flowWorkspace: A Package for Importing flowJo Workspaces into R

Greg Finak <gfinak@fhcrc.org>

March 20, 2015

1 Purpose

The purpose of this package is to provide functionality to import relatively simple *flowJo* workspaces into R. By this we mean, accessing the samples, groups, transformations, compensation matrices, gates, and population statistics in the *flowJo* workspace, and replicating these using (primarily) *flowCore* functionality.

2 Why Another flowJo Workspace Import Package?

There was a need to import *flowJo* workspaces into R for comparative gating. The *flowFlowJo* package did not meet our needs. Many groups have legacy data with associated flowJo XML workspace files in version 2.0 format that they would like to access using BioConductor's tools. Hopefully this package will fill that need.

3 Support

This package supports importing of **Version 2.0 XML workspaces only**. We cannot import **.jo** files directly. You will have to save them in XML workspace format, and ensure that that format is *workspace version 2.0*. The package has been tested and works with files generated using flowJo version 9.1 on Mac OS X. XML generated by older versions of *flowJo* on windows should work as well. We do not yet support *flowJo*'s **Chimera** XML schema, though that support will be provided in the future.

The package supports import of only a subset of the features present in a flowJo workspace. The package allows importing of sample and group names, gating hierarchy, compensation matrices, data transformation functions, a subset of gates, and population counts.

BooleanGates are now supported by flowWorkspace.

4 Data Structures

The following section walks through opening and importing a flowJo workspace.

4.1 Loading the library

Simply call:

```
> library(flowWorkspace)
```

The library depends on numerous other packages, including *graph, XML, Rgraphviz, flowCore, flowViz, RBGL.*

4.2 Opening a Workspace

We represent flowJo workspaces using `flowJoWorkspace` objects. We only need to know the path to, and filename of the flowJo workspace.

```
> d<-system.file("extdata",package="flowWorkspaceData");
> wsfile<-list.files(d,pattern="A2004Analysis.xml",full=T)
```

In order to open this workspace we call:

```
> ws<-openWorkspace(wsfile)
> summary(ws)

FlowJo Workspace Version  2.0
File location:  D:/biocbld/bbs-3.0-bioc/R/library/flowWorkspaceData/extdata
File name:  A2004Analysis.xml
Workspace is open.

Groups in Workspace
         Name Num.Samples
1 All Samples           2
```

We see that this a version 2.0 workspace file. It's location and filename are printed. Additionally, you are notified that the workspace file is open. This refers to the fact that the XML document is internally represented using 'C' data structures from the *XML* package. After importing the file, the workspace must be explicitly closed using `closeWorkspace()` in order to free up that memory.

4.3 Parsing the Workspace

With the workspace file open, we have not yet imported the XML document. The next step parses the XML workspace and creates R data structures to represent some of the information therein. Specifically, by calling `parseWorkspace()` the user will be presented with a list of *groups* in the workspace file and need to choose one group to import. Why only one? Because of the way flowJo handles data transformation and compensation. Each group of samples is associated with a compensation matrix and specific data transformation. These are applied to all samples in the group. When a particular group of samples is imported, the package generates a *GatingHierarchy* for each sample, describing the set of gates applied to the data (note: polygons, rectangles, quadrants, and ovals and boolean gates are supported). The set of GatingHierarchies for the group of samples is stored in a *GatingSet* object. Calling `parseWorkspace()` is quite verbose, informing the user as each gate is created. The parsing can also be done non–interactively by specifying which group to import directly in the function call (either an index or a group name). An additional optional argument `execute=T/F` specifies whether you want to load, compensate, transform the data and compute statistics immediately after parsing the XML tree.

```
> G<-parseWorkspace(ws,name=1,path=ws@path,isNcdf=FALSE,cleanup=FALSE,keep.indices=TR
> #Lots of output here suppressed for the vignette.
```

When isNcdf flag is set TRUE,the data is stored in ncdf format on disk.

```
> G

A GatingSet with 2 samples
```

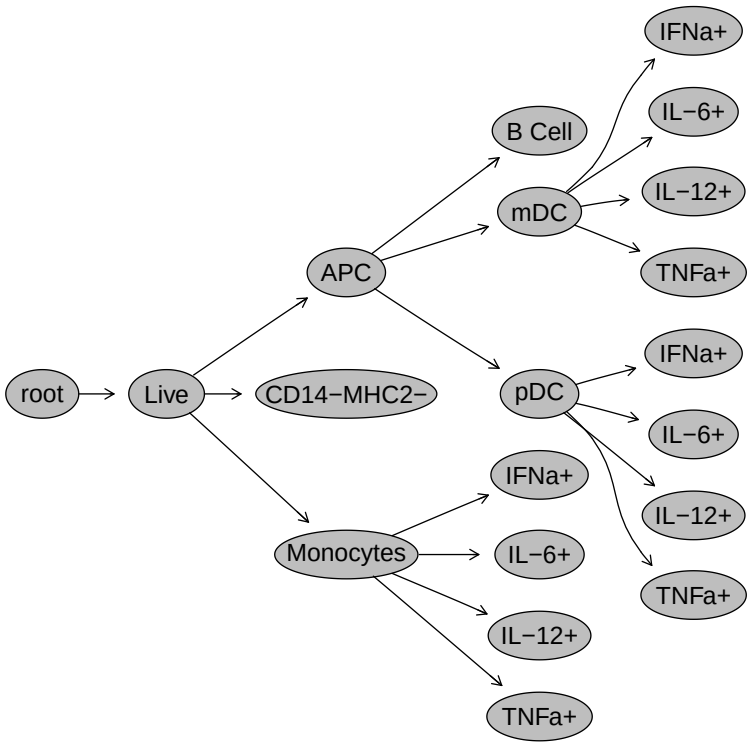
We have generated a *GatingSet* with 2 samples, each of which has 19 associated gates. Subsets of gating hierarchies can be accessed using the standard R subset syntax.

At this point we have parsed the workspace file and generate the gating hierarchy associated with each sample imported from the file. The data have

been loaded, compensated, and transformed in the workspace, and gating has been executed. The resulting *GatingSet* contains a replicated analysis of the original flowJo workspace.

We can plot the gating hierarchy for a given sample:

```
> gh<-G[[1]]
> plot(gh)
```



We can list the nodes (populations) in the gating hierarchy:

```
> nodelist<-getNodes(gh, path = 1)
> nodelist
```

```
 [1] "root"       "Live"       "APC"        "B Cell"     "mDC"
 [6] "IFNa+"      "IL-6+"      "IL-12+"     "TNFa+"      "pDC"
[11] "IFNa+"      "IL-6+"      "IL-12+"     "TNFa+"      "CD14-MHC2-"
[16] "Monocytes"  "IFNa+"      "IL-6+"      "IL-12+"     "TNFa+"
```

Note that the number preceding the period in the node names is just an identifier to help uniquely label populations in the gating hierarchy. It does

not represent any information about population statistics. When the node name itself is already unique,such as "Live","B Cell" ,the prefix is ommitted by default. We can force the prefix to be added to each node by setting `prefix` to "all".

```
> getNodes(gh, prefix="all", path = 1)
```

```
 [1] "root"       "Live"       "APC"        "B Cell"     "mDC"
 [6] "IFNa+"      "IL-6+"      "IL-12+"     "TNFa+"      "pDC"
[11] "IFNa+"      "IL-6+"      "IL-12+"     "TNFa+"      "CD14-MHC2-"
[16] "Monocytes"  "IFNa+"      "IL-6+"      "IL-12+"     "TNFa+"
```

Alternativly, the the full path of the node can be displayed by setting `path` to "full".

```
> getNodes(gh, path = "full")
```

```
 [1] "root"                    "/Live"                   "/Live/APC"
 [4] "/Live/APC/B Cell"        "/Live/APC/mDC"           "/Live/APC/mDC/IFNa+"
 [7] "/Live/APC/mDC/IL-6+"     "/Live/APC/mDC/IL-12+"    "/Live/APC/mDC/TNFa+"
[10] "/Live/APC/pDC"           "/Live/APC/pDC/IFNa+"     "/Live/APC/pDC/IL-6+"
[13] "/Live/APC/pDC/IL-12+"    "/Live/APC/pDC/TNFa+"     "/Live/CD14-MHC2-"
[16] "/Live/Monocytes"         "/Live/Monocytes/IFNa+"   "/Live/Monocytes/IL-6+"
[19] "/Live/Monocytes/IL-12+" "/Live/Monocytes/TNFa+"
```

We can get a specific gate definition:

```
> node<-nodelist[3]
> g<-getGate(gh,node)
> g
```

```
Polygonal gate 'APC' with 14 vertices in dimensions <PerCP-CY5-5-A> and <PE-CY7-A>
```

We can get the population proportion (relative to its parent) for a single population:

```
> getProp(gh,node)
```

```
[1] 0.08402716
```

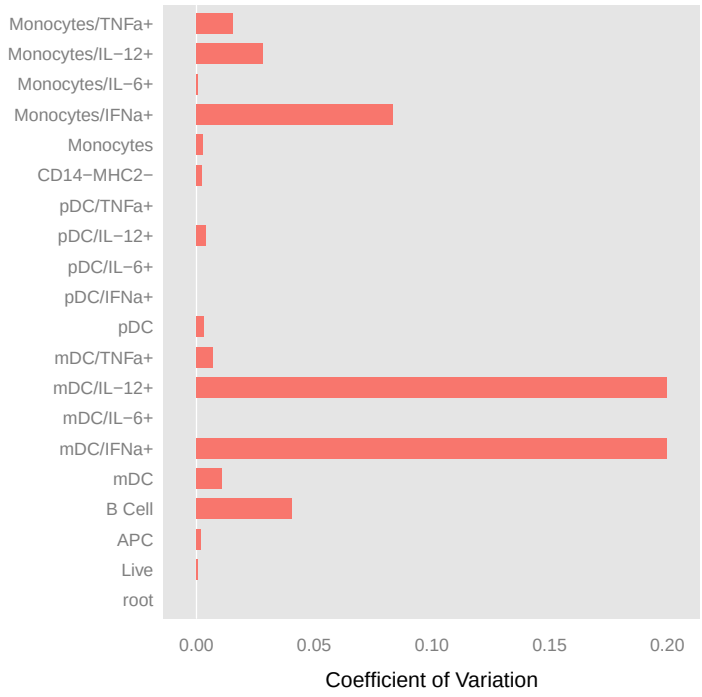
Or we can retrieve the population statistics for all populations in the sample:

```
> getPopStats(gh)
```

	flowCore.freq	flowJo.freq	flowJo.count	flowCore.count	node
1:	1.000000000	1.000000000	61832	61832	root
2:	0.800297581	0.801235606	49542	49484	Live
3:	0.084027160	0.083585645	4141	4158	APC
4:	0.592352092	0.548418256	2271	2463	B Cell
5:	0.123376623	0.121226757	502	513	mDC
6:	0.005847953	0.003984064	2	3	mDC/IFNa+
7:	0.042884990	0.043824701	22	22	mDC/IL-6+
8:	0.005847953	0.003984064	2	3	mDC/IL-12+
9:	0.140350877	0.141434263	71	72	mDC/TNFa+
10:	0.107984608	0.107703453	446	449	pDC
11:	0.002227171	0.002242152	1	1	pDC/IFNa+
12:	0.000000000	0.000000000	0	0	pDC/IL-6+
13:	0.561247216	0.560538117	250	252	pDC/IL-12+
14:	0.000000000	0.000000000	0	0	pDC/TNFa+
15:	0.544074852	0.540854225	26795	26923	CD14-MHC2-
16:	0.058928138	0.059161923	2931	2916	Monocytes
17:	0.003772291	0.004435346	13	11	Monocytes/IFNa+
18:	0.237654321	0.236779256	694	693	Monocytes/IL-6+
19:	0.047325103	0.049812351	146	138	Monocytes/IL-12+
20:	0.250685871	0.257250085	754	731	Monocytes/TNFa+

We can plot the coefficients of variation between the counts derived using flowJo and flowCore for each population:

```
> print(plotPopCV(gh))
```



We can plot individual gates: note the scale of the transformed axes. Second argument can be either a numeric index

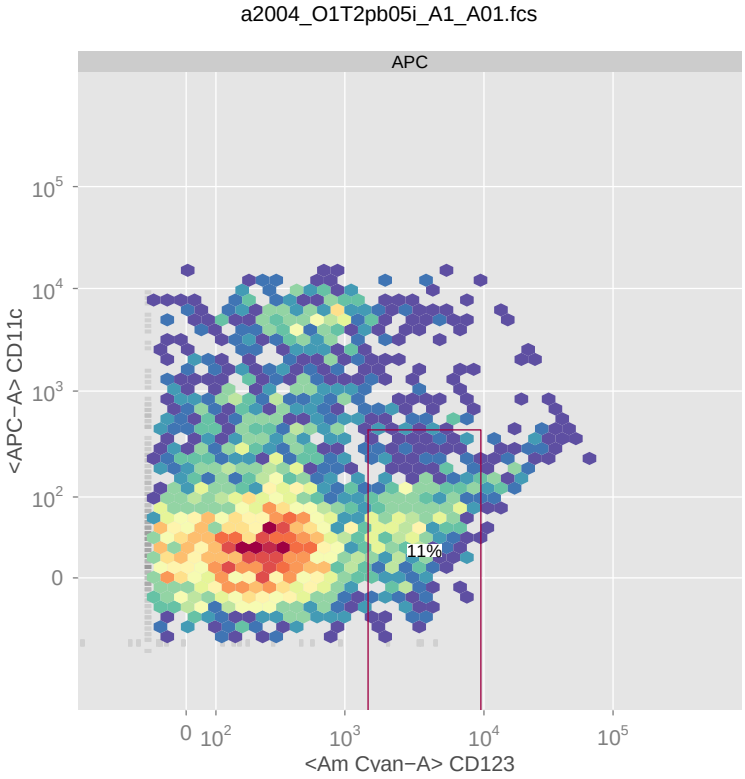
```
> plotGate(gh, 10)
```

or unique node name,

```
> plotGate(gh, "pDC")
```

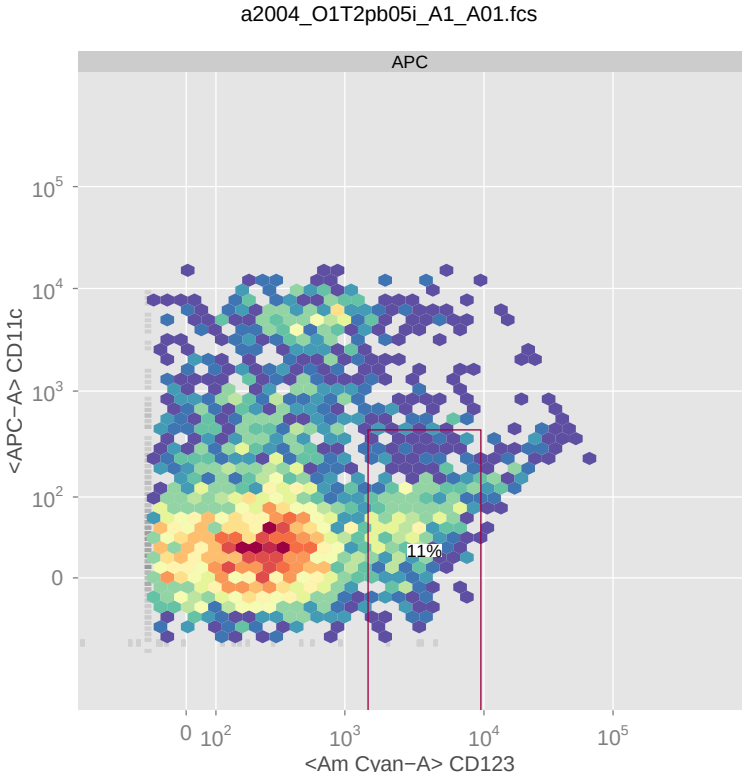
or a full/parital gating path,

```
> plotGate(gh, "APC/pDC")
```



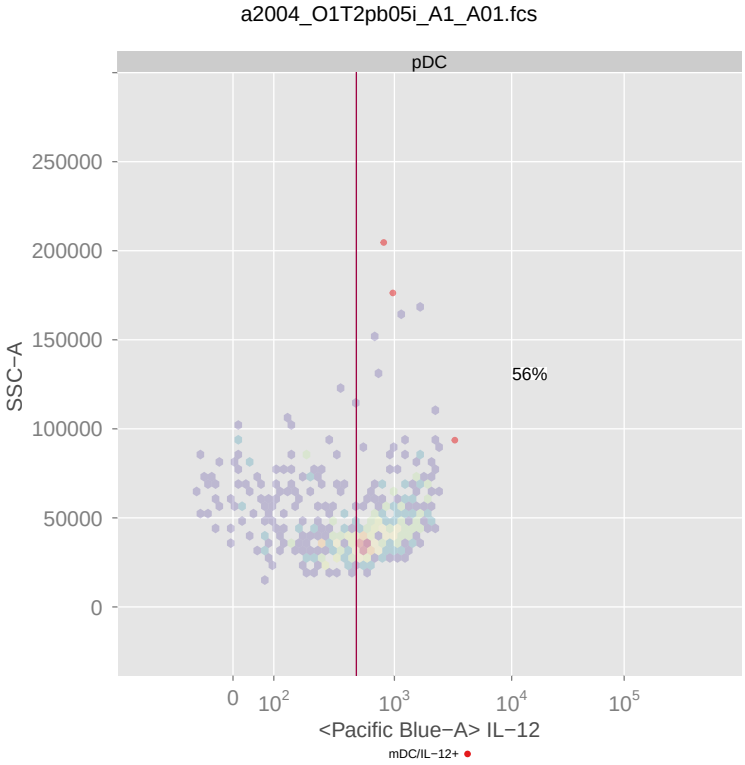
hexbin can be enabled to speed up plotting.

```
> plotGate(gh, "pDC", xbin =32)
```



`overlay` can be used to plot another cell population on top of the existing gates specified by `y`.

```
> plotGate(gh,y = "pDC/IL-12+", xbin =32, overlay = "mDC/IL-12+")
```



`overlay` can be either a `numeric` or `character` scalar indicating the index or gating path of a gate/population within the `GatingHierarchy` or a `logical` vector that indicates the cell event indices representing a sub-cell population.

If we have metadata associated with the experiment, it can be attached to the `GatingSet`.

```
> d<-data.frame(sample=factor(c("sample 1", "sample 2")),treatment=factor(c("sample",
> pd<-pData(G)
> pd<-cbind(pd,d)
> pData(G)<-pd
> pData(G);
```

```
                                                 name   sample treatment
a2004_O1T2pb05i_A1_A01.fcs a2004_O1T2pb05i_A1_A01.fcs sample 1    sample
a2004_O1T2pb05i_A2_A02.fcs a2004_O1T2pb05i_A2_A02.fcs sample 2   control
```

We can retrieve the subset of data associated with a node:

```
> getData(gh,node)
```

```
flowFrame object 'a2004_01T2pb05i_A1_A01.fcs'
with 4158 cells and 23 observables:
                 name  desc      range   minRange   maxRange
$P1             FSC-A  <NA> 262254.000 -111.00000 262143.000
$P2             FSC-H  <NA> 262143.000    0.00000 262143.000
$P3             FSC-W  <NA> 262143.000    0.00000 262143.000
$P4             SSC-A  <NA> 262254.000 -111.00000 262143.000
$P5             SSC-H  <NA> 262143.000    0.00000 262143.000
$P6             SSC-W  <NA> 262143.000    0.00000 262143.000
$P7       <Am Cyan-A> CD123   3661.959  435.34379   4097.303
$P8         Am Cyan-H CD123   3641.837  455.00000   4096.837
$P9  <Pacific Blue-A> IL-12   3927.974  169.60860   4097.582
$P10   Pacific Blue-H IL-12   3641.837  455.00000   4096.837
$P11          <APC-A> CD11c   4405.818 -308.01302   4097.805
$P12            APC-H CD11c   3641.837  455.00000   4096.837
$P13      <APC-CY7-A>  IL-6   3714.446  382.93207   4097.378
$P14        APC-CY7-H  IL-6   3641.837  455.00000   4096.837
$P15    <Alexa 700-A>  TNFa   3712.753  384.62271   4097.376
$P16      Alexa 700-H  TNFa   3641.837  455.00000   4096.837
$P17         <FITC-A>  IFNa   4180.519  -82.81306   4097.706
$P18           FITC-H  IFNa   3641.837  455.00000   4096.837
$P19  <PerCP-CY5-5-A> MHCII   4942.398 -844.59317   4097.805
$P20    PerCP-CY5-5-H MHCII   3641.837  455.00000   4096.837
$P21       <PE-CY7-A>  CD14   4942.398 -844.59317   4097.805
$P22         PE-CY7-H  CD14   3641.837  455.00000   4096.837
$P23             Time  <NA>   9918.400   89.00000  10007.400
323 keywords are stored in the 'description' slot
```

When applied to the `GatingSet`,a `flowSet` or `ncdfFlowSet` is returned.

```
> getData(G,node);

A flowSet with 2 experiments.

An object of class 'AnnotatedDataFrame'
  rowNames: a2004_01T2pb05i_A1_A01.fcs a2004_01T2pb05i_A2_A02.fcs
  varLabels: name sample treatment
  varMetadata: labelDescription

  column names:
  FSC-A FSC-H FSC-W SSC-A SSC-H SSC-W <Am Cyan-A> Am Cyan-H <Pacific Blue-A> Pacific
```

Or we can retrieve the indices specifying if an event is included inside or outside a gate using:

```
> table(getIndices(gh,node))

FALSE  TRUE
57674  4158
```

The indices returned are relative to the parent population (member of parent AND member of current gate), so they reflect the true hierarchical gating structure.

If we wish to do compensation or transformation manually, we can retrieve all the compensation matrices from the workspace:

```
> C<-getCompensationMatrices(gh);
> C

Compensation object 'defaultCompensation':
               Am Cyan-A Pacific Blue-A    APC-A APC-CY7-A Alexa 700-A
Am Cyan-A        1.00000        0.04800 0.000000    0.0000     0.00000
Pacific Blue-A   0.38600        1.00000 0.000529    0.0000     0.00000
APC-A            0.00642        0.00235 1.000000    0.0611     0.19800
APC-CY7-A        0.03270        0.02460 0.084000    1.0000     0.02870
Alexa 700-A      0.07030        0.05800 0.016200    0.3990     1.00000
FITC-A           0.74500        0.02090 0.001870    0.0000     0.00000
PerCP-CY5-5-A    0.00368        0.00178 0.015300    0.0269     0.07690
PE-CY7-A         0.01330        0.00948 0.000951    0.1380     0.00182
                  FITC-A PerCP-CY5-5-A PE-CY7-A
Am Cyan-A       0.028500       0.00104  0.00000
Pacific Blue-A  0.000546       0.00000  0.00000
APC-A          -0.000611       0.00776  0.00076
APC-CY7-A       0.002690       0.00304  0.01010
Alexa 700-A     0.001530       0.10800  0.00679
FITC-A          1.000000       0.04180  0.00281
PerCP-CY5-5-A   0.000000       1.00000  0.07030
PE-CY7-A        0.002340       0.03360  1.00000
```

Or we can retrieve transformations:

```
> T<-getTransformations(gh)
> names(T)
```

```
 [1] "A2004-A2005_06i <Alexa 700-A>"    " <Alexa 700-H>"
 [3] "A2004-A2005_06i <Am Cyan-A>"      " <Am Cyan-H>"
 [5] "A2004-A2005_06i <APC-A>"          "A2004-A2005_06i <APC-CY7-A>"
 [7] " <APC-CY7-H>"                     " <APC-H>"
 [9] "A2004-A2005_06i <FITC-A>"         " <FITC-H>"
[11] "A2004-A2005_06i <Pacific Blue-A>" " <Pacific Blue-H>"
[13] "A2004-A2005_06i <PE-CY7-A>"       " <PE-CY7-H>"
[15] "A2004-A2005_06i <PerCP-CY5-5-A>"  " <PerCP-CY5-5-H>"

> T[[1]]

function (x, deriv = 0)
{
    deriv <- as.integer(deriv)
    if (deriv < 0 || deriv > 3)
        stop("'deriv' must be between 0 and 3")
    if (deriv > 0) {
        z0 <- double(z$n)
        z[c("y", "b", "c")] <- switch(deriv, list(y = z$b, b = 2 *
            z$c, c = 3 * z$d), list(y = 2 * z$c, b = 6 * z$d,
            c = z0), list(y = 6 * z$d, b = z0, c = z0))
        z[["d"]] <- z0
    }
    res <- stats:::.splinefun(x, z)
    if (deriv > 0 && z$method == 2 && any(ind <- x <= z$x[1L]))
        res[ind] <- ifelse(deriv == 1, z$y[1L], 0)
    res
}
<environment: 0x061b0ec4>
attr(,"type")
[1] "caltbl"
```

`getTransformations` returns a list of functions to be applied to different dimensions of the data. Above, the transformation is applied to this sample, the appropriate dimension is transformed using a channel–specific function from the list.

The list of samples in a workspace can be accessed by:

```
> getSamples(ws);
```

```
  sampleID                       name count compID pop.counts
1        1 a2004_01T2pb05i_A1_A01.fcs 61832      1         19
2        2 a2004_01T2pb05i_A2_A02.fcs 45363      1         19
```

And the groups can be accessed by:

```
> getSampleGroups(ws)

    groupName groupID sampleID
1 All Samples       0        1
2 All Samples       0        2
```

The `compID` column tells you which compensation matrix to apply to a group of files, and similarly, based on the name of the compensation matrix, which transformations to apply.

4.4 add/remove gates

`GatingSet` is equivalent to the `workFlow` in flowCore package ,which provides methods to build a gating tree from raw FCS files and add or remove flowCore gates(or populations) to or from it. Firstly,we start from a flowSet that contains three ungated flow samples:

```
> data(GvHD)
> #select raw flow data
> fs<-GvHD[1:3]
```

Then we can transform the raw data:

```
> tf <- transformList(colnames(fs[[1]])[3:6], asinh, transformationId="asinh")
> fs_trans<-transform(fs,tf)
```

and create a gatingset from the transformed flowSet:

```
> gs <- GatingSet(fs_trans)
> gs

A GatingSet with 3 samples

> gh1<-gs[[1]]
> getNodes(gh1)

[1] "root"
```

It now only contains the root node. We can add one rectangleGate:

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

[1] 2

> getNodes(gh1)

[1] "root"       "/rectangle"
```

Note that the gate is added to root node by default if parent is not specified. Then we add a quadGate to the new population generated by the rectangeGate which is named after filterId of the gate because the name is not specified when `add` method is called.

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

[1] 3 4 5 6

> getNodes(gh1)

[1] "root"                           "/rectangle"
[3] "/rectangle/CD15 FITC-CD45 PE+" "/rectangle/CD15 FITC+CD45 PE+"
[5] "/rectangle/CD15 FITC+CD45 PE-" "/rectangle/CD15 FITC-CD45 PE-"
```

Here quadGate produces four population nodes/populations whose names are named after dimensions of gate if not specified.

Boolean Gate can also be defined and added to GatingSet:

```
> bg<-booleanFilter(`CD15 FITC-CD45 PE+|CD15 FITC+CD45 PE-`)
> bg

booleanFilter filter 'CD15 FITC-CD45 PE+|CD15 FITC+CD45 PE-' evaluating the expressio
CD15 FITC-CD45 PE+|CD15 FITC+CD45 PE-

> nodeID2<-add(gs,bg,parent="rectangle")
> nodeID2
```

```
[1] 7
```

```
> getNodes(gh1)
```

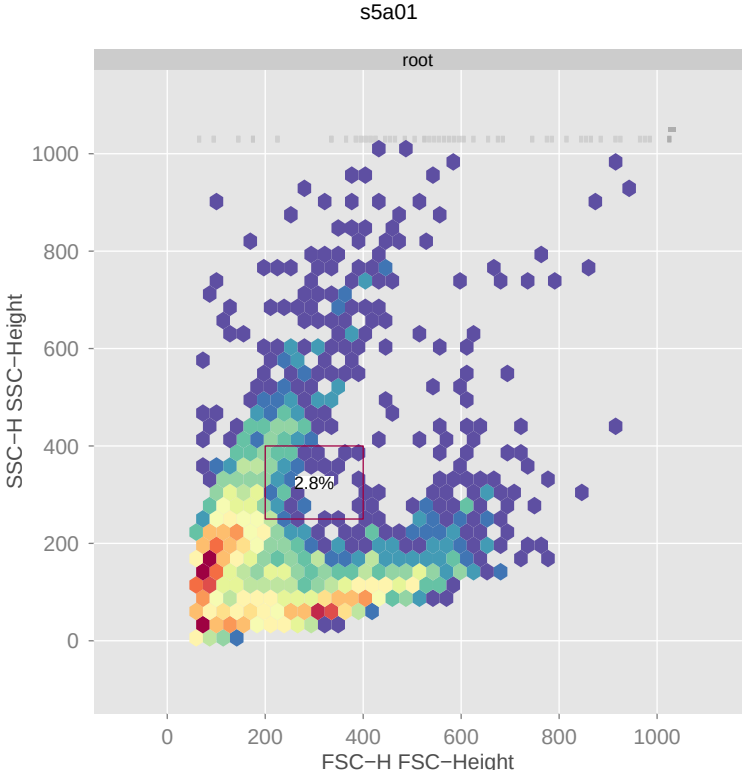
```
[1] "root"
[2] "/rectangle"
[3] "/rectangle/CD15 FITC-CD45 PE+"
[4] "/rectangle/CD15 FITC+CD45 PE+"
[5] "/rectangle/CD15 FITC+CD45 PE-"
[6] "/rectangle/CD15 FITC-CD45 PE-"
[7] "/rectangle/CD15 FITC-CD45 PE+|CD15 FITC+CD45 PE-"
```

Now the gating tree is finished but the actual gating is done by `recompute` method:

```
> recompute(gs)
```

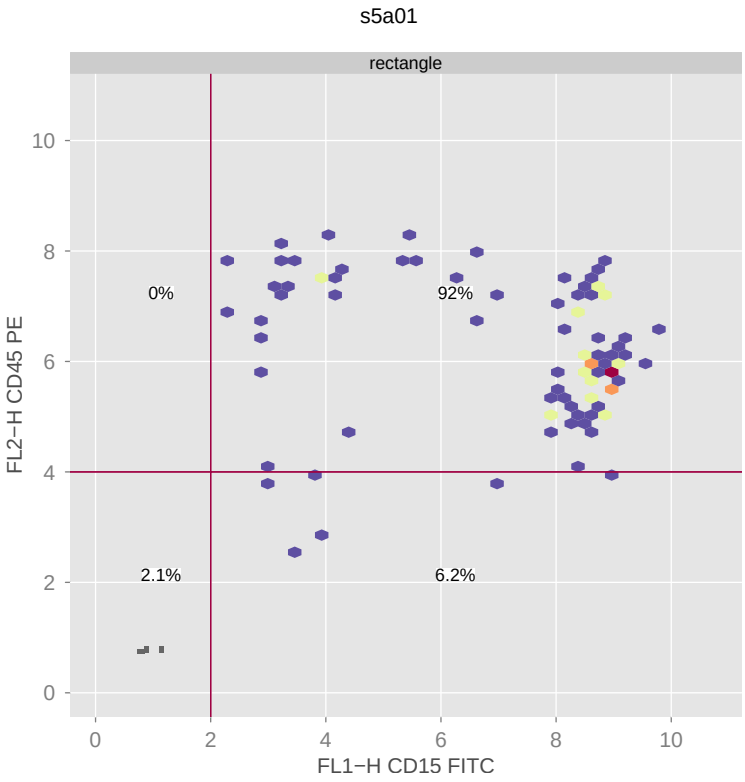
After gating is finished,gating results can be visualized by `plotGate` method:

```
> plotGate(gh1,"rectangle") #plot one Gate
```



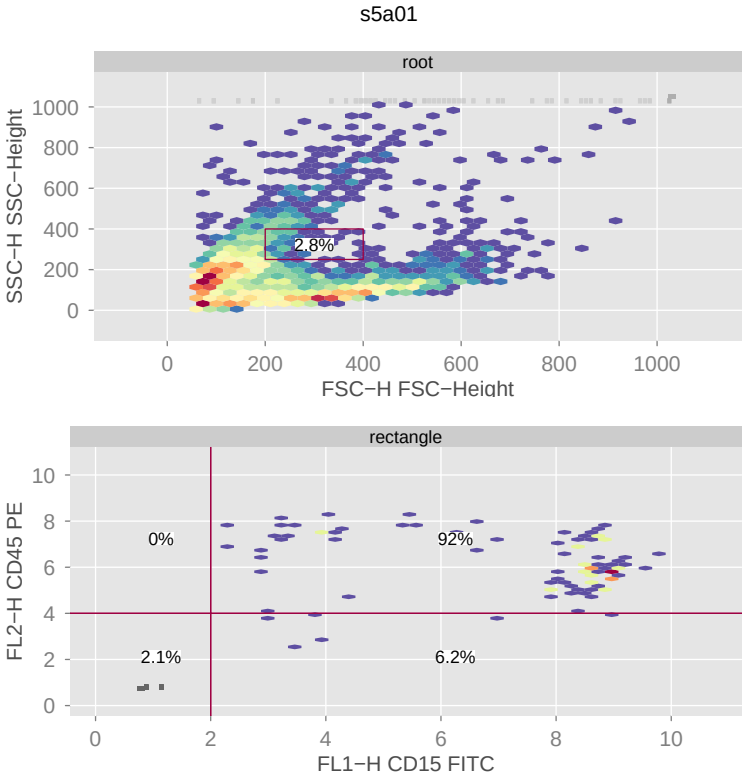
The second argument of `plotGate` is used to specify node index/name. Multiple gates can be plotted on the same pannel:

```
> plotGate(gh1,nodeIDs)
```



We may also want to plot all the gates without specifying the gate index:

```
> plotGate(gh1)
```

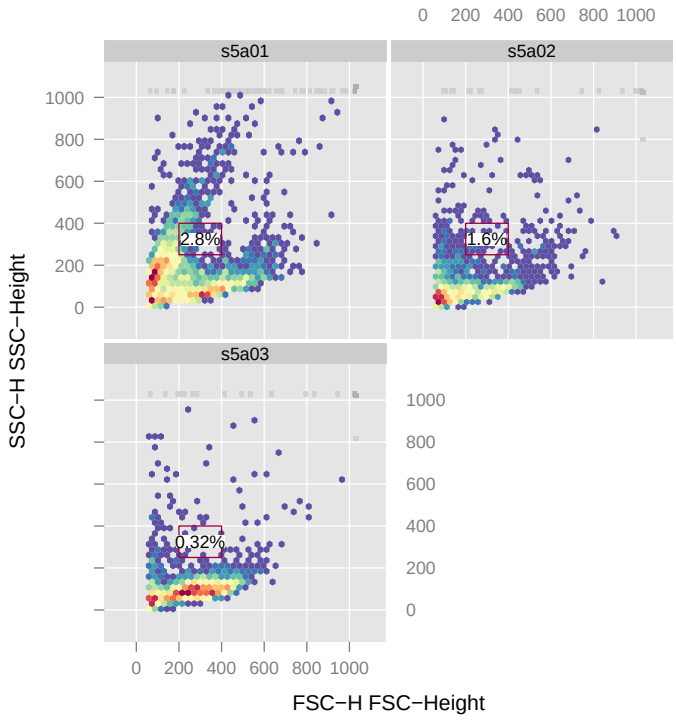


Boolean gate is skipped by default and can be enabled by:

```
> plotGate(gh1,bool=TRUE)
```

Note that smoothing may be applied automatically if there are not enough events after gating Sometime it is more useful to compare gates across samples using lattice plot by applying `plotGate` to a `GatingSet`:

```
> plotGate(gs,getNodes(gs)[nodeID])
```



The gating hierarchy is plotted by:

```
> plot(gh1,bool=TRUE)
```

If we want to remove one node, simply:

```
> Rm('rectangle', gs)
> getNodes(gh1)
```

```
[1] "root"
```

As we see,removing one node causes all its descendants to be removed as well.

4.5 archive and clone

Oftentime, we need to save a GatingSet including the gated flow data,gates and populations to disk and reload it later on. It can be achieved by:

```
>          save_gs(gs,file = "~/gs")
>          G1 <- load_gs(file = "~/gs")
```

We also provide the `clone` method to make a full copy of an existing `GatingSet`:

```
>         gs_cloned <- clone(gs)
```

Note that the `GatingSet` contains environment slots and external pointer that point to the internal C data structure. So make sure to use these methods in order to save or make a copy of existing object. The regular R assignment (<-) or `save` routine doesn't work as expected for the `GatingSet` object.

4.6 Exporting to FlowJo OSX 9.2

The `exportAsFlowJoXML` function can be used to export a `flowCore::workFlow` as an XML workspace for FlowJo 9.2 OSX. If flowWorkspace has been used to import an existing FlowJo workspace, `flowWorkspace2flowCore` can be used to obtain a `workFlow` for exporting. Currently this function can export one workFlow at a time.

4.7 Deprecated Functionality

The following behaviour is no longer supported and has been replace by more extensive netCDF support via the ncdfFlow package. If you have particularly large data files (millions of events), then you won't want to keep the data around, nor the indices for gate membership. Instead, pass the options `cleanup=TRUE, keep.indices=FALSE` to the `execute()` function, and the data will be scrubbed after computing population statistics. With future improvements making use of the netCDF framework, and bitvector representations of population memberships; this will improve memory usage in high–throughput unsupervised analysis settings.

5 Troubleshooting

If this package is throwing errors when parsing your workspace, and you are certain your workspace is version 2.0, contact the package author. If you can send your workspace by email, we can test, debug, and fix the package so that it works for you. Our goal is to provide a tool that works, and that people find useful.

6 Future Improvements

We are working on support for flowJo XML workspaces exported from the Windows version of flowJo. Efforts are underway to integrate GatingSet and GatingHierarchy objects more closely with the rest of the flow infrastructure.