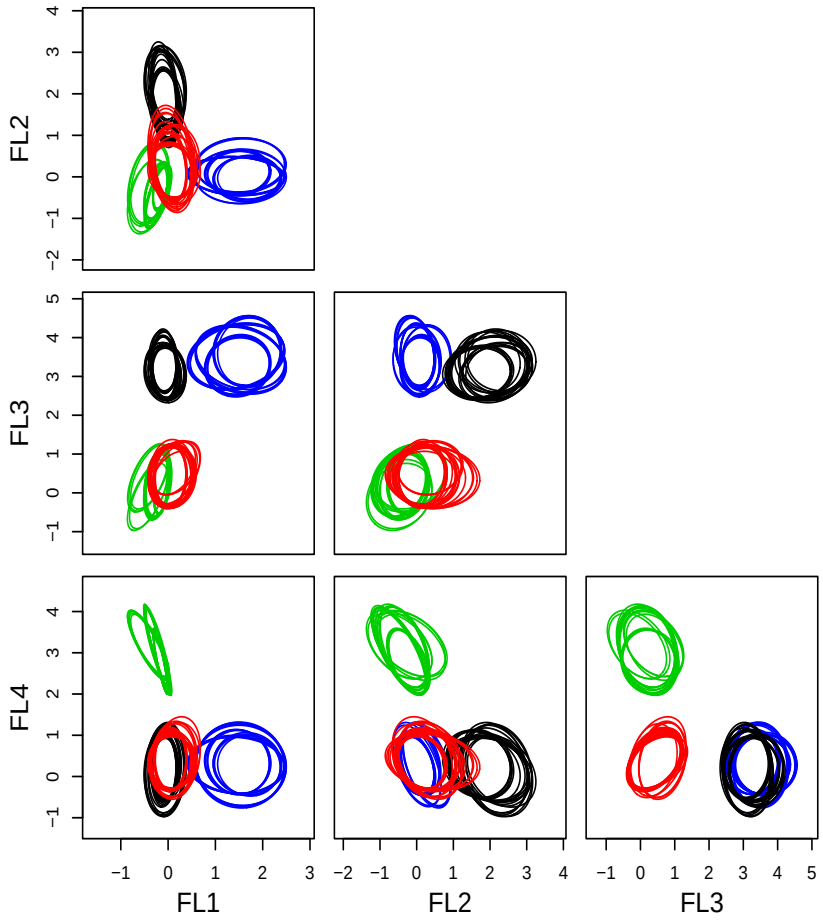
Option-1 (default): plot contours of each cluster of the meta-clusters

```
> plot(template)
```



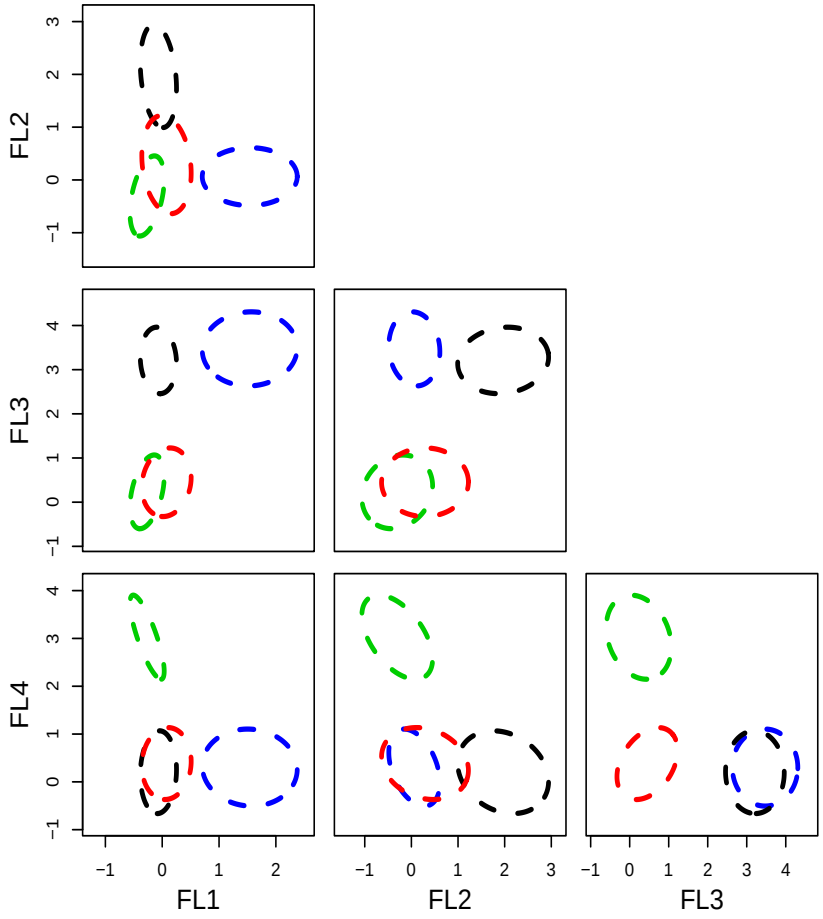
Option-2: plot contours of each cluster of the meta-clusters with defined color

```
> plot(template, color.mc=c('blue','black','green3','red'))
```



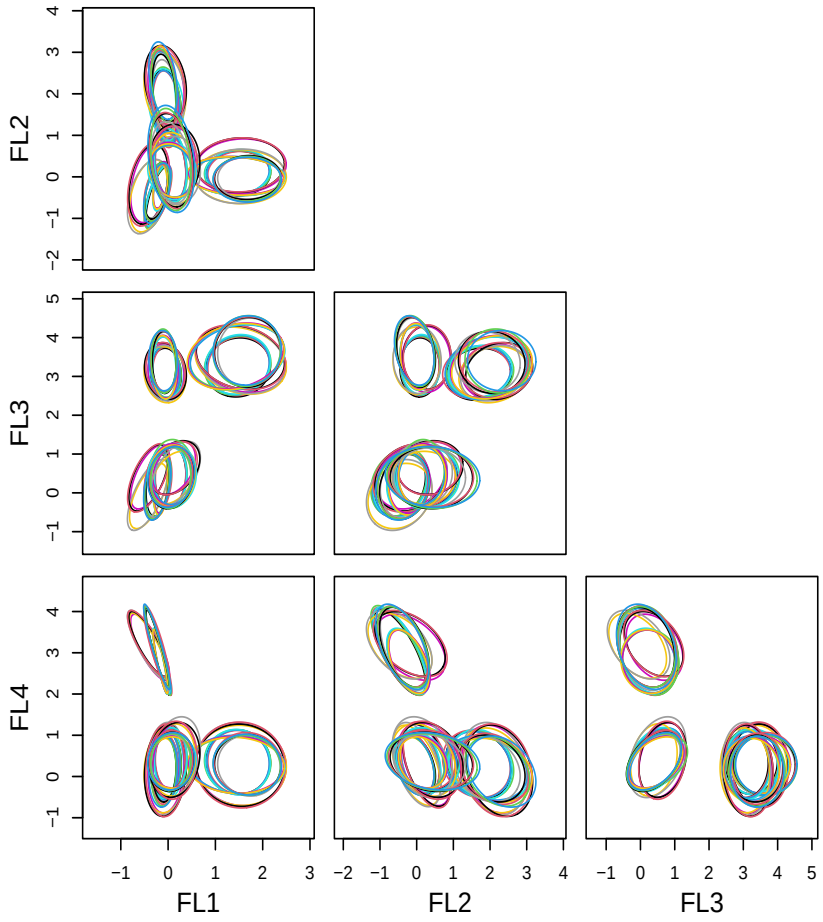
Option-3: plot contours of the meta-clusters with defined color

```
> plot(template, plot.mc=TRUE, color.mc=c('blue','black','green3','red'))
```



Option-4: plot contours of each cluster of the meta-clusters with different colors for different samples

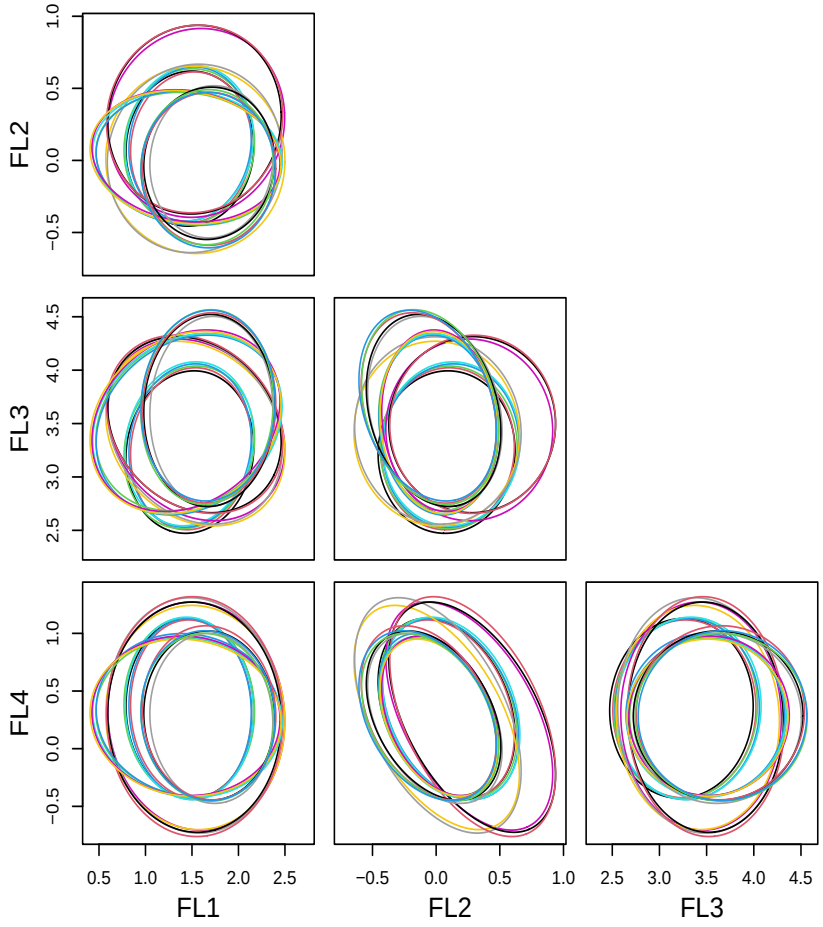
```
> plot(template, colorbysample=TRUE)
```



7.2 Retrieving and plotting a meta-cluster from a template

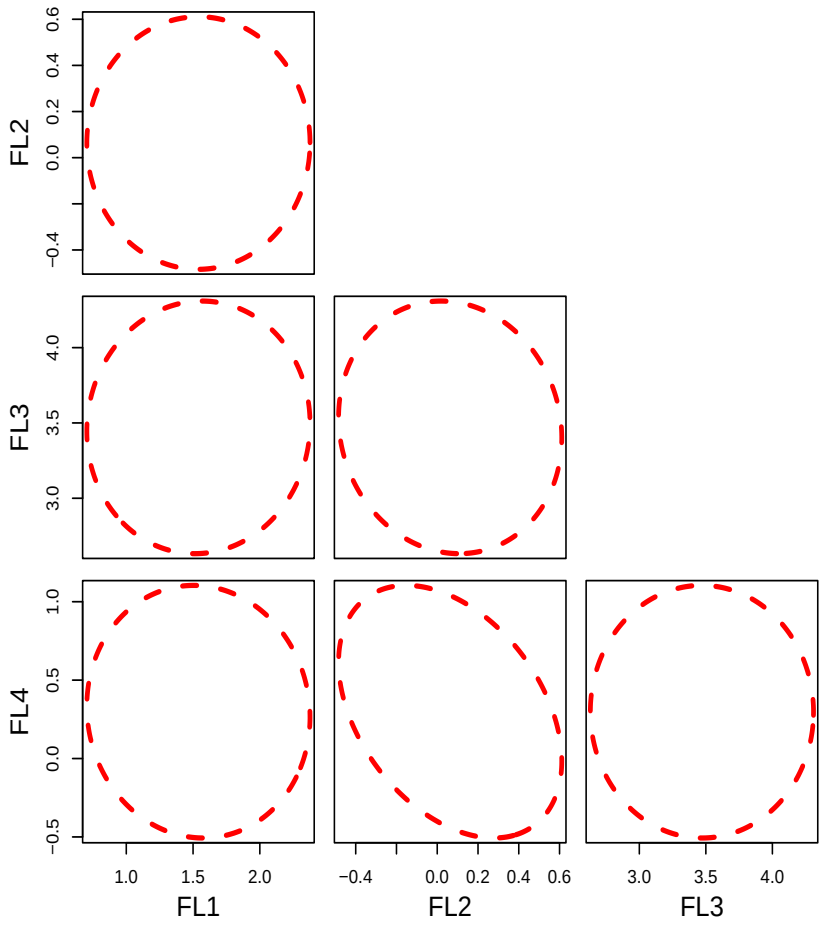
Similar to template, we plot a meta-cluster as a contour plot of the distribution of the underlying clusters or the combined meta-cluster. We consider cells in clusters or in the meta-cluster are normally distributed and represent the distribution with ellipsoid. The axes of an ellipsoid is estimated from the eigen values and eigen vectors of the covariance matrix. We then plot the bi-variate projection of the ellipsoid as 2-D ellipses.

```
> # retrieve a metacluster from a template
> mc = get.metaClusters(template)[[1]]
> summary(mc)

An Object of class 'MetaCluster' :
Number of clusters in this MetaCluster: 20
MetaCluster size = 195568

MetaCluster center:
[1] 1.54021923 0.06349614 3.47060192 0.29862616

MetaCluster covariance matrix:
              [,1]          [,2]          [,3]          [,4]
[1,]  0.1165207491  0.0001517564  0.0036705228 -0.0049142970
[2,]  0.0001517564  0.0500229560 -0.0062793598 -0.0288248427
[3,]  0.0036705228 -0.0062793598  0.1173861123 -0.0002442223
[4,] -0.0049142970 -0.0288248427 -0.0002442223  0.1082847736

> # plot all participating cluster in this meta-cluster
> plot(mc)
>
```


FL2
FL3
FL4
FL1
FL2
FL3
0.5
1.0
1.5
2.0
2.5
−0.5
0.0
0.5
1.0
2.5
3.0
3.5
4.0
4.5

We can plot the outline of the combined meta-cluster as well.

```
> plot(mc, plot.mc=TRUE)
```



References

[1] A. Azad, S. Pyne, and A. Pothen. Matching phosphorylation response patterns of antigen-receptor-stimulated T cells via flow cytometry. *BMC Bioinformatics*, 13(Suppl 2):S10, 2012.

[2] Ariful Azad, Johannes Langguth, Youhan Fang, Alan Qi, and Alex Pothen. Identifying rare cell populations in comparative flow cytometry. *Lecture Notes in Computer Science*, 6293:162–175, 2010.

[3] G. Finak, J.M. Perez, A. Weng, and R. Gottardo. Optimizing transformations for automated, high throughput analysis of flow cytometry data. *BMC Bioinformatics*, 11(1):546, 2010.

[4] Richard Arnold Johnson and Dean W Wichern. *Applied multivariate statistical analysis*, volume 5. Prentice hall Upper Saddle River, NJ, 2002.

[5] K. Lo, R.R. Brinkman, and R. Gottardo. Automated gating of flow cytometry data via robust model-based clustering. *Cytometry Part A*, 73(4):321–332, 2008.

[6] S. Pyne, X. Hu, K. Wang, E. Rossin, T.I. Lin, L.M. Maier, C. Baecher-Allan, G.J. McLachlan, P. Tamayo, D.A. Hafler, et al. Automated high-dimensional flow cytometric data analysis. *Proceedings of the National Academy of Sciences*, 106(21):8519–8524, 2009.