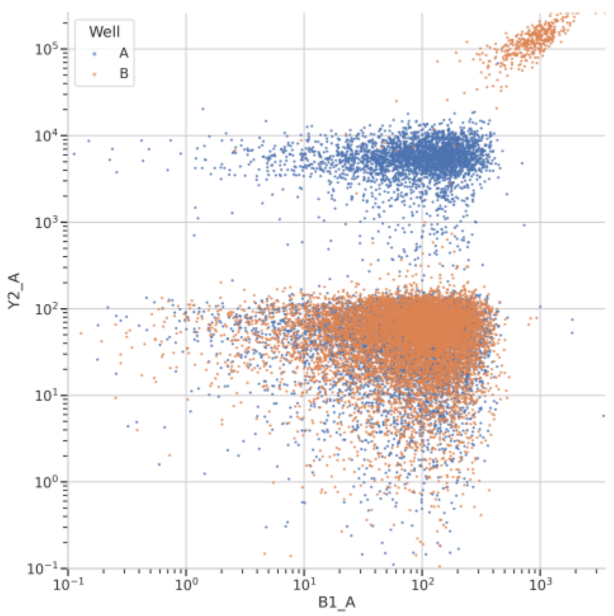
Cytoflow Documentation

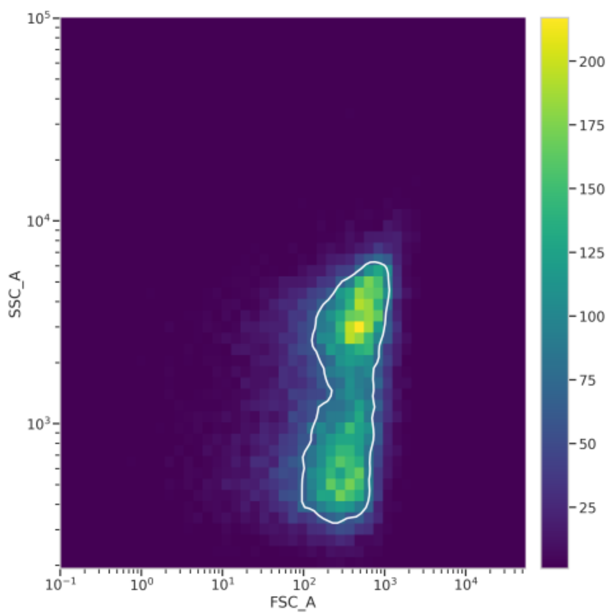
Release 1.2+9.gc8e40ee.dirty

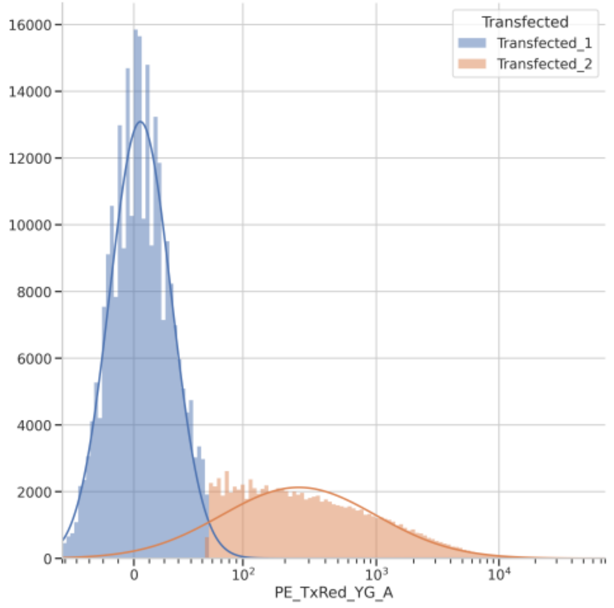
Brian Teague

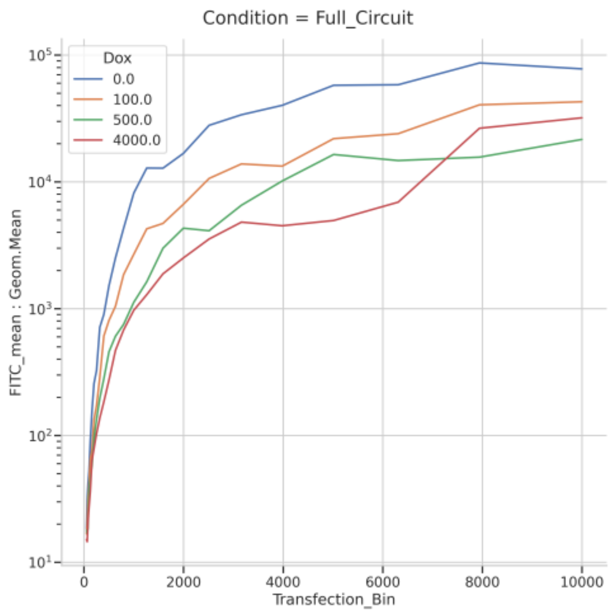
Apr 07, 2022

CONTENTS









Cytoflow is a **point-and click program** and a **Python library** for analyzing flow cytometry data. It was written by Brian Teague to address shortcomings in currently-available flow software.

CHAPTER

ONE

HOW IS CYTOFLOW DIFFERENT?

- An emphasis on **metadata.** Cytoflow assumes you are measuring the fluorescence of several samples that were treated differently: collected at different times, treated with varying levels of a chemical, etc. You specify these sample conditions up front, then use those conditions to control your analysis.
- The analysis is represented as a **workflow**. Operations such as gating and compensation are applied sequentially; a workflow can be saved a re-used or shared with colleagues.
- **Good visualization.** Make publication-ready plots right from the software.
- **Thoughtful documentation.** Each module, operation and visualization is documented, and particular attention has been paid to getting users up-and-running quickly.
- Cytoflow is built on **Python modules** that you can use in your own workflows. The library was designed to work particularly well in a Jupyter notebook.
- Cytoflow is **free and open-source.** Contribute bug reports at the GitHub project page or download the source code to modify it for your own needs. Then, contribute your changes back so the rest of the community can benefit from them.

If you'd like to see Cytoflow in action, here are some *screenshots*, or you can check out a screencast on Youtube.

CHAPTER

TWO

GETTING STARTED

Quick installation instructions for the point-and-click program:

- **Windows**: Download the installer, then run it.
- **MacOS (10.10+)**: Download the ZIP file. Unzip the file, then double-click to run the program. Depending on your security settings, you may have to specifically enable this program (it's not signed.)
- **Linux**: Download the tarball. Extract it, then run the *cytoflow* binary.

More detailed installation instructions *can be found in the user's manual.*.

CHAPTER

THREE

DOCUMENTATION

The *user manual* is broken into four different sets of documentation:

- *Tutorials* take you step-by-step through some basic analyses. Start here if you're new to Cytoflow.
- *How-to guides* are recipes. They guide you through some common use-cases. They are more advanced than tutorials and assume some knowledge of how Cytoflow works.
- *User guides* discuss how Cytoflow works "under the hood", which can make you a better Cytoflow user for more advanced or non-standard analyses.
- *Reference pages* document every operation and view. These are the same pages that are displayed in Cytoflow's "Help" panel.

CHAPTER

FOUR

FOR DEVELOPERS

If you want to use Python to analyze flow cytometry data, then Cytoflow is for you! I've found Cytoflow useful for both *interactive data exploration* (ie, poking data to see what it's telling you) and *automated data analysis* (ie, writing scripts and pipelines to process lots of data.)

To get a taste of what Cytoflow can do, check out an example Jupyter notebook.

Then, head over to the *developers' manual*. There, you'll find:

- *Tutorial Jupyter notebooks* to get you started writing your own analyses.
- *In-depth examples* that demonstrate more advanced usage.
- *How-to guides* to help you with common tasks.
- *Developer's guides* to help you understand how Cytoflow is structured (particularly useful for writing your own modules.)
- An *API reference*, documenting all the user-facing classes, operations and views.

CHAPTER FIVE

FOR CONTRIBUTORS

Hooray! We'd love to have you.

The Cytoflow source code is hosted over at GitHub. Feel free to report bugs, request enhancements, fork the codebase and start hacking. Also, check out the *Developers' guides* for an overview and justification of Cytoflow's design, as well as useful information for writing your own modules, operations and views.

5.1 User manual

Note: If there's something you'd like to do that isn't well-explained by the user manual, please file an issue on GitHub.

The **User Manual** is divided into four sections:

1. *Tutorials* are step-by-step examples to get you started using Cytoflow. They include both some basic tutorials as well as examples of some more advanced analyses.

5.1.1 Tutorial: Quickstart

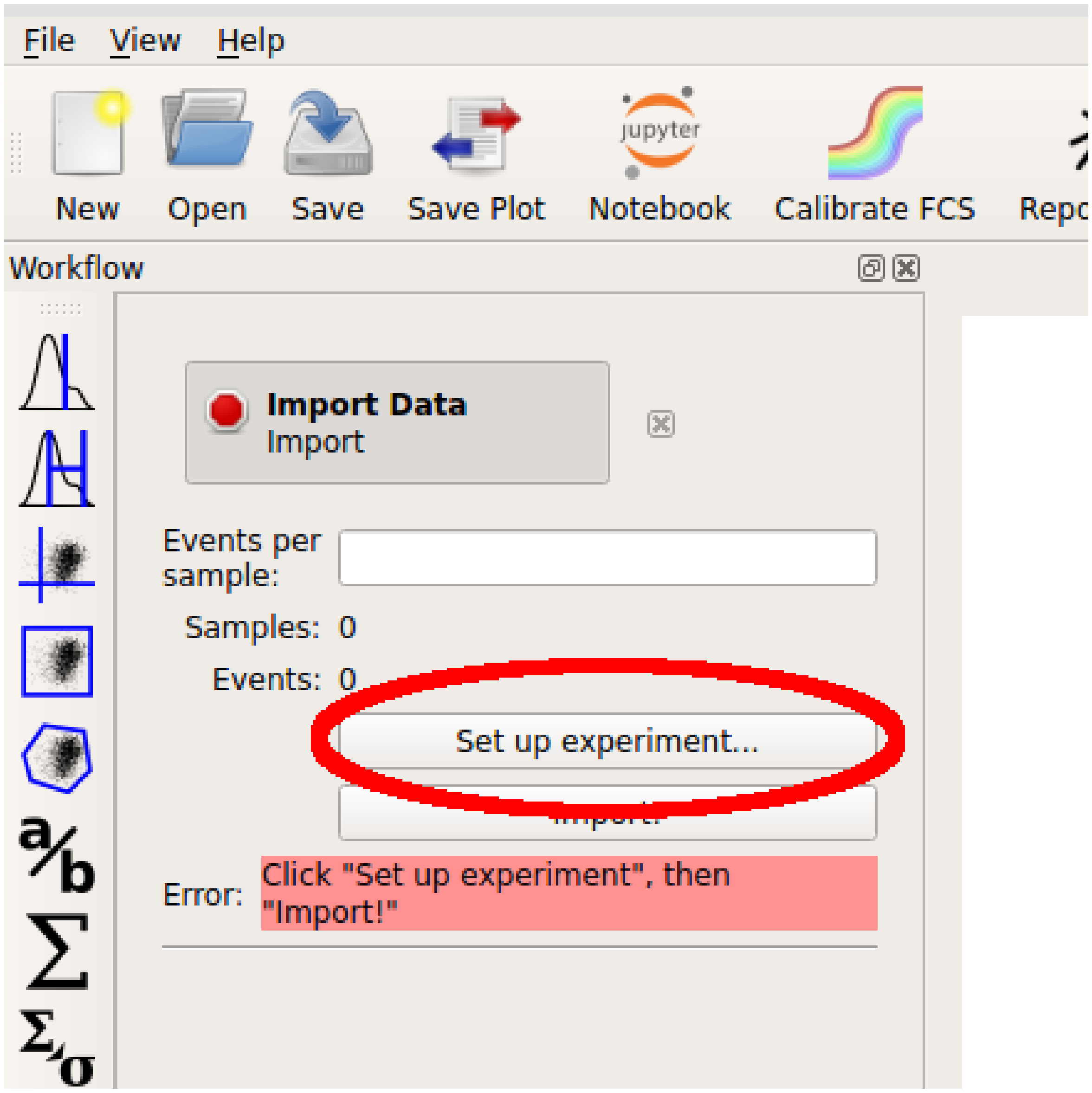
Welcome to Cytoflow! This document will demonstrate importing some data and running a very basic analysis using the GUI. It should help you get started with the following steps:

- Importing data
- Basic visualizations
- Gating
- Summary statistics

If you'd like to follow along, you can do so by downloading one of the **cytoflow-#####-examples-basic.zip** files from the Cytoflow releases page on GitHub.

Importing data

Start the software. The left panel is the "workflow" panel, and upon startup has a single operation named **Import Data**. Click *Set up experiment…*

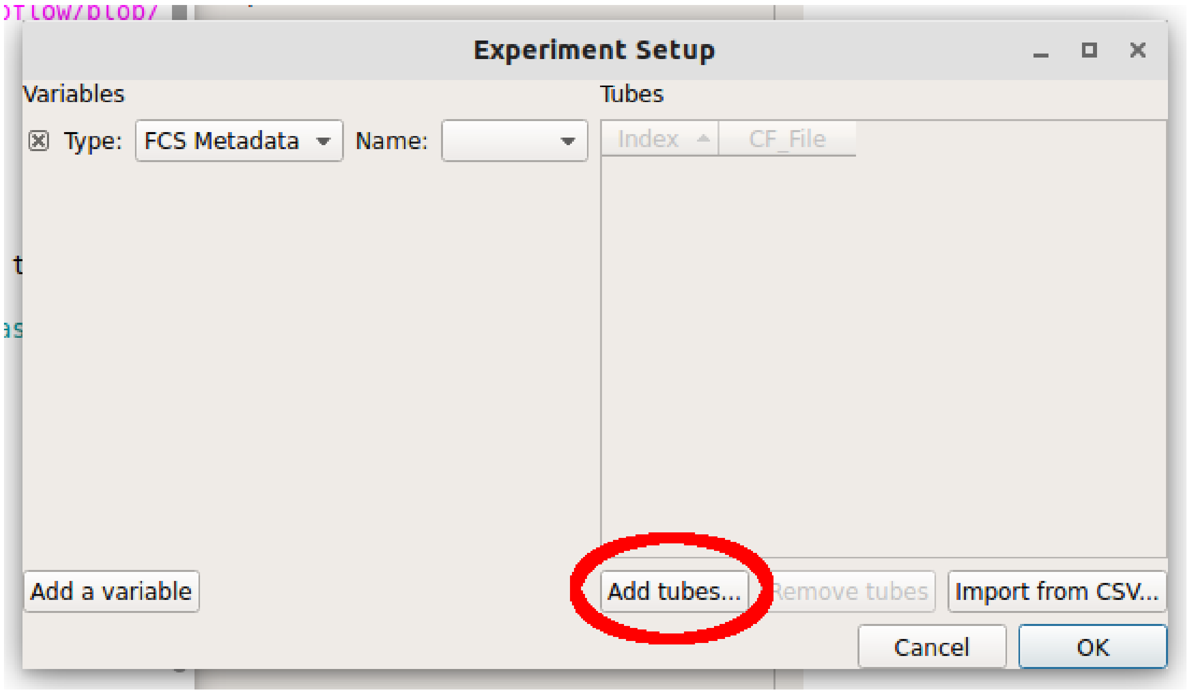


Samples: 0

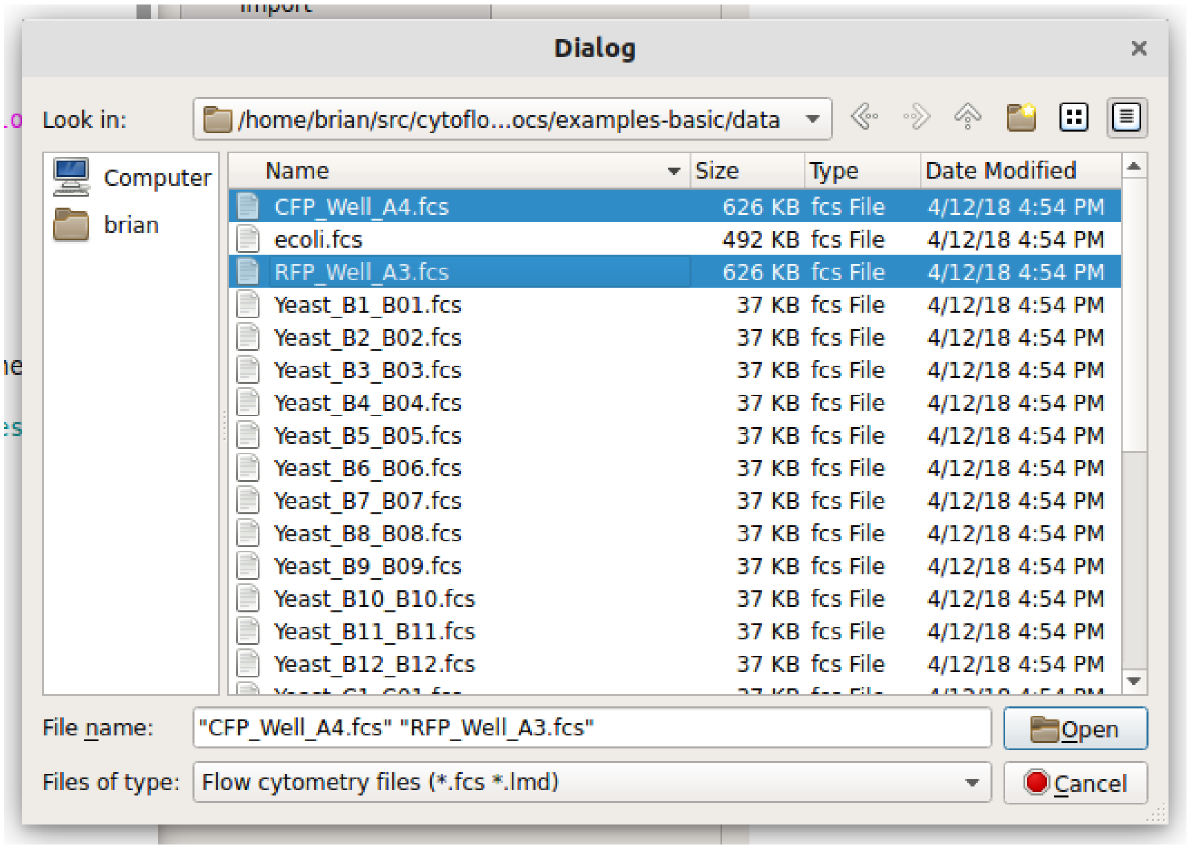
Events: 0

Error: Click "Set up experiment", then "Import!"

In the **Experiment Setup** dialog that appears, click *Add tubes…*



Choose *CFP_Well_A4.fcs* and *RFP_Well_A3.fcs*. You can select multiple files by holding down the *Control* key (Windows or Linux) or the *Command* key (Mac) while clicking on the files.



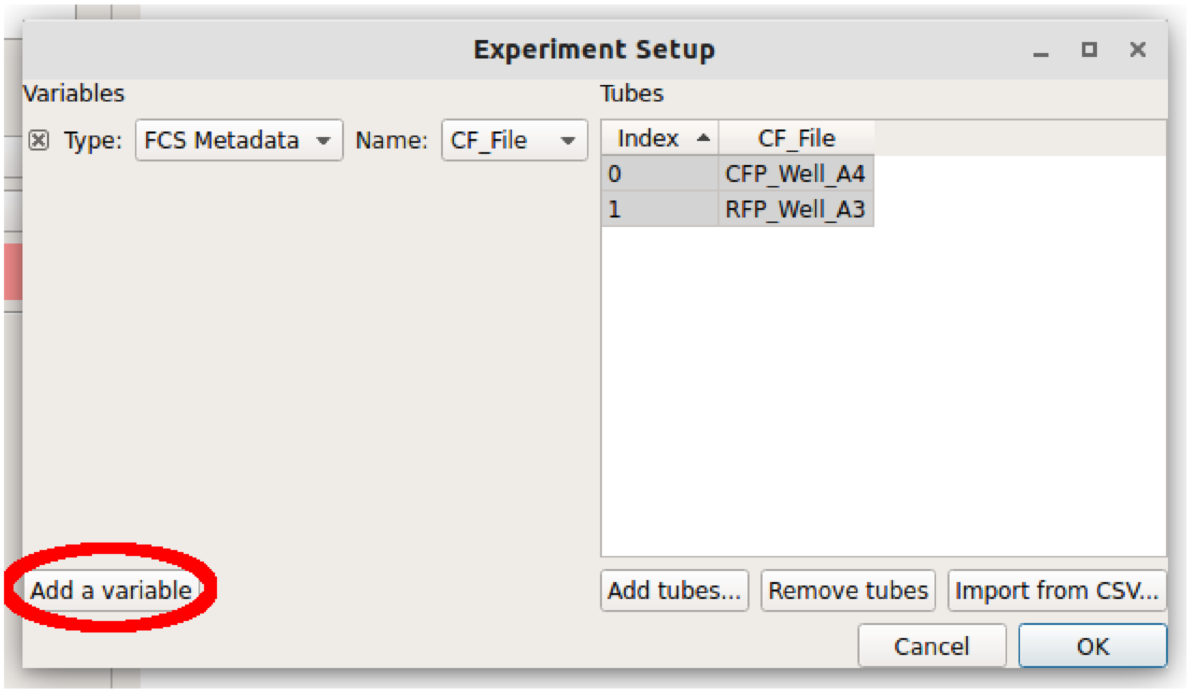
Look in: /home/brian/src/cytoflo...ocs/examples-basic/data

Name	Size	Type	Date Modified
CFP_Well_A4.fcs	626 KB	fcs File	4/12/18 4:54 PM
ecoli.fcs	492 KB	fcs File	4/12/18 4:54 PM
RFP_Well_A3.fcs	626 KB	fcs File	4/12/18 4:54 PM
Yeast_B1_B01.fcs	37 KB	fcs File	4/12/18 4:54 PM
Yeast_B2_B02.fcs	37 KB	fcs File	4/12/18 4:54 PM
Yeast_B3_B03.fcs	37 KB	fcs File	4/12/18 4:54 PM
Yeast_B4_B04.fcs	37 KB	fcs File	4/12/18 4:54 PM
Yeast_B5_B05.fcs	37 KB	fcs File	4/12/18 4:54 PM
Yeast_B6_B06.fcs	37 KB	fcs File	4/12/18 4:54 PM
Yeast_B7_B07.fcs	37 KB	fcs File	4/12/18 4:54 PM
Yeast_B8_B08.fcs	37 KB	fcs File	4/12/18 4:54 PM
Yeast_B9_B09.fcs	37 KB	fcs File	4/12/18 4:54 PM
Yeast_B10_B10.fcs	37 KB	fcs File	4/12/18 4:54 PM
Yeast_B11_B11.fcs	37 KB	fcs File	4/12/18 4:54 PM
Yeast_B12_B12.fcs	37 KB	fcs File	4/12/18 4:54 PM

File name: "CFP_Well_A4.fcs" "RFP_Well_A3.fcs"

Files of type: Flow cytometry files (*.fcs *.lmd)

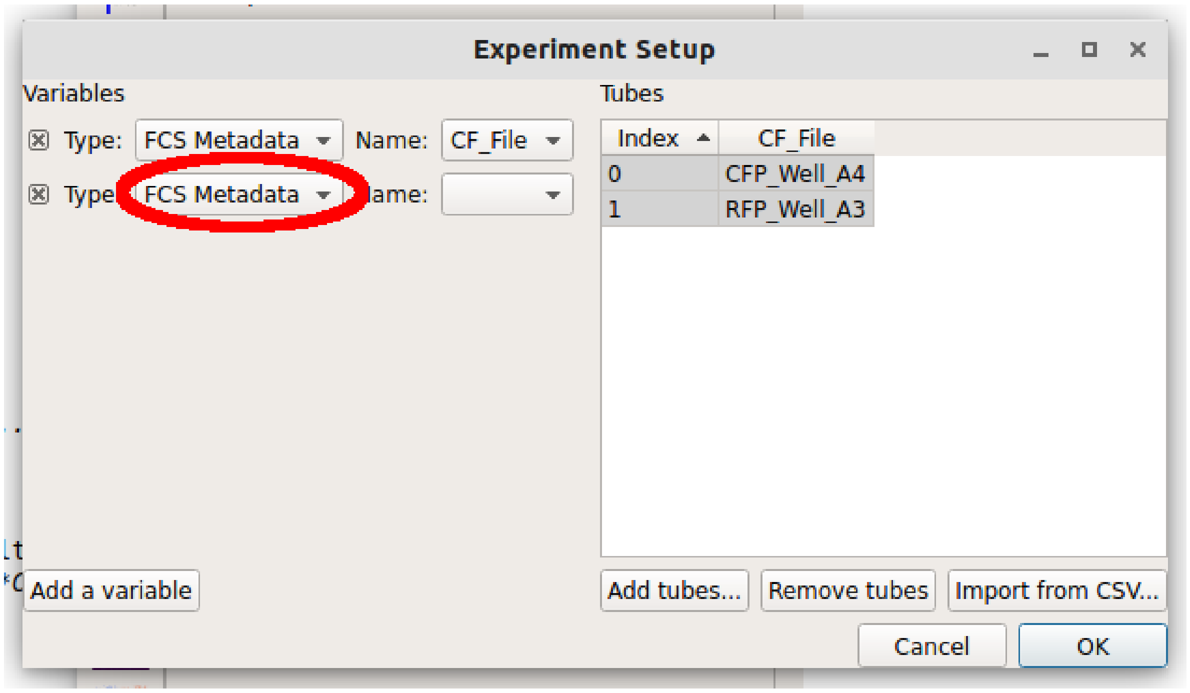
Each file is now a row in the *Tubes* table. Cytoflow assumes that your FCS files were all collected under the same instrument settings (voltages, etc) but have varying experimental conditions (ie your independent variables.) We now have to tell Cytoflow what those conditions are for each tube. Click *Add a variable*.



Type: FCS Metadata Name: CF_File

Index	CF_File
0	CFP_Well_A4
1	RFP_Well_A3

We can make a new variable that is a *Category*, a *Number* or *True/False* (ie a boolean.) Since these two tubes were exposed to different amounts of the small molecule *doxycycline*, pull down the "Type" selector and choose *Number*.



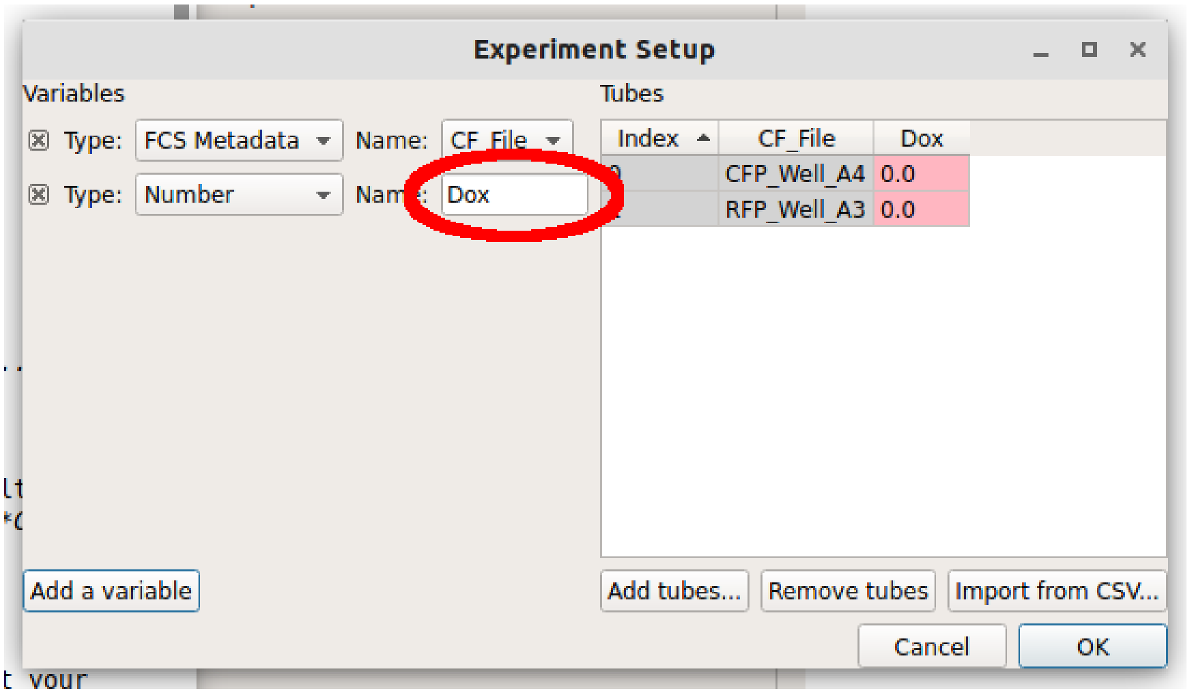
Type: FCS Metadata Name: CF_File

Type: FCS Metadata Name:

Index	CF_File
0	CFP_Well_A4
1	RFP_Well_A3

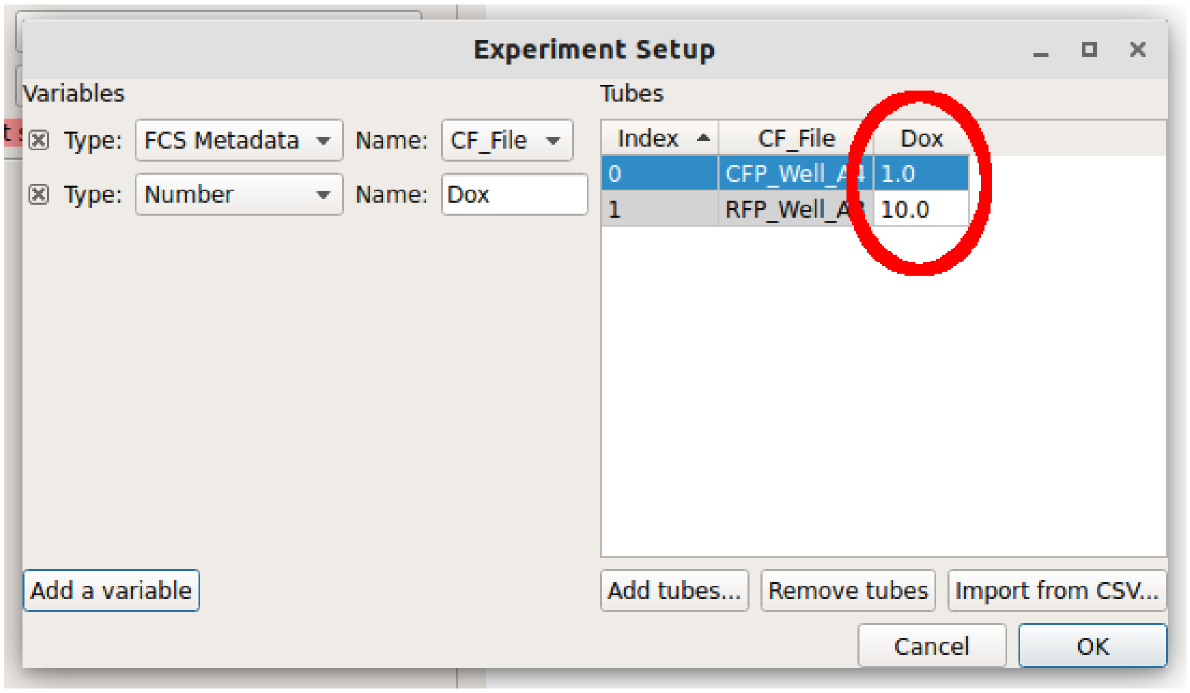
Note: If you have an experimental variable saved as metadata in your FCS file, see the *Frequently Asked Questions*.

Type the name "Dox" into the *Name* box. Notice how there's now a new column named "Dox" in the *Tubes* table.



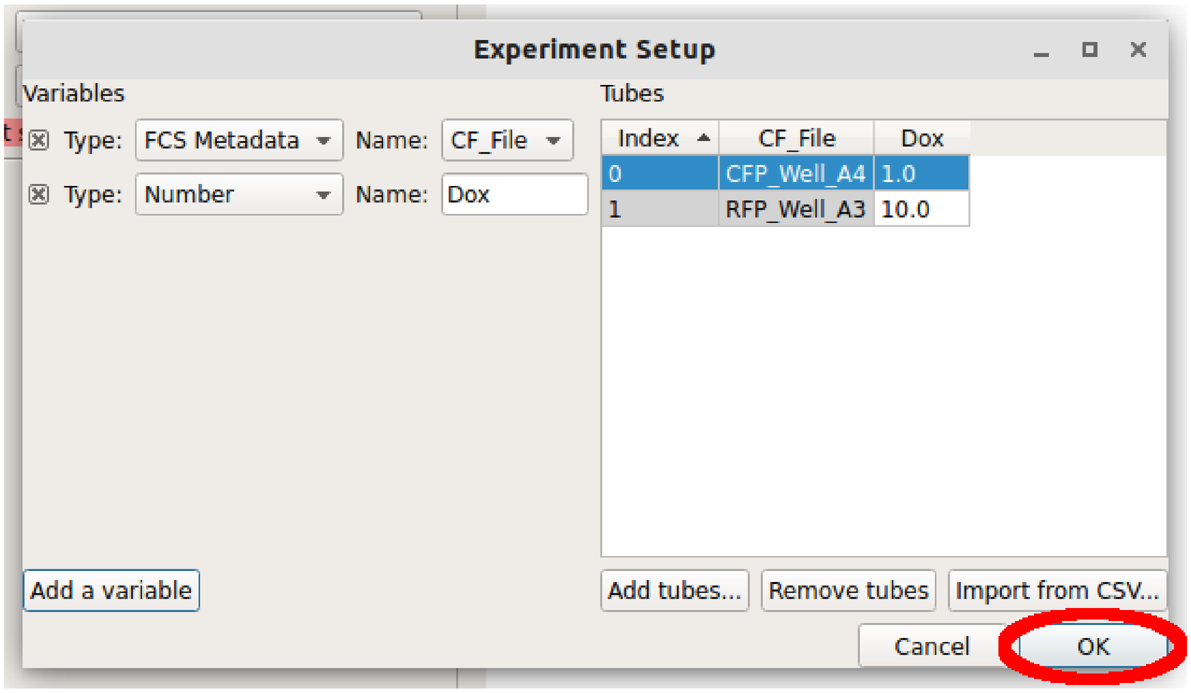
Index	CF_File	Dox
0	CFP_Well_A4	0.0
1	RFP_Well_A3	0.0

Note how the new column is red. Each tube must have a *unique set of conditions* to import the data – any tube that shares conditions with another tube is labeled red like this. Type "1.0" into the "Dox" column for the first row (CFP_Well_A4) and "10.0" into the "Dox" column for the second row (RFP_Well_A3). Notice how all the cells are white.



Index	CF_File	Dox
0	CFP_Well_A4	1.0
1	RFP_Well_A3	10.0

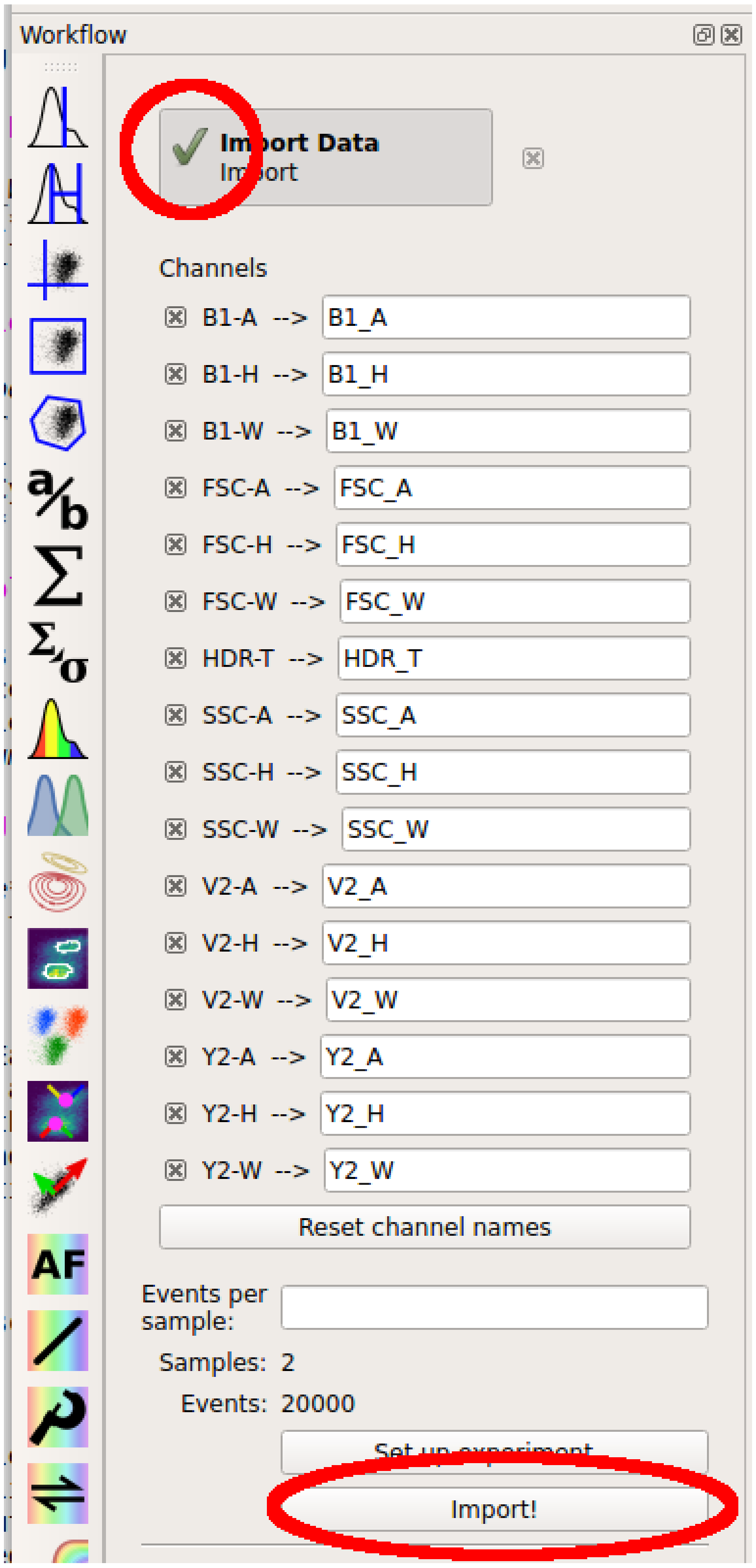
Click "OK" to return to the main screen.



Index	CF_File	Dox
0	CFP_Well_A4	1.0
1	RFP_Well_A3	10.0

Now, all the channels in those files show up in the Import dialog. You can rename the channels if you'd like, or remove channels that you're not using, but we won't worry about that here. Click "Import!" to import the data. Note that the red stop-sign on the module header changes to a green check-mark to show that the operation succeeded.

Basic plotting


Workflow

Import Data
Import

Channels

B1-A -->	B1_A
B1-H -->	B1_H
B1-W -->	B1_W
FSC-A -->	FSC_A
FSC-H -->	FSC_H
FSC-W -->	FSC_W
HDR-T -->	HDR_T
SSC-A -->	SSC_A
SSC-H -->	SSC_H
SSC-W -->	SSC_W
V2-A -->	V2_A
V2-H -->	V2_H
V2-W -->	V2_W
Y2-A -->	Y2_A
Y2-H -->	Y2_H
Y2-W -->	Y2_W

Reset channel names

Events per sample:

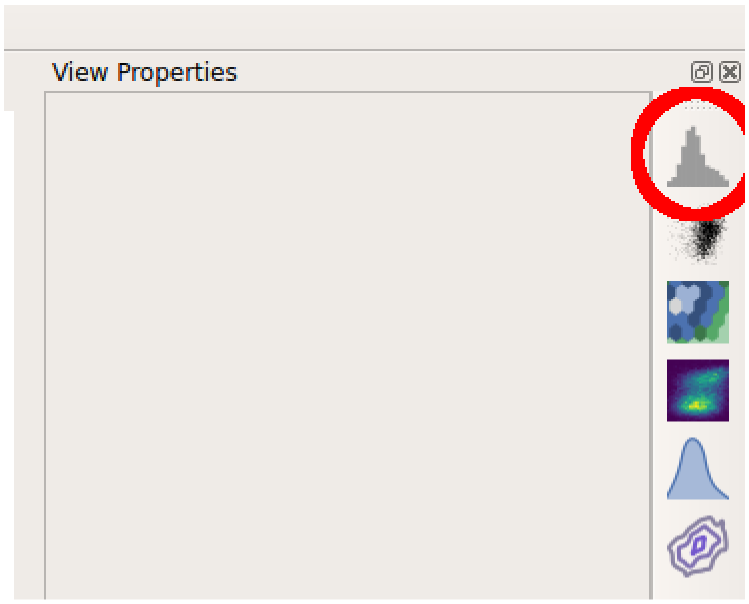
Samples: 2

Events: 20000

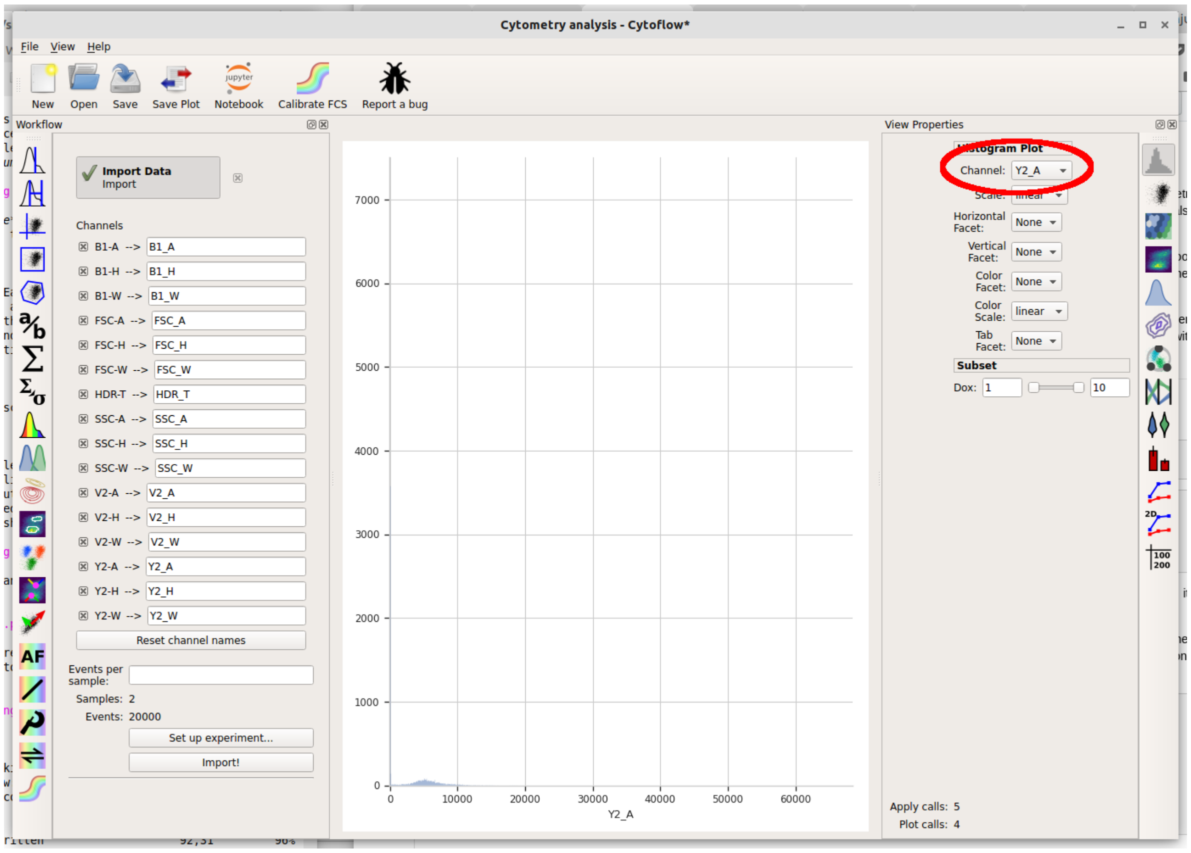
Set up experiment

Import!

Let’s plot the data. The right panel shows your view settings; choose the Histogram plot button.



The controls in the right panel are the view parameters. We won't get a plot until we choose a channel to view. Pull down the *Channel* selector and choose "Y2_A".



Import Data
Import

Channels

B1-A --> B1_A
B1-H --> B1_H
B1-W --> B1_W
FSC-A --> FSC_A
FSC-H --> FSC_H
FSC-W --> FSC_W
HDR-T --> HDR_T
SSC-A --> SSC_A
SSC-H --> SSC_H
SSC-W --> SSC_W
V2-A --> V2_A
V2-H --> V2_H
V2-W --> V2_W
Y2-A --> Y2_A
Y2-H --> Y2_H
Y2-W --> Y2_W

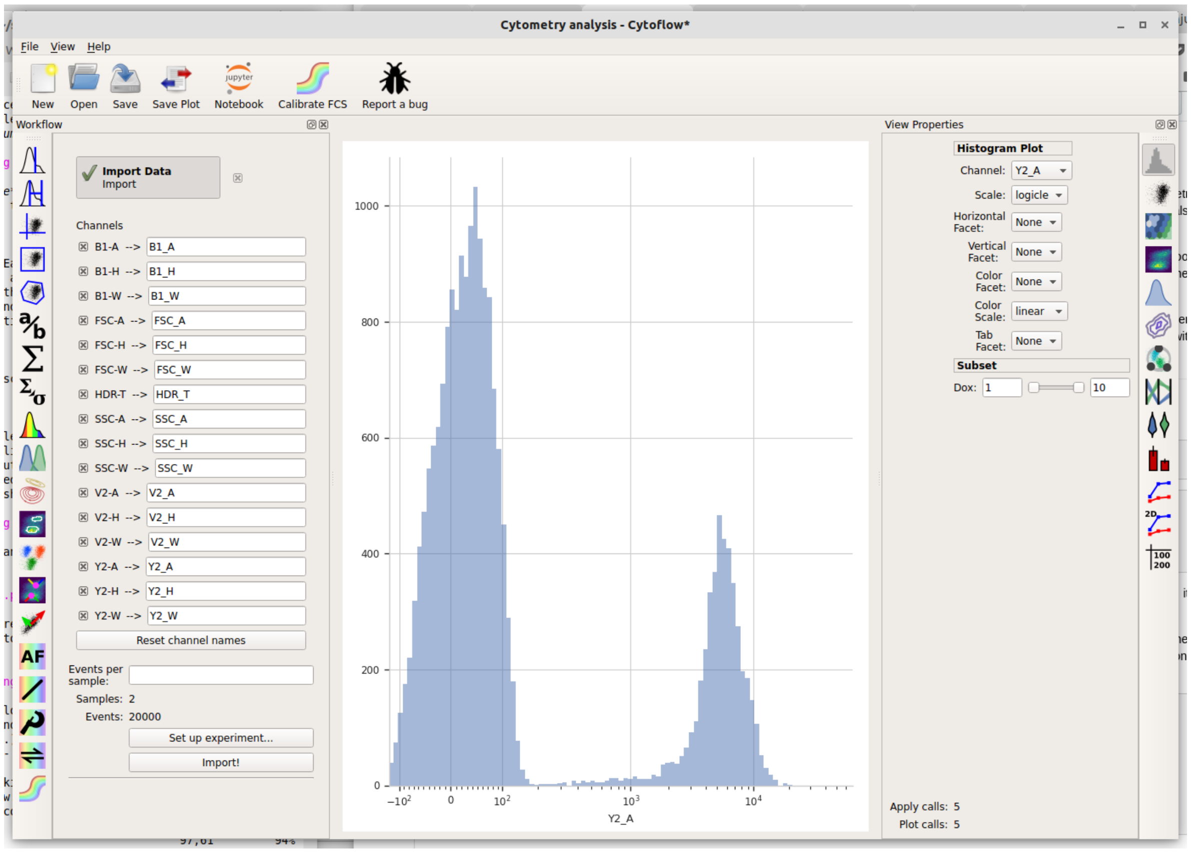
Events per sample:
Samples: 2
Events: 20000

Histogram Plot
Channel: Y2_A
Scale: linear
Horizontal Facet: None
Vertical Facet: None
Color Facet: None
Color Scale: linear
Tab Facet: None

Subset
Dox: 1 10

Apply calls: 5
Plot calls: 4

Cytoflow immediately displays a plot. Unfortunately, most of the data in this channel is clustered around 0, which makes it difficult to see using a linear scale (the default.) We can use a different scale by selecting the "Scale" option – choose "logicle" instead.



Import Data
Import

Channels

B1-A --> B1_A
B1-H --> B1_H
B1-W --> B1_W
FSC-A --> FSC_A
FSC-H --> FSC_H
FSC-W --> FSC_W
HDR-T --> HDR_T
SSC-A --> SSC_A
SSC-H --> SSC_H
SSC-W --> SSC_W
V2-A --> V2_A
V2-H --> V2_H
V2-W --> V2_W
Y2-A --> Y2_A
Y2-H --> Y2_H
Y2-W --> Y2_W

Reset channel names

Events per sample:
Samples: 2
Events: 20000

Set up experiment...
Import!

Histogram Plot

Channel: Y2_A
Scale: logicle
Horizontal Facet: None
Vertical Facet: None
Color Facet: None
Color Scale: linear
Tab Facet: None

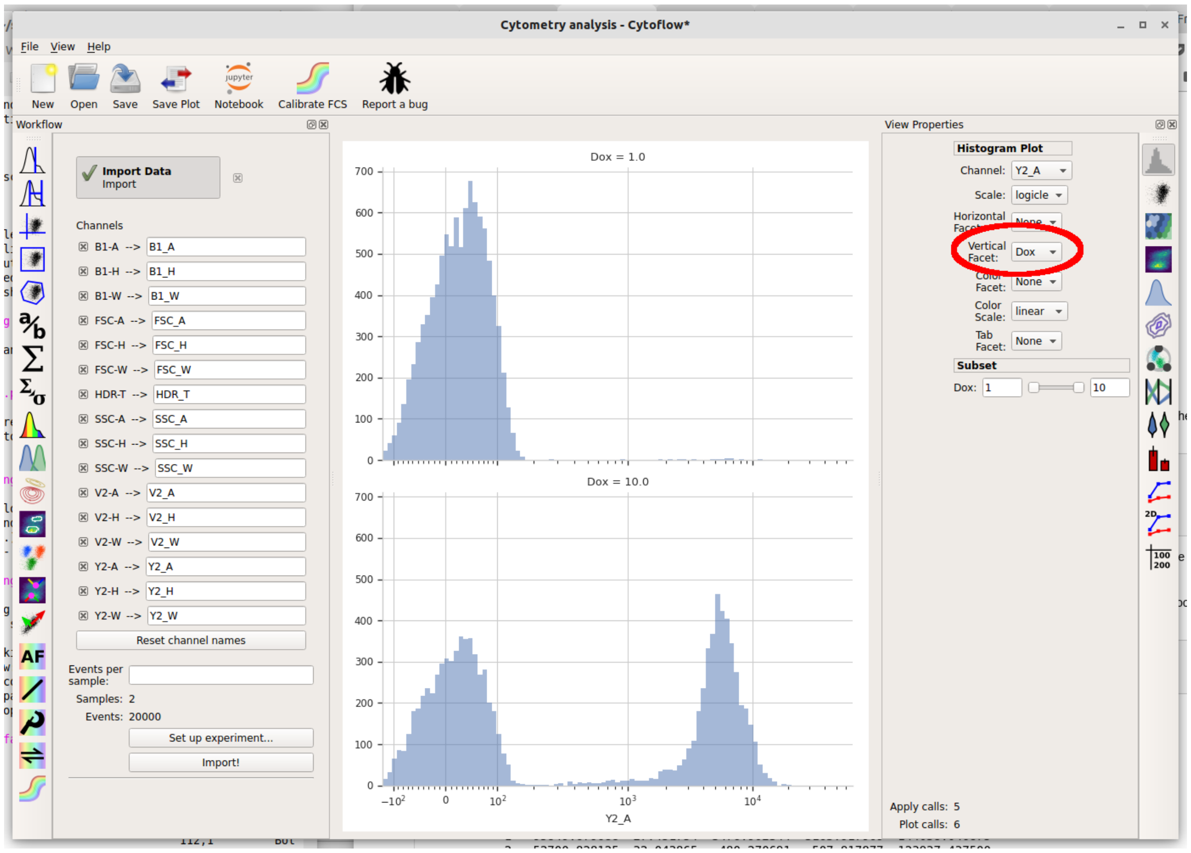
Subset

Dox: 1 10

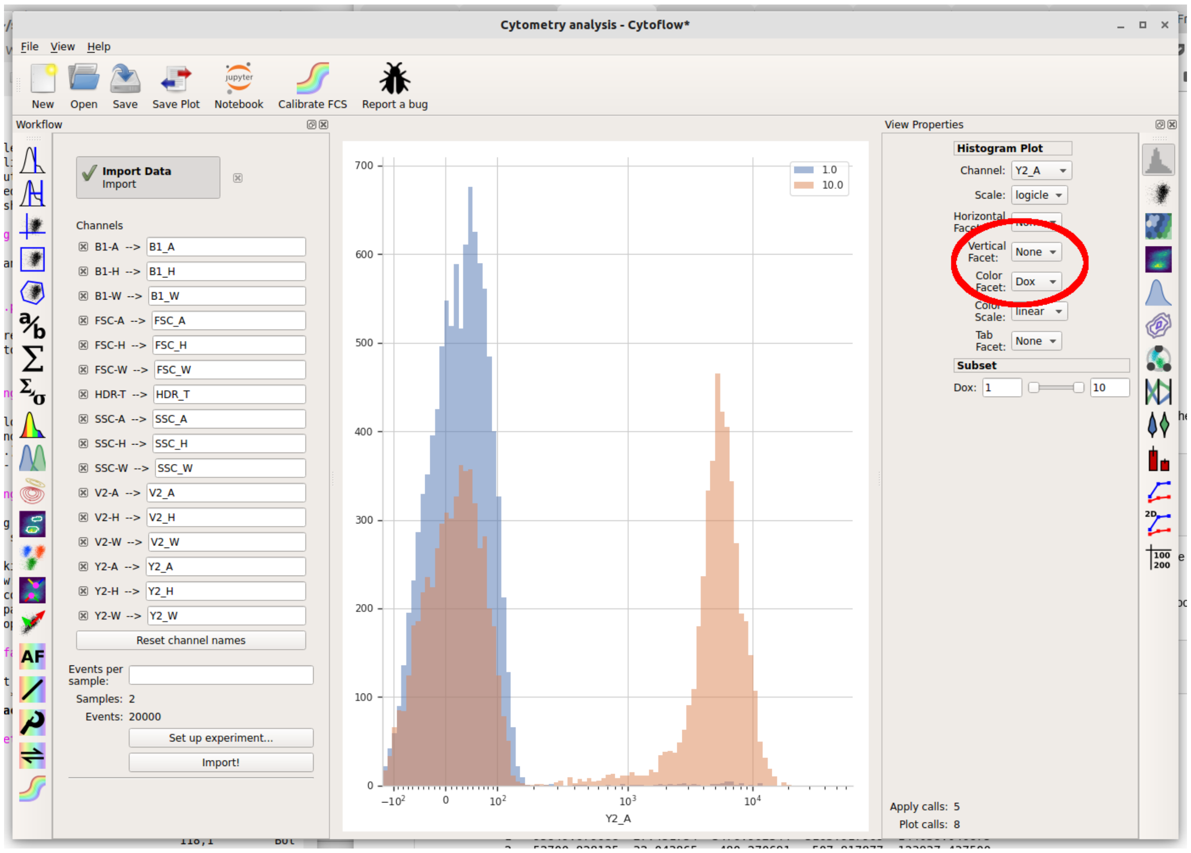
Apply calls: 5
Plot calls: 5

The *logicle* scale is interesting – it's linear around 0 and logarithmic elsewhere, and we can immediately see that this data is bimodal.

But wait – which tube are we looking at? Here we encounter one of the central principles of Cytoflow: **by default, we're looking at the entire data set.** This data comes from both tubes, not just one. We can tell Cytoflow to make a separate plot for each Dox concentration by changing the *Vertical facet* option to "Dox".



We can also plot the two different Dox concentrations on the same plot using different colors by setting *Color Facet* to “Dox” instead. Note that **you must change Vertical Facet back to “None” as well.**



Channels
B1-A --> B1_A
B1-H --> B1_H
B1-W --> B1_W
FSC-A --> FSC_A
FSC-H --> FSC_H
FSC-W --> FSC_W
HDR-T --> HDR_T
SSC-A --> SSC_A
SSC-H --> SSC_H
SSC-W --> SSC_W
V2-A --> V2_A
V2-H --> V2_H
V2-W --> V2_W
Y2-A --> Y2_A
Y2-H --> Y2_H
Y2-W --> Y2_W

Events per sample:
Samples: 2
Events: 20000

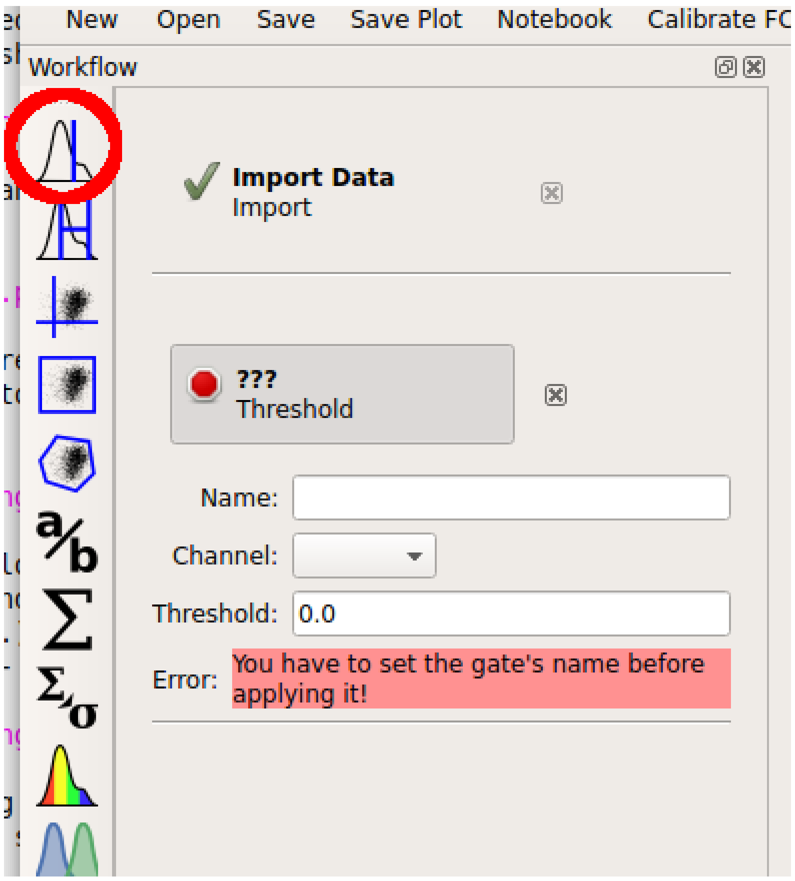
Histogram Plot
Channel: Y2_A
Scale: logicle
Horizontal Facet: None
Vertical Facet: None
Color Facet: Dox
Color Scale: linear
Tab Facet: None

Subset
Dox: 1 10

Apply calls: 5
Plot calls: 8

Basic gating

So there's a clear difference between the two tubes: one has a substantial population above ~200 in the Y2-A channel and the other doesn't. What is the proportion of "high" cell in each tube? To count these two populations, we first have to use a gate to separate them. Let's use a threshold gate. First, make one by choosing the threshold gate on the operations toolbar:



Import Data
Import

???
Threshold

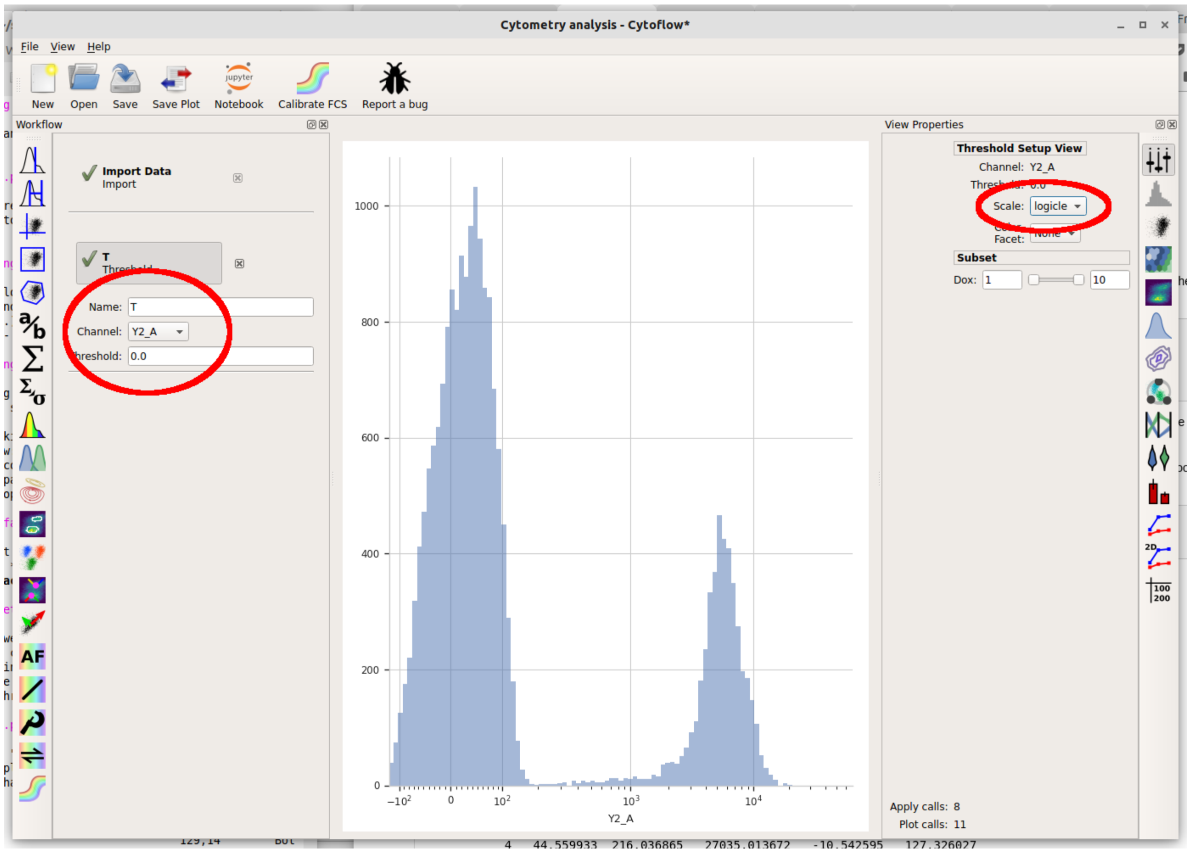
Name:

Channel:

Threshold: 0.0

Error: You have to set the gate's name before applying it!

Then, set the operation's name to "T" and the channel to "Y2-A". Note that when you set the channel, a plot appears in the central pane. It's on a linear scale again – change it to "logicle" on the plot options pane.

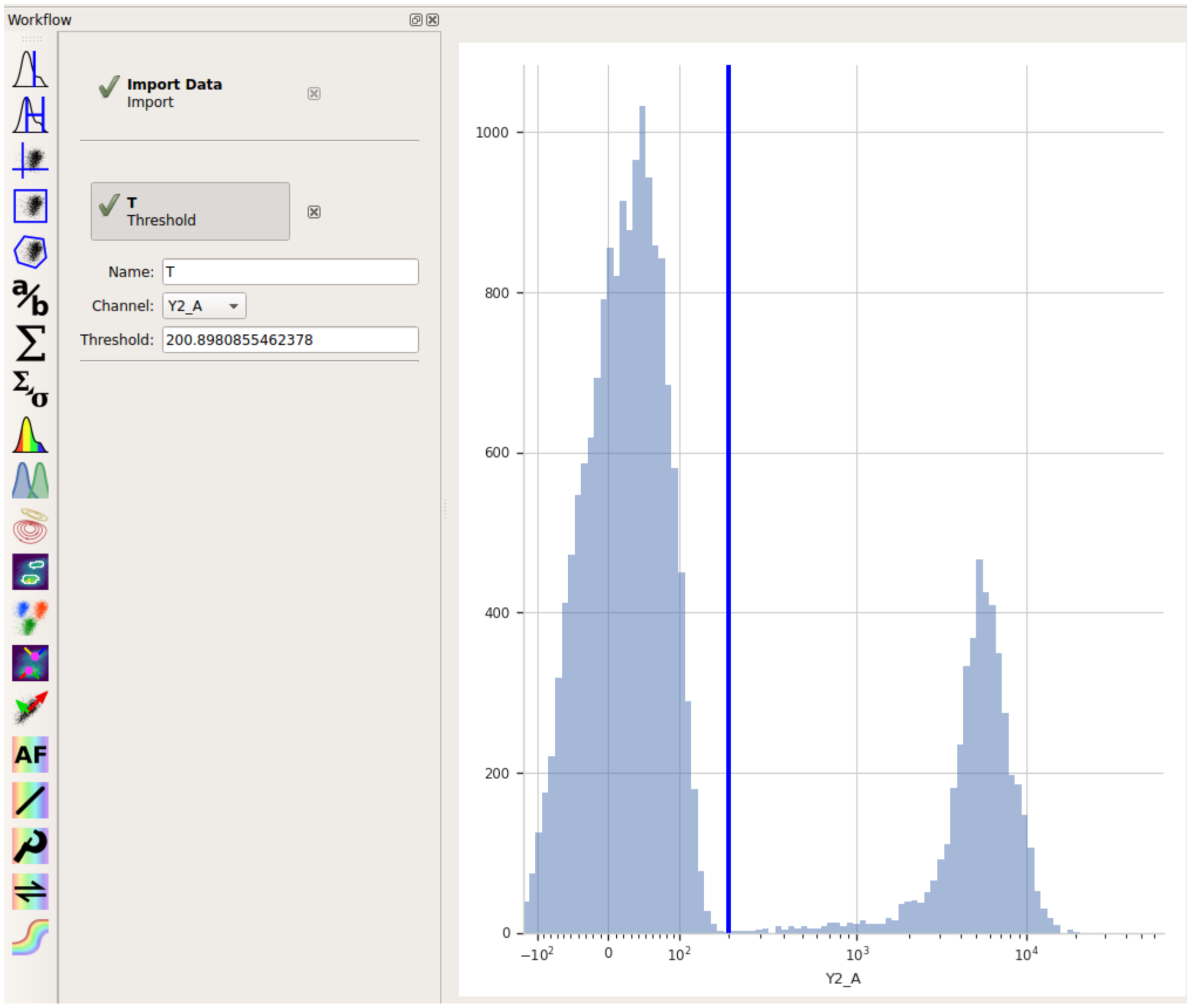


Name: T
Channel: Y2_A
Threshold: 0.0

Channel: Y2_A
Threshold: 0.0
Scale: logicle
Facet: None
Dox: 1 10

Apply calls: 8
Plot calls: 11

Note that when you move your mouse across the center pane, you now get a blue cursor that follows it. You can set the threshold by clicking on the pane. Choose a value about 200 (ie, one tic above the 10^2 major label.)


Workflow

Import Data
Import

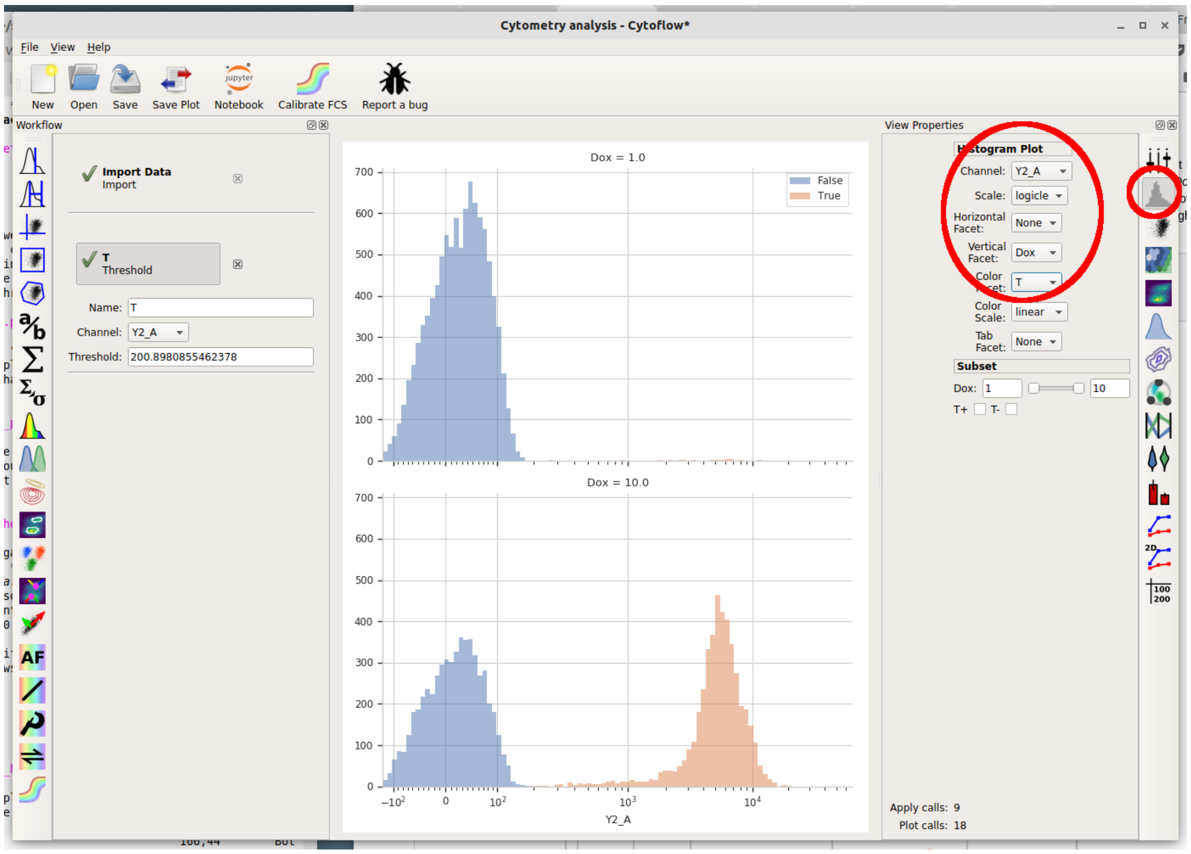
T
Threshold

Name: T
Channel: Y2_A
Threshold: 200.8980855462378

When you created a new Threshold gate, *you added a new condition to the data set.* This condition is *exactly like the "Dox" condition you set up when you imported your data.* That is, now there are some events that are `Dox = 1 and T = True`, some events that are `Dox = 1 and T = False`, some events that are `Dox = 10 and T = True`, and some events that are `Dox = 10 and T = False`.

You can get a good feel for this if you make a new Histogram. Set the histogram parameters as follows:

- `Channel = "Y2_A"`
- `Scale = "logicle"`
- `Vertical Facet = "Dox"`
- `Color Facet = "T"`



Import Data
Import

T
Threshold

Name: T
Channel: Y2_A
Threshold: 200.8980855462378

Histogram Plot
Channel: Y2_A
Scale: logicle
Horizontal Facet: None
Vertical Facet: Dox
Color Facet: T
Color Scale: linear
Tab Facet: None

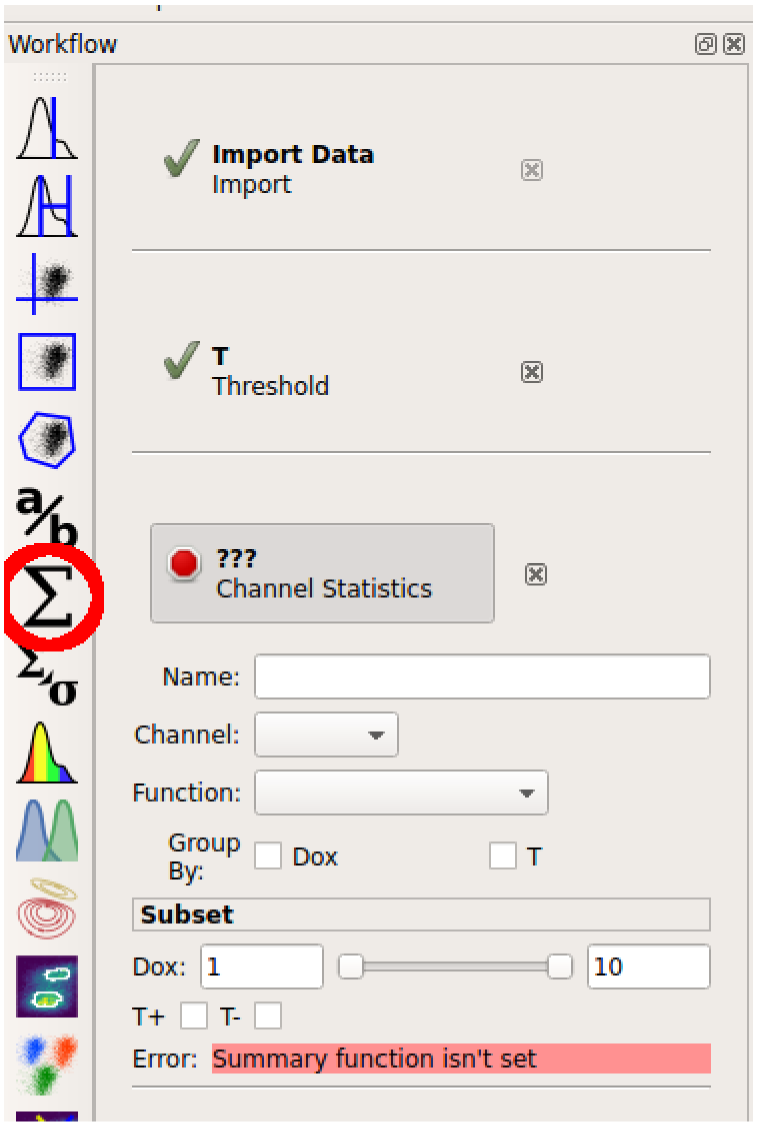
Subset
Dox: 1 10
T+ T-

Apply calls: 9
Plot calls: 18

What are we looking at? The two plots, top and bottom, represent the different Dox amounts (look at the titles!) Each is showing the "high" and "low" populations we identified with the Threshold gate in different colors. Play around with the different facets until you are comfortable with what does what. Also poke at the "subset" controls. (Don't worry, you won't break anything!)

Basic statistics

Cytoflow's reason for existing is to let you do quantitative flow cytometry. So lets quanitate those populations – how many events are in each of them? Once you've identified populations, Cytoflow lets you compute a number of *summary statistics* about each population, then graph statistics. To create a new statistic, choose the large "sigma" button on the operations toolbar, which creates a new Channel Statistic operation.



Import Data
Import

T
Threshold

???
Channel Statistics

Name:
Channel:
Function:
Group By: Dox T

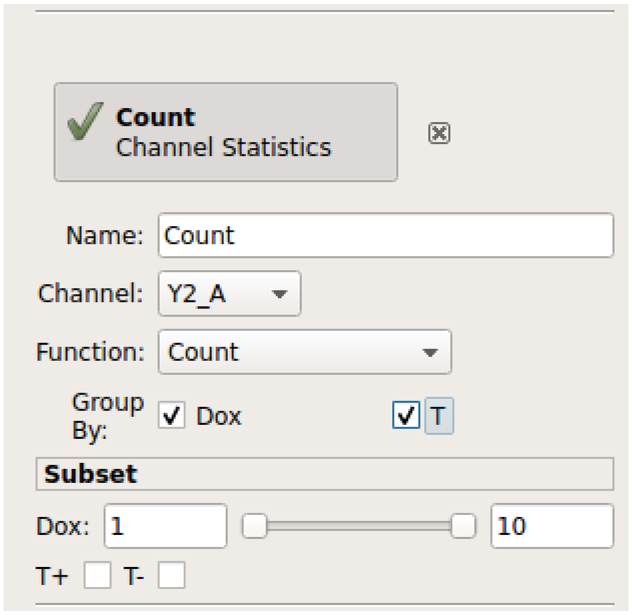
Subset

Dox: 1 10

T+ T-

Error: Summary function isn't set

Set the name of the new statistic to “Count”. Choose the “Y2_A” channel, and set the “Function” to “Count”. Under “Group by”, check *both* the “Dox” and “T” tic boxes.



Name: Count

Channel: Y2_A

Function: Count

Group By: [✓] Dox [✓] T

Subset

Dox: 1 10

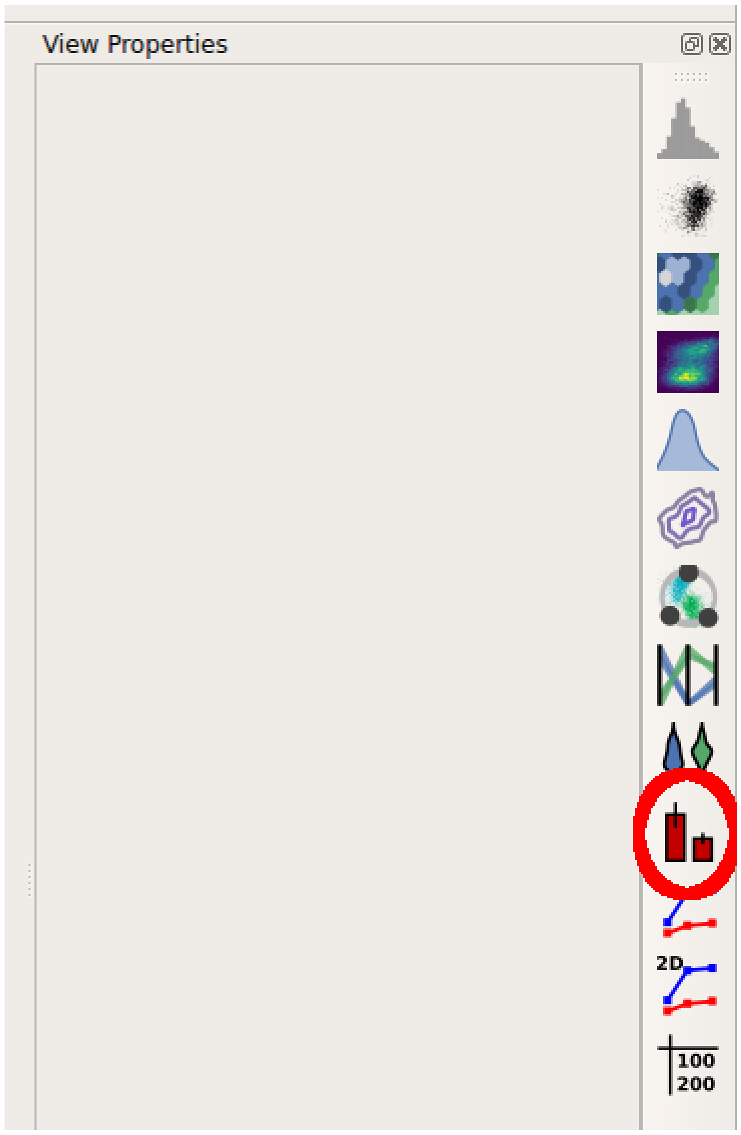
T+ [ ] T- [ ]

The "Group by" settings are particularly important. You're telling Cytoflow *which groups you want to compute the function on.* Cytoflow will break your data set up into unique combinations of all of these variables (which could be experimental conditions, like "Dox", or gates, like "T", or other subsets from other operations) and compute the function for each unique subset. So, what we've asked Cytoflow to do is break the data into four subsets:

- Dox = 1 and T = True
- Dox = 1 and T = False
- Dox = 10 and T = True
- Dox = 10 and T = False

and then compute the "Count" function on each subset.

Finally, let's plot that summary statistic. Choose the bar plot from the Views toolbar:

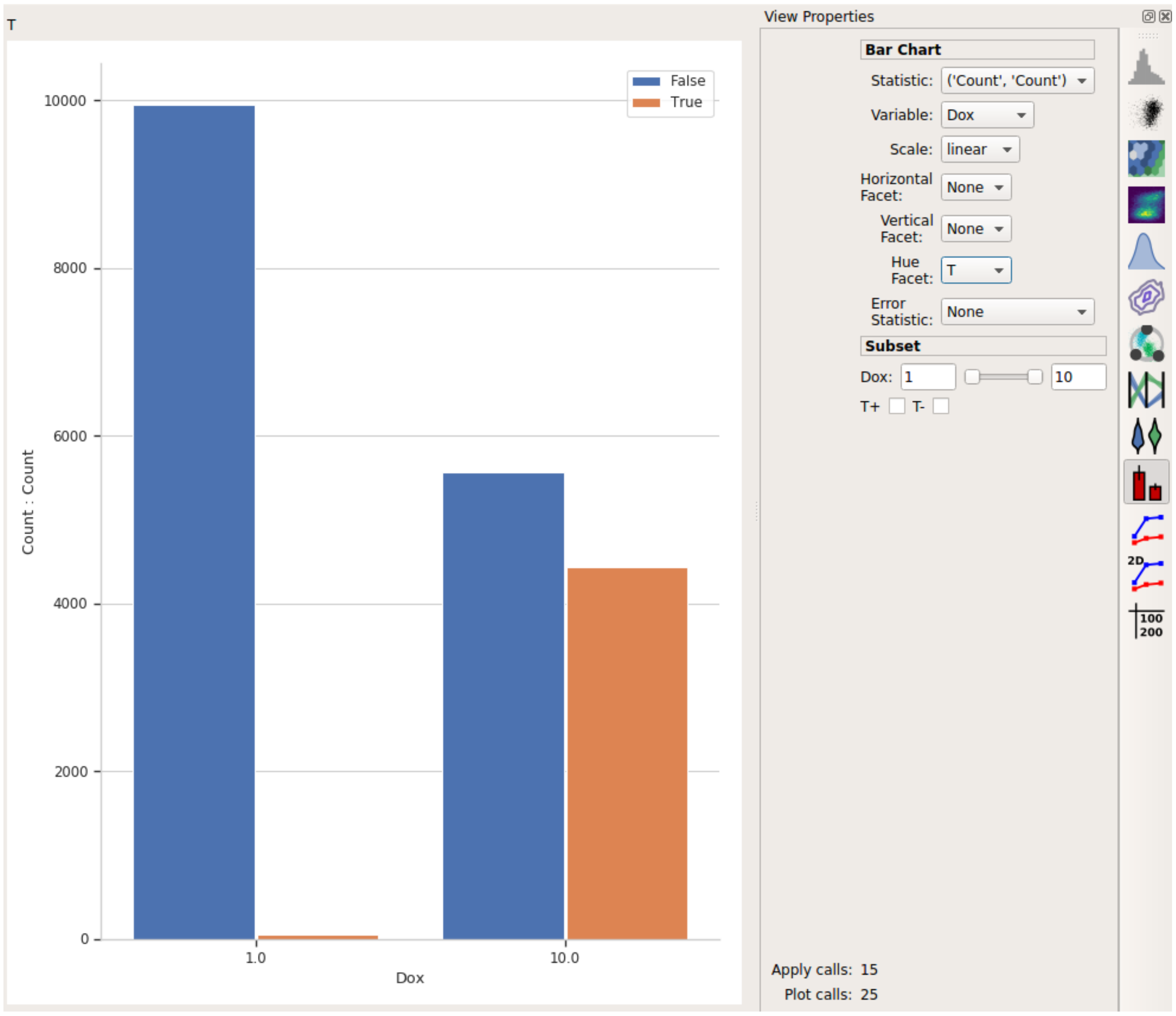


Set the view parameters as follows:

- Statistic = ('Count', 'Count')

  Note: the new statistic is called ('Count', 'Count') because the channel statistic operaton's *name* was "Count" and the *function* you applied was also named "Count".

- Variable = "Dox"
- Hue Facet = "T"

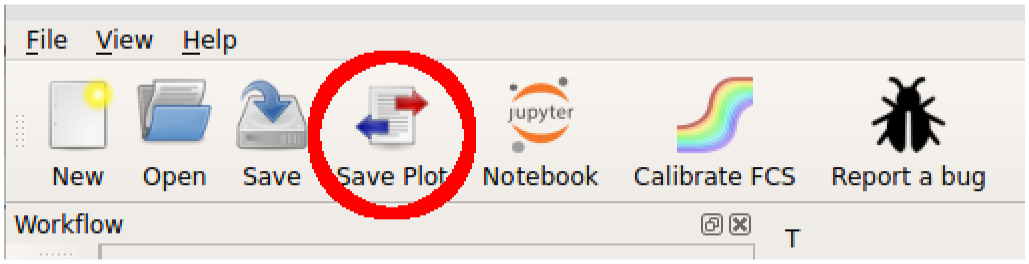


Bar Chart
Statistic: ('Count', 'Count')
Variable: Dox
Scale: linear
Horizontal Facet: None
Vertical Facet: None
Hue Facet: T
Error Statistic: None
Subset
Dox: 1 10
T+ T-
Apply calls: 15
Plot calls: 25

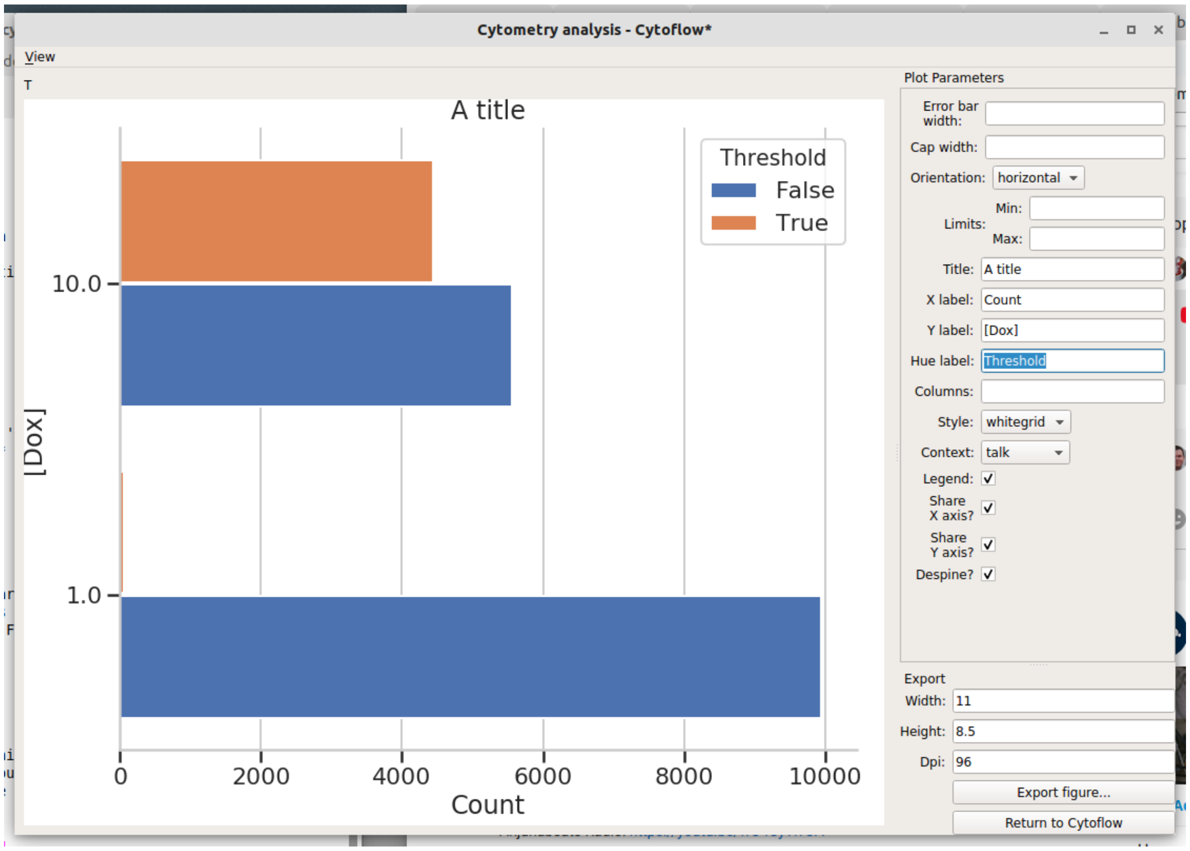
This is the bar plot we wanted: comparing different Dox levels (the two bars on the left vs. the two bars on the right) and how many events were below the threshold (T = False, in blue) vs how many were above it (T = True, in orange.)

Export the plot

I like to think that Cytoflow's graphics are nice-looking. Possibly nice enough to publish! (Also, if you publish using Cytoflow, please cite it!) To export the plot, choose "Save plot…" from the toolbar at the top.



In this dialog, you can set many of the visual parameters for the plot, such as the axis labels and plot title. You can also export the figure with a given size (in inches) and resolution (in dots-per-inch) by clicking "Export figure…".



Plot Parameters

Error bar width:
Cap width:
Orientation: horizontal
Limits: Min: Max:
Title: A title
X label: Count
Y label: [Dox]
Hue label: Threshold
Columns:
Style: whitegrid
Context: talk
Legend: ✓
Share X axis? ✓
Share Y axis? ✓
Despine? ✓

Export
Width: 11
Height: 8.5
Dpi: 96

To return to Cytoflow, click "Return to Cytoflow".

5.1.2 Tutorial: Dose-Response

A common way to use flow cytometry is to analyze a dose-response experiment: cells were treated with increasing doses of some drug or compound, and we want to see how the response changed as we increased the amount of the compound. In this case, I'm treating an engineered yeast line with isopentyladenine, or IP; the yeast cells are engineered with a basic GFP reporter that is expressed in response to IP. We measured GFP fluorescence after 12 hours, at which time we expect the cells to be at steady-state.

(The experiment is described in more detail here: Chen et al, Nature Biotech 2005)

If you'd like to follow along, you can do so by downloading one of the **cytoflow-#####-examples-basic.zip** files from the Cytoflow releases page on GitHub.

Importing Data

Start `Cytoflow`. A workflow always starts with an **Import Data** operation; click the **Set up experiment button...**

Remember that we need to tell `Cytoflow` about the experimental conditions for each sample we're analysing. In this case, we only have one experimental variable, *IP*.

- Click **Add a variable**
- Change its type to **Number** and its name to **IP**.
- Click **Add tubes...**

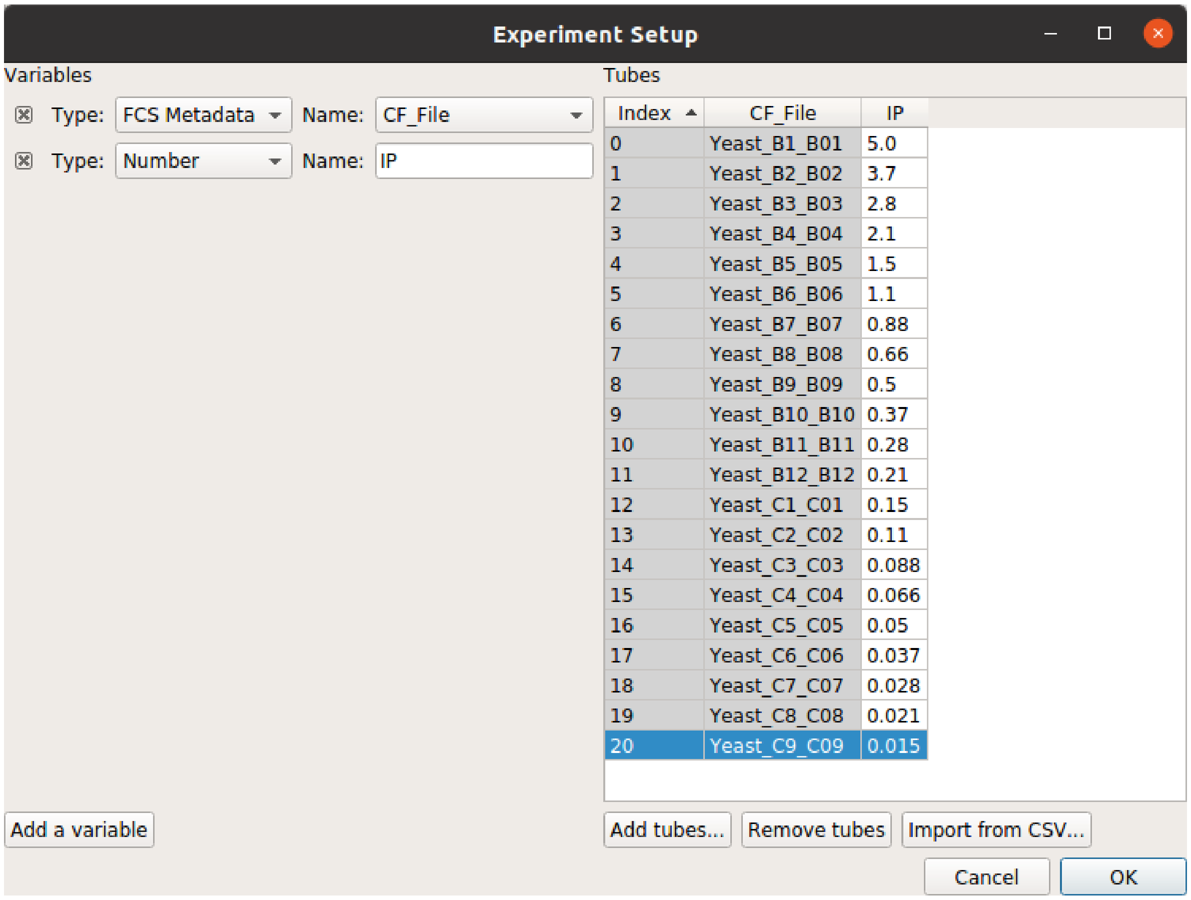
- Select all of the tubes whose names start with *Yeast_B1* through *Yeast_C9*.

Note: Remember, you can select multiple files by holding down the *Control* or *Command* key.

- Fill in the experimental values for the **IP** column. I did a serial dilution; use the the table below for reference.

File	IP
Yeast_B1_B01.fcs	5.0
Yeast_B2_B02.fcs	3.7
Yeast_B3_B03.fcs	2.8
Yeast_B4_B04.fcs	2.1
Yeast_B5_B05.fcs	1.5
Yeast_B6_B06.fcs	1.1
Yeast_B7_B07.fcs	0.88
Yeast_B8_B08.fcs	0.66
Yeast_B9_B09.fcs	0.5
Yeast_B10_B10.fcs	0.37
Yeast_B11_B11.fcs	0.28
Yeast_B12_B12.fcs	0.21
Yeast_C1_C01.fcs	0.15
Yeast_C2_C02.fcs	0.11
Yeast_C3_C03.fcs	0.089
Yeast_C4_C04.fcs	0.066
Yeast_C5_C05.fcs	0.05
Yeast_C6_C06.fcs	0.037
Yeast_C7_C07.fcs	0.028
Yeast_C8_C08.fcs	0.021
Yeast_C9_C09.fcs	0.015

At the end, your table should look like this:



Type: FCS Metadata Name: CF_File

Type: Number Name: IP

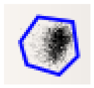
Index	CF_File	IP
0	Yeast_B1_B01	5.0
1	Yeast_B2_B02	3.7
2	Yeast_B3_B03	2.8
3	Yeast_B4_B04	2.1
4	Yeast_B5_B05	1.5
5	Yeast_B6_B06	1.1
6	Yeast_B7_B07	0.88
7	Yeast_B8_B08	0.66
8	Yeast_B9_B09	0.5
9	Yeast_B10_B10	0.37
10	Yeast_B11_B11	0.28
11	Yeast_B12_B12	0.21
12	Yeast_C1_C01	0.15
13	Yeast_C2_C02	0.11
14	Yeast_C3_C03	0.088
15	Yeast_C4_C04	0.066
16	Yeast_C5_C05	0.05
17	Yeast_C6_C06	0.037
18	Yeast_C7_C07	0.028
19	Yeast_C8_C08	0.021
20	Yeast_C9_C09	0.015

Note: Filling out these tables can be a pain, especially if you've already got this information in a table somewhere else already. If so, you can actually import the table directly, following the instructions at *HOWTO: Import an experiment from a table*

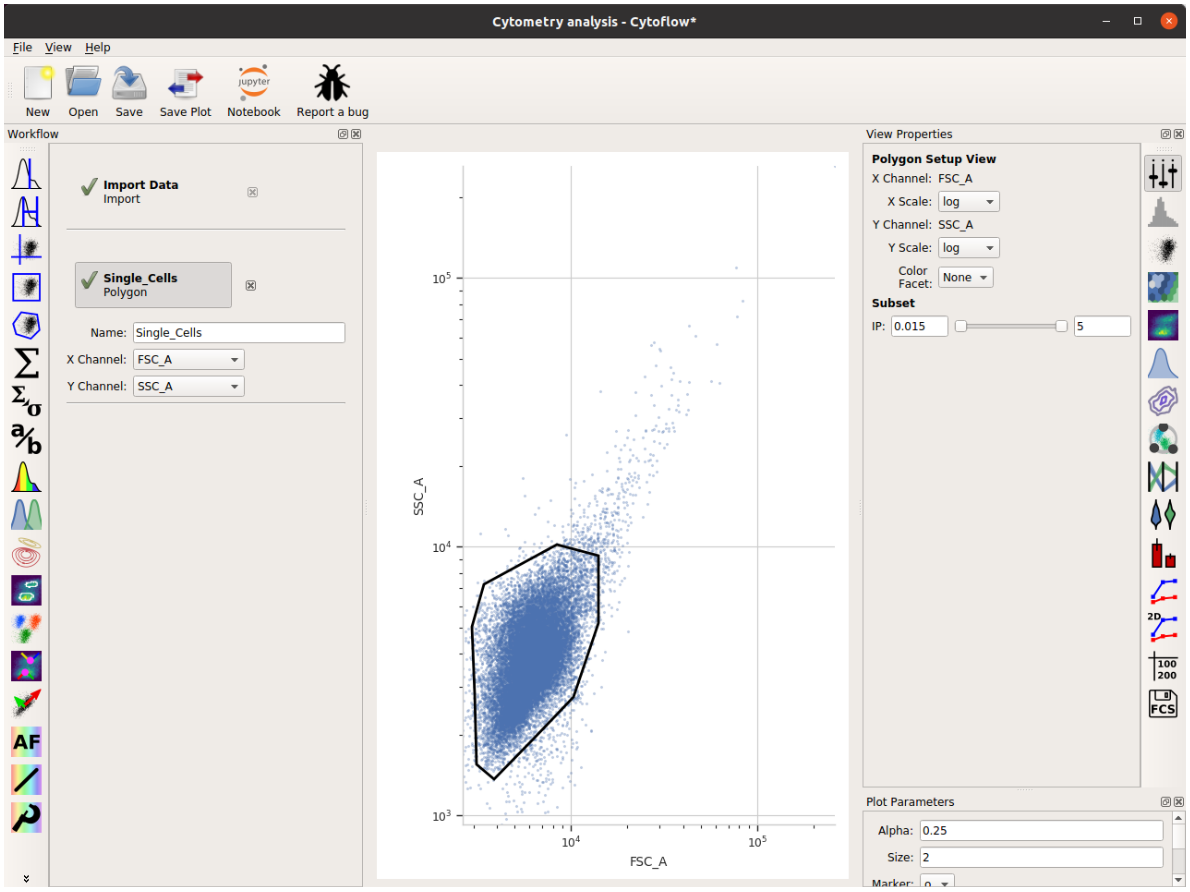
Click **Import!** to import the data.

Filter out clumps and debris

Because these cultures were grown on a roller drum, they are quite uniform in size – but there are still some clumps. Let's filter that out with a polygon gate.

- Click the polygon gate button on the operations toolbar: 
- Name the gate *Single_Cells* – so we can refer to this subset later – and set the X and Y channels to FSC_A and SSC_A. (These are the forward and side-scatter parameters.)
- The initial plot is hard to work with – on the View pane, change both the X and Y scale to **log**.
- Draw a polygon around the major population the center of the plot. Single-click to set a new vertex; double-click to close the polygon.

Here's what my window looks like now:



Import Data
Import

Single_Cells
Polygon

Name: Single_Cells
X Channel: FSC_A
Y Channel: SSC_A

Polygon Setup View
X Channel: FSC_A
X Scale: log
Y Channel: SSC_A
Y Scale: log
Color Facet: None
Subset
IP: 0.015 5

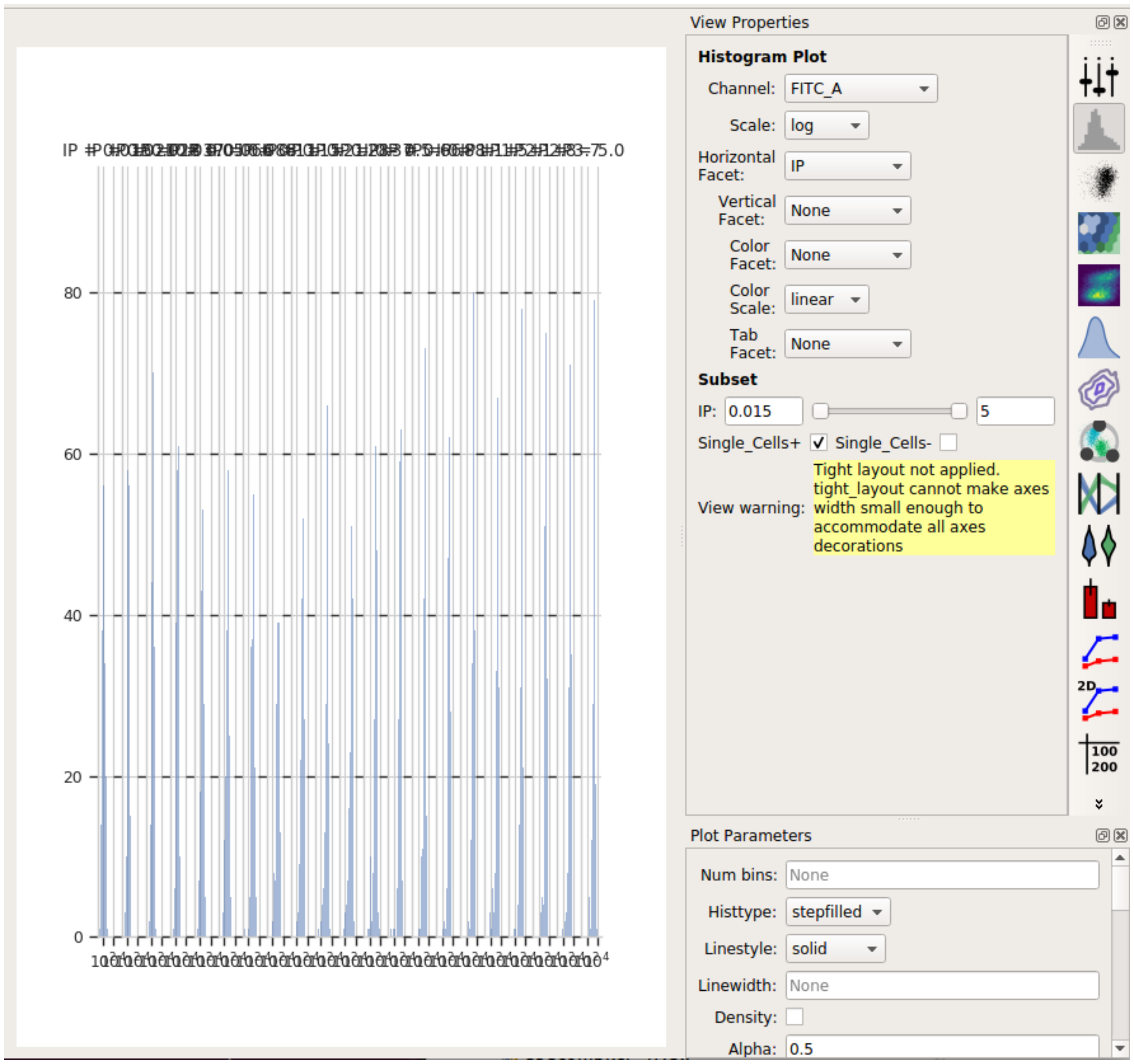
Alpha: 0.25
Size: 2
Marker: o

Look at the FITC_A channel

Let's use a histogram to see if we're seeing a dose-response.

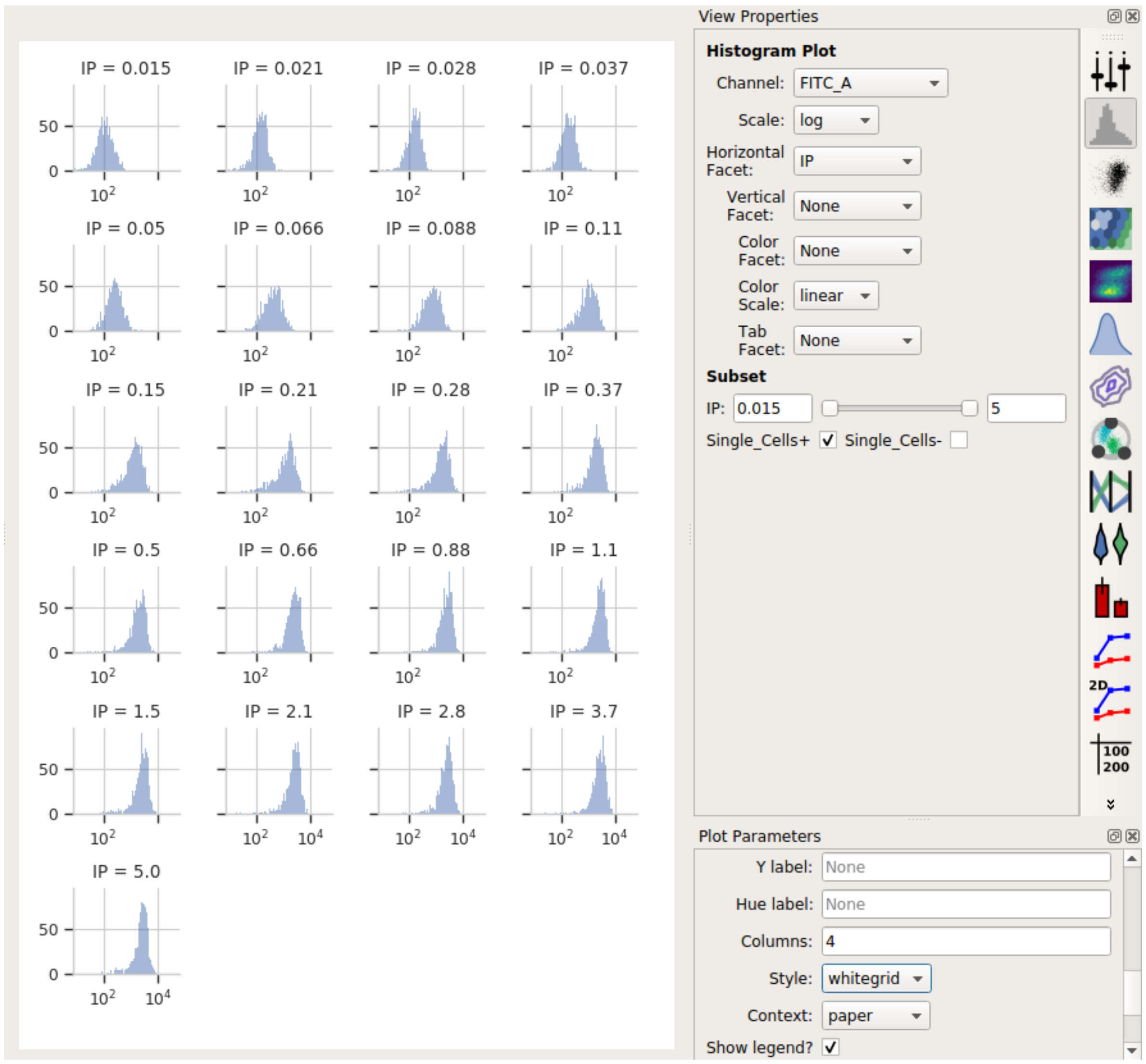
- Choose the histogram view:
- Set the channel to **FITC_A** and the scale to **log**
- Set the **Horizontal Facet** to **IP** – this will give us one plot for each different value of **IP**
- To only look at the cells in the **Single_Cells** gate, under **Subset**, click the check-box next to **Single_Cells+**

The result is the following plot:



View warning: Tight layout not applied. tight_layout cannot make axes width small enough to accommodate all axes decorations

We've got so many different values of **IP** that we can't squeeze them all next to eachother. However, we *can* ask `Cytoflow` to "wrap" them onto several lines, like the word-wrap on a word processor. To do so, in the **Plot Parameters** pane (bottom-right in the default layout), scroll down to **Columns** and set it to 4.



View Properties

Histogram Plot

Channel: FITC_A

Scale: log

Horizontal Facet: IP

Vertical Facet: None

Color Facet: None

Color Scale: linear

Tab Facet: None

Subset

IP: 0.015 — 5

Single_Cells+ ✓ Single_Cells- ☐

Plot Parameters

Y label: None

Hue label: None

Columns: 4

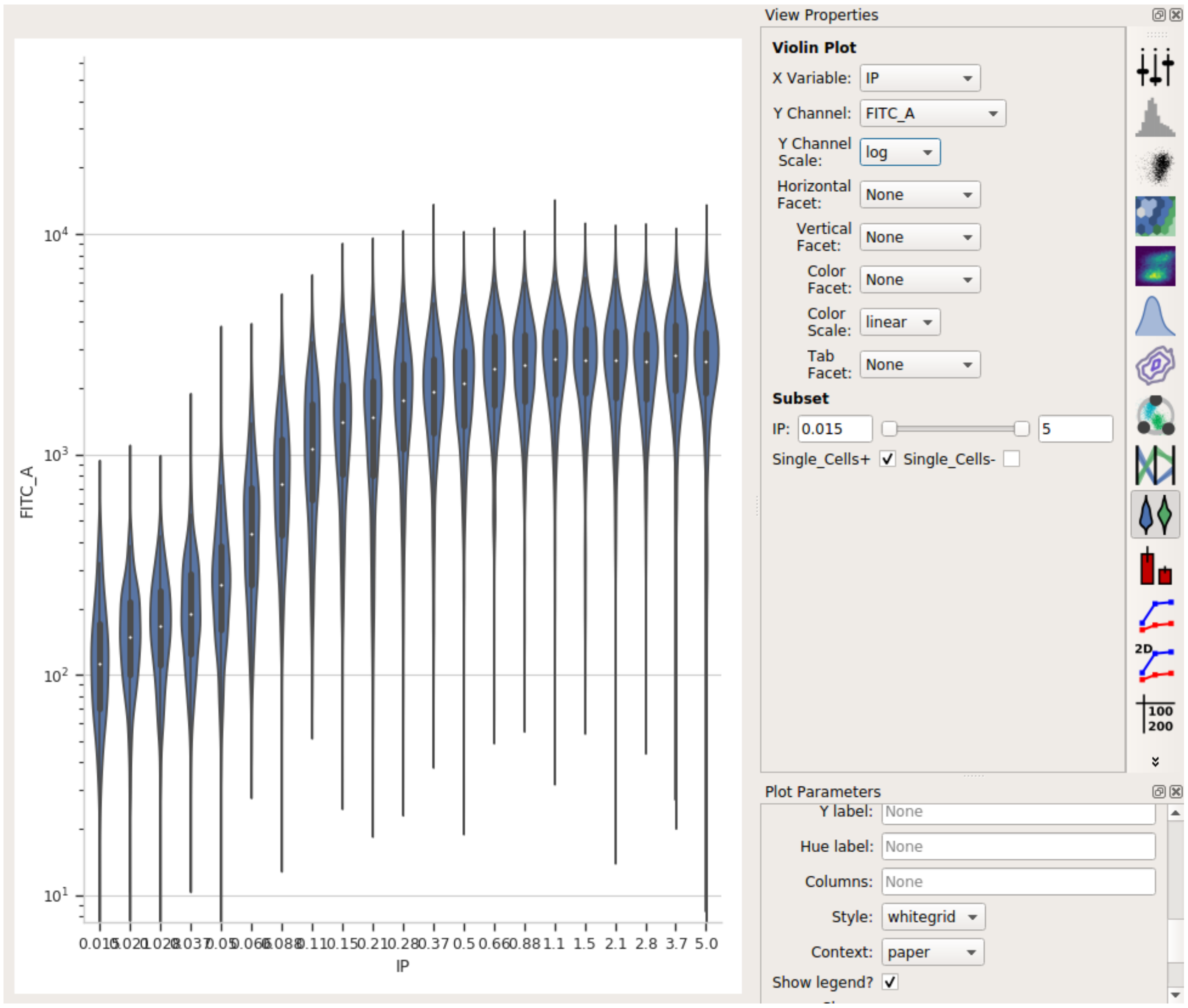
Style: whitegrid

Context: paper

Show legend? ✓

Ah, much better. We can see each plot, and we're clearly seeing an increase. However, histograms are kind of a terrible way to compare lots of distributions like this. A better way is a *violin plot*.

- Choose the violin plot view:
- Set the **X variable** to **IP**, the **Y variable** to **FITC_A**, and the **Y channel scale** to **log**.
- As above, set **Subset** to **Single_Cells+**



View Properties

Violin Plot

X Variable: IP

Y Channel: FITC_A

Y Channel Scale: log

Horizontal Facet: None

Vertical Facet: None

Color Facet: None

Color Scale: linear

Tab Facet: None

Subset

IP: 0.015 5

Single_Cells+ ✓ Single_Cells-

Plot Parameters

Y label: None

Hue label: None

Columns: None

Style: whitegrid

Context: paper

Show legend? ✓

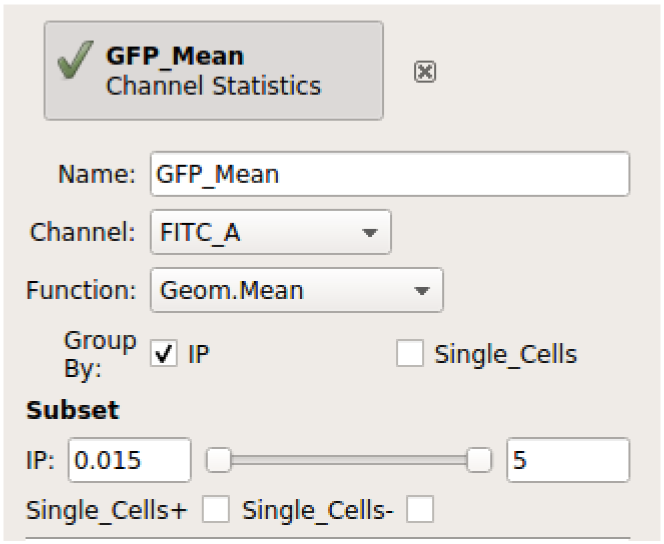
I love how a violin plot lets you compare distributions side-by-side. In this case, it's very clear that there's a clear dose-dependent response as IP concentration increases, as well as a clear saturation of the response.

Summarize the dose-response curve on a line plot

Next, let's make a "traditional" dose-response curve with a scatter plot, where the X axis shows the amount of IP and the Y axis shows the geometric mean of the **FITC_A** channel.

- Add a **Channel Statistics** operation: Σ
- Give the new statistic a name – I called it *GFP_Mean* – and choose the channel we want to analyze (*FITC_A*) and the function we want to apply (*Geom.Mean*)
- Now we need to tell `Cytoflow` which subsets of our data we want to apply the function to. We want the geometric mean computed for every different value of IP; so set **Group by** to **IP**.
- Again, we only want to analyze the cells in the *Single_Cells* gate – so set **Subset** to *Single_Cells+*.

At the end, your operation should look like this:



GFP_Mean
Channel Statistics

Name: GFP_Mean

Channel: FITC_A

Function: Geom.Mean

Group By: [✓ IP] [ ] Single_Cells

Subset

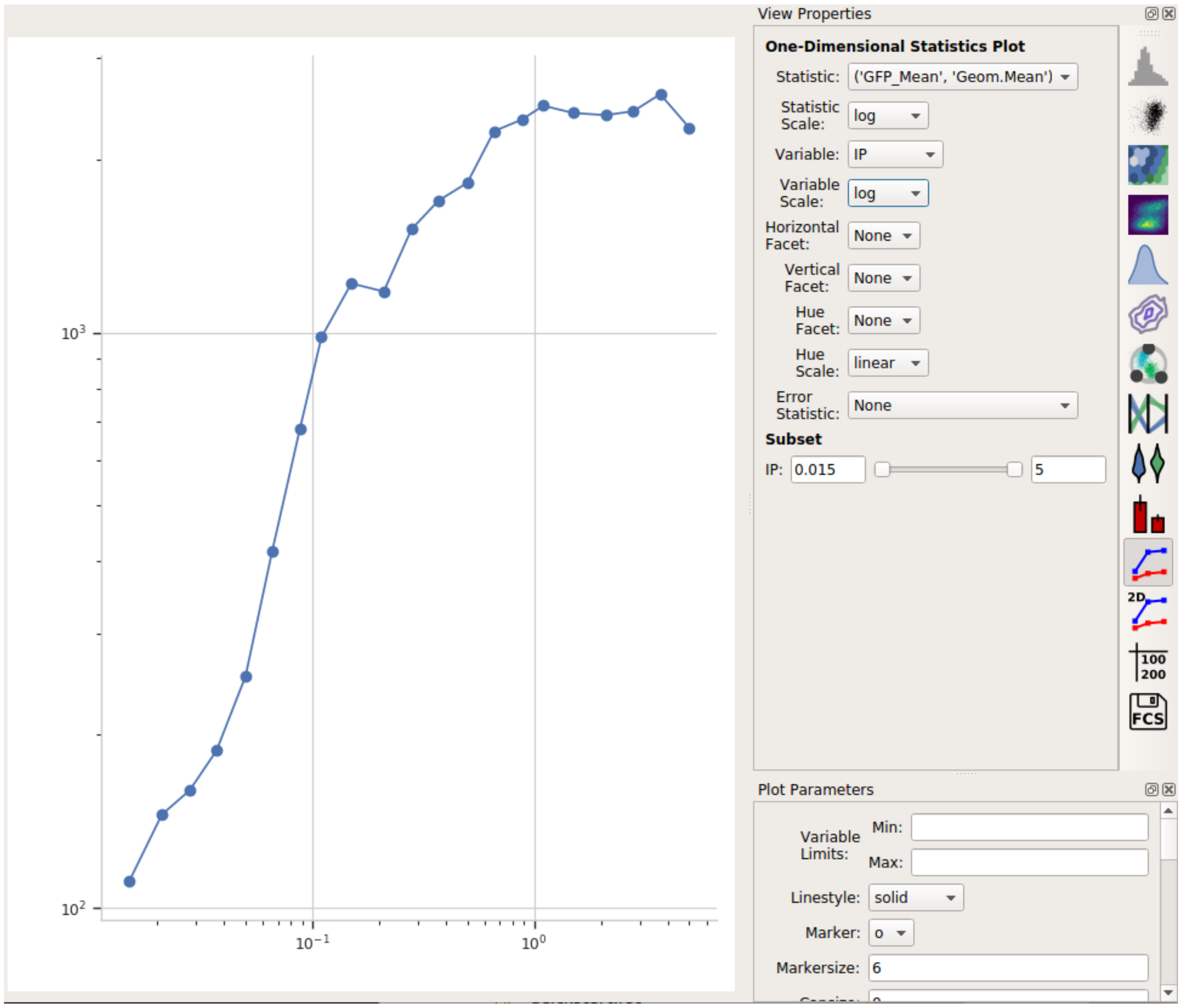
IP: 0.015 — 5

Single_Cells+ [ ] Single_Cells- [ ]

Now that we've made a new summary statistic, we want to plot it!

- Open the **1D Statistics View**:
- Set **Statistic** to the name of the statistic we just created: *('GFP_Mean', 'Geom.Mean')* (note that it shows us both the name of the operation that created the statistic, and the function that we used.)
- Set the **Statistic Scale** to **log**. This is how the plot will scale the Y axis.
- Set **Variable** to the variable we want on the X axis – in this case, *IP*.
- Set **Variable Scale** to *log* – this is the scale on the Y axis.

Et voila, a scatter plot:



View Properties

One-Dimensional Statistics Plot

Statistic: ('GFP_Mean', 'Geom.Mean')
Statistic Scale: log
Variable: IP
Variable Scale: log
Horizontal Facet: None
Vertical Facet: None
Hue Facet: None
Hue Scale: linear
Error Statistic: None

Subset

IP: 0.015 5

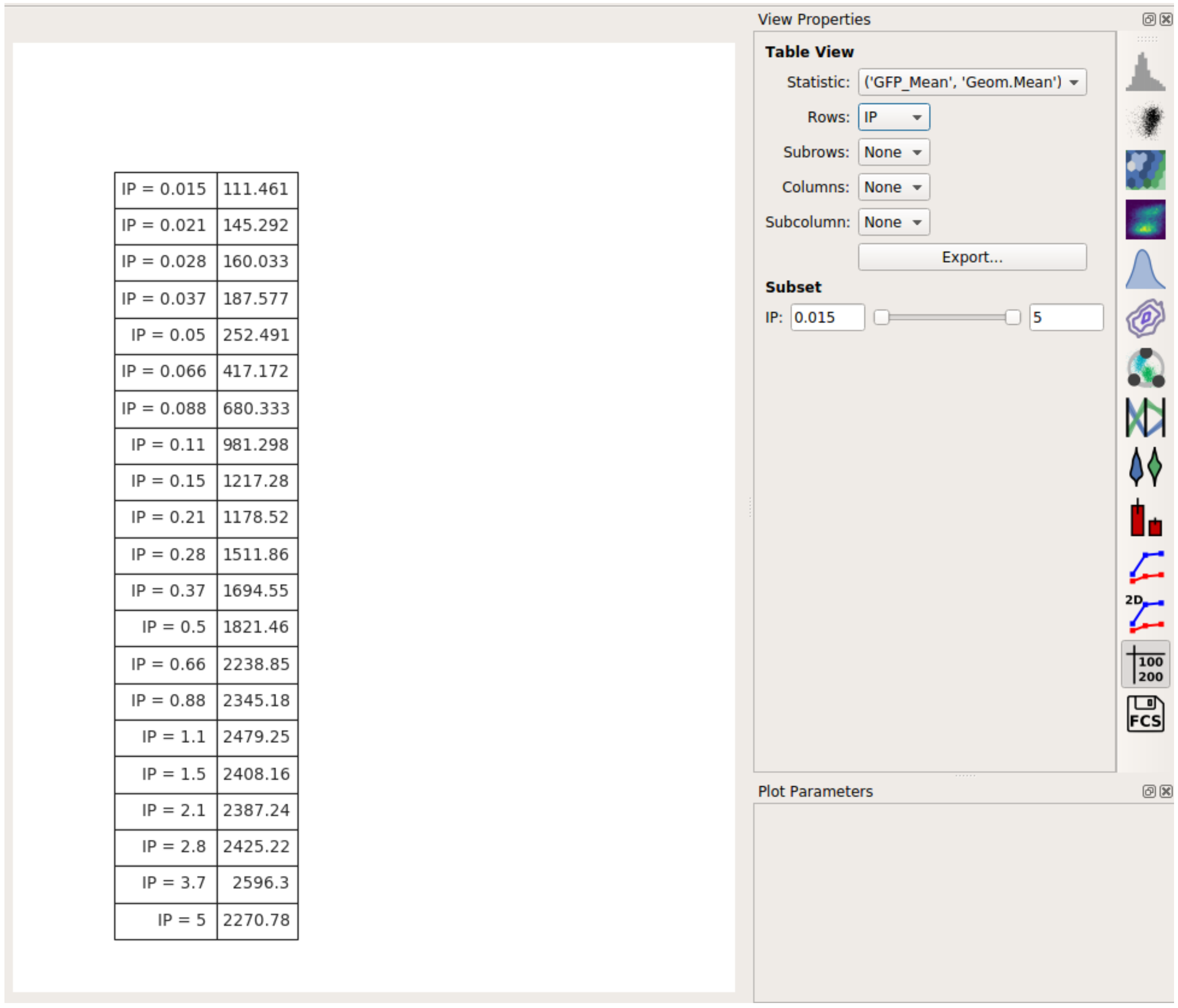
Plot Parameters

Variable Limits: Min: Max:
Linestyle: solid
Marker: o
Markersize: 6

Export the dose-response curve as a table

Often, we want this data avilable for downstream analyses. Any statistic you've computed, you can also export as a table (for imporing into a spreadsheet or other plotting or analysis tool.)

- Choose the **Table View**: 100 200
- Set **Statistic** to the same statistic we were just looking at: *('GFP_Mean', 'Geom.Mean')*
- Set *Row* to the variable you'd like to put on different rows. In this case, there's only one, so set it to *IP*.
- You can preview the table in the center plot pane. To export it to a CSV file, click **Export…**



IP = 0.015	111.461
IP = 0.021	145.292
IP = 0.028	160.033
IP = 0.037	187.577
IP = 0.05	252.491
IP = 0.066	417.172
IP = 0.088	680.333
IP = 0.11	981.298
IP = 0.15	1217.28
IP = 0.21	1178.52
IP = 0.28	1511.86
IP = 0.37	1694.55
IP = 0.5	1821.46
IP = 0.66	2238.85
IP = 0.88	2345.18
IP = 1.1	2479.25
IP = 1.5	2408.16
IP = 2.1	2387.24
IP = 2.8	2425.22
IP = 3.7	2596.3
IP = 5	2270.78

View Properties

Table View

Statistic: ('GFP_Mean', 'Geom.Mean')

Rows: IP

Subrows: None

Columns: None

Subcolumn: None

Export...

Subset

IP: 0.015 5

Plot Parameters

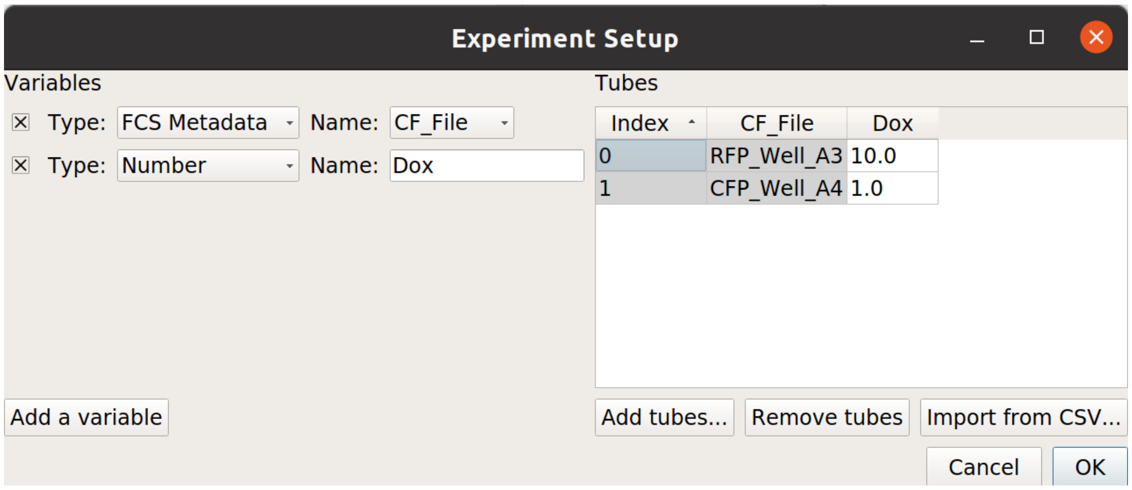
5.1.3 Tutorial: Machine Learning

One of the directions Cytoflow is going in that I'm most excited about is the application of advanced machine learning methods to flow cytometry analysis. After all, cytometry data is just a high-dimensional data set with many data points: making sense of it can take advantage of some of the sophisticated methods that have seen great success with other high-throughput biological data (such as microarrays.)

The following tutorial goes in depth on a common machine-learning method, Gaussian Mixture Models, then demonstrates with briefer examples some of the other machine learning methods that are implemented in Cytoflow.

Import the data

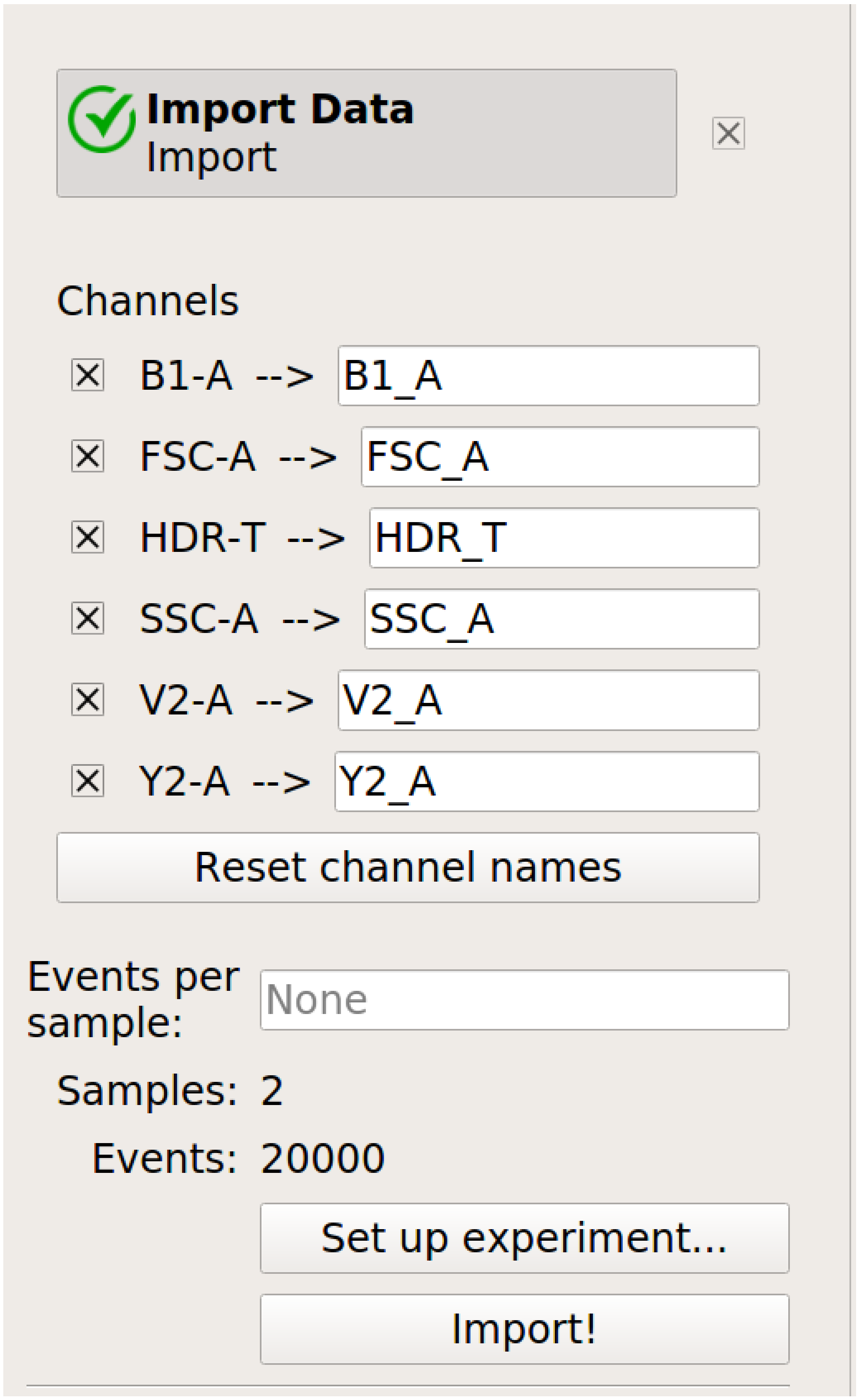
- Start Cytoflow. Under the **Import Data** operation, choose **Set up experiment…**
- Add one variable, *Dox*. Make it a *Number*.
- From the *examples-basic* data set, import *RFP_Well_A3.fcs* and *CFP_Well_A4.fcs*. Assign them *Dox = 10.0* and *Dox = 1.0*, respectively. Your setup should look like this:



Type: FCS Metadata Name: CF_File
Type: Number Name: Dox

Index	CF_File	Dox
0	RFP_Well_A3	10.0
1	CFP_Well_A4	1.0

- Choose **OK**
- Optional: Remove the **-W** and **-H** channels from the **Import Data** operation; we're only using the **-A** channels. Your **Import Data** operation should look like this:



Channels

☒	B1-A -->	B1_A
☒	FSC-A -->	FSC_A
☒	HDR-T -->	HDR_T
☒	SSC-A -->	SSC_A
☒	V2-A -->	V2_A
☒	Y2-A -->	Y2_A

Events per sample: None

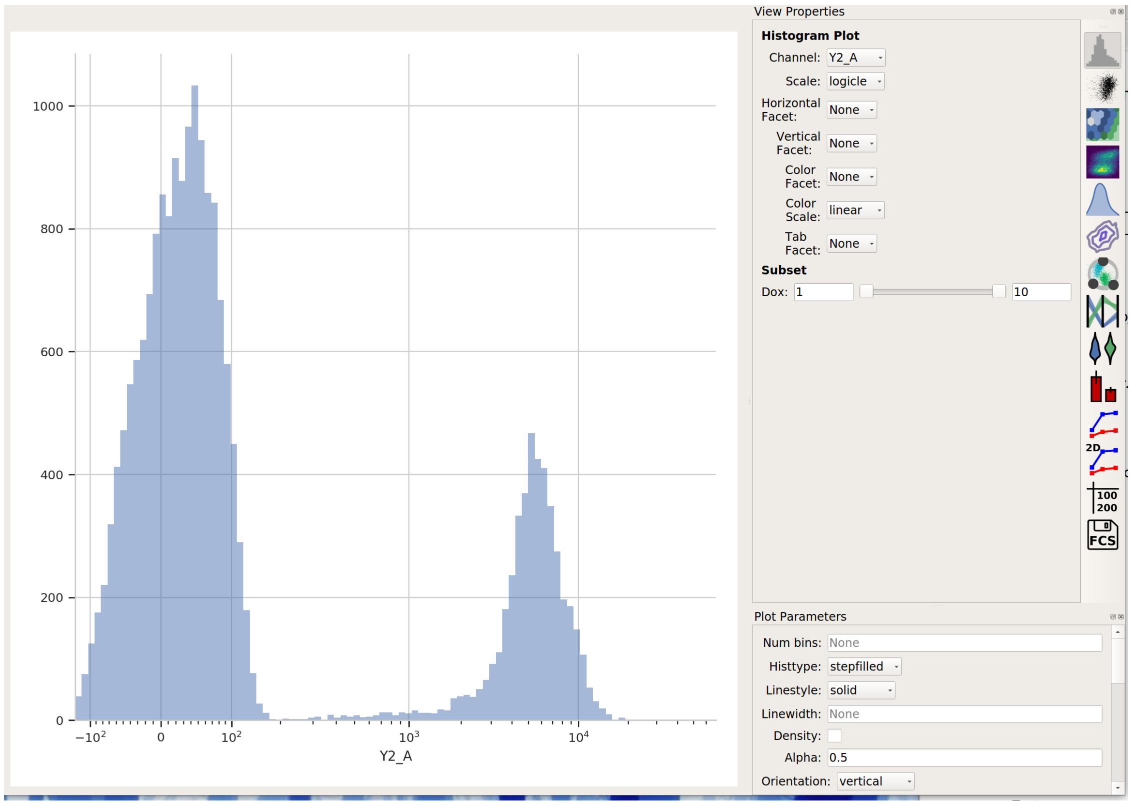
Samples: 2

Events: 20000

- Click the **Import!** button in the **Import Data** operation.

Examine the data

- Create a **Histogram** view. Set the channel to **Y2-A** and the scale to **logicle**.



View Properties

Histogram Plot

Channel:	Y2_A
Scale:	logicle
Horizontal Facet:	None
Vertical Facet:	None
Color Facet:	None
Color Scale:	linear
Tab Facet:	None

Subset

Dox: 1 10

Plot Parameters

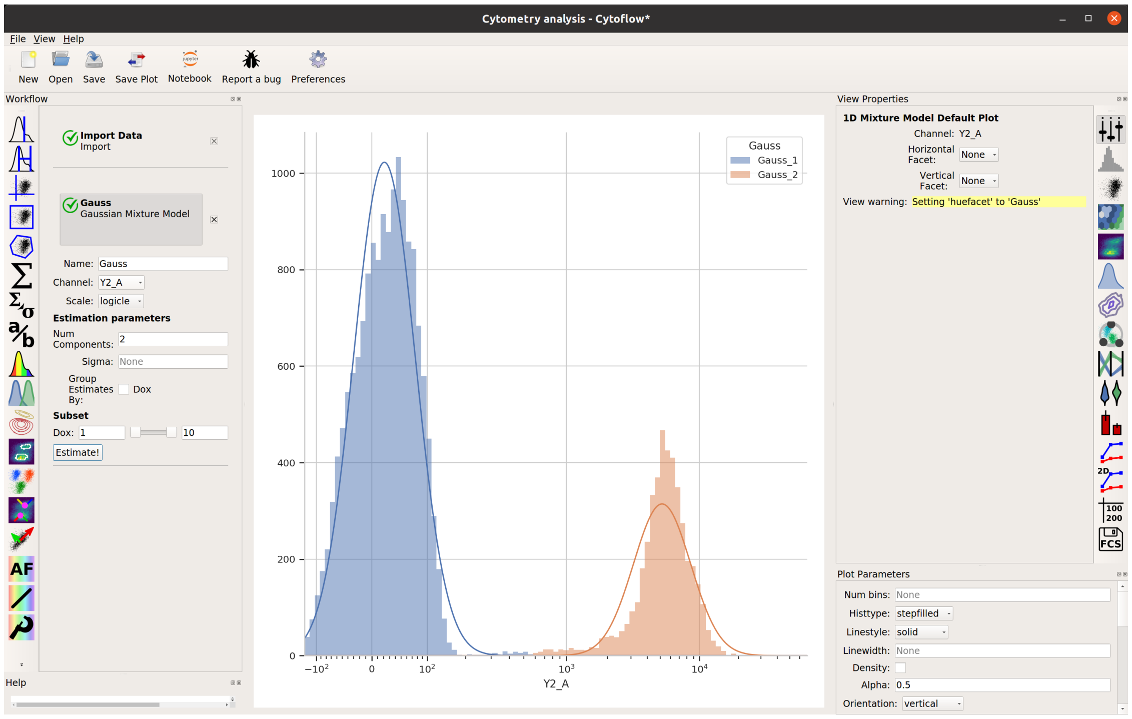
Num bins:	None
Histtype:	stepfilled
Linestyle:	solid
Linewidth:	None
Density:	☐
Alpha:	0.5
Orientation:	vertical

This data looks pretty bi-modal to me. If we wanted to separate out those two populations, a **Threshold** operation would do nicely. However, using a Gaussian mixture model comes with some advantages which we will see.

Create a Gaussian mixture model

- To model this data as a mixture of Gaussians, click the **1D Mixture Model** button:
- Set the parameters as follows:
  - **Name** – "Gauss" (or something memorable – it's arbitrary)
  - **Channel** – **Y2-A** (the channel we're applying the model to)
  - **Scale** – **logicle** (we want the *logicle* scale applied to the data before we model it)
  - **Num Components** – 2 (how many components are in the mixture?)

. . . and click **Estimate!** You'll see a plot like this:



Import Data
Import

Gauss
Gaussian Mixture Model

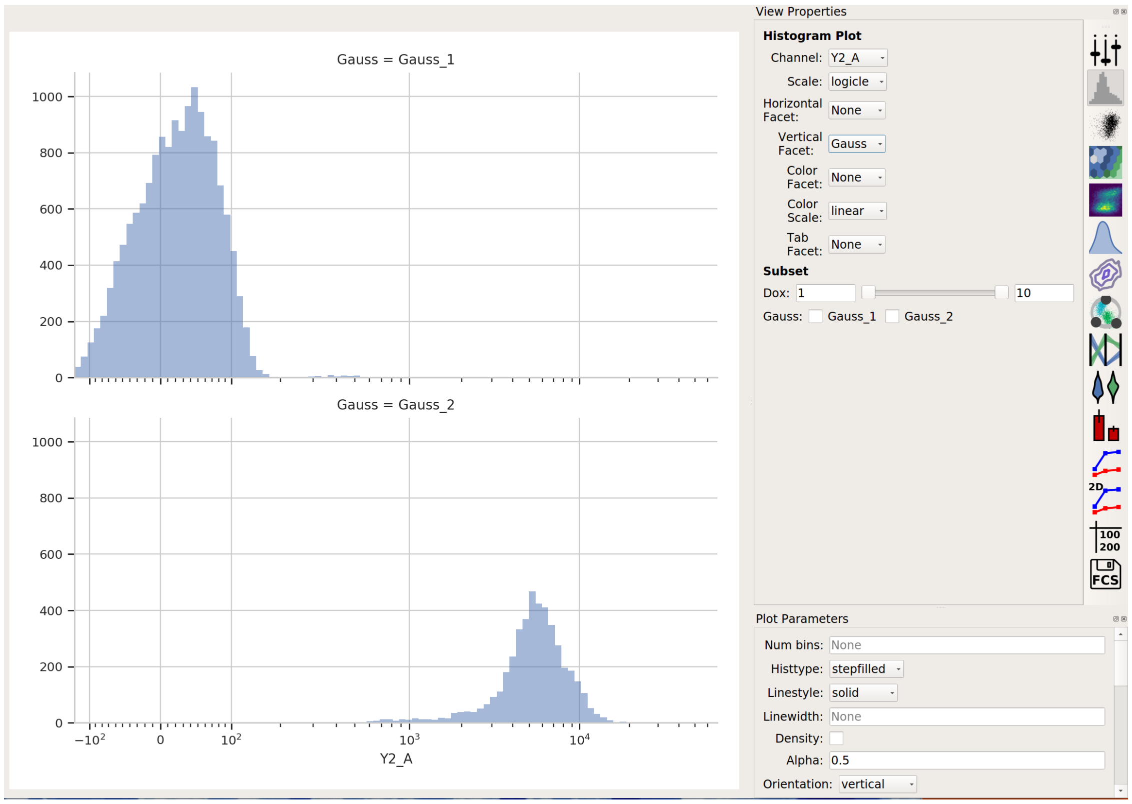
Name: Gauss
Channel: Y2_A
Scale: logicle
Estimation parameters
Num Components: 2
Sigma: None
Group Estimates By: Dox
Subset
Dox: 1 10
Estimate!

View Properties
1D Mixture Model Default Plot
Channel: Y2_A
Horizontal Facet: None
Vertical Facet: None
View warning: Setting 'huefacet' to 'Gauss'

Plot Parameters
Num bins: None
Histtype: stepfilled
Linestyle: solid
Linewidth: None
Density:
Alpha: 0.5
Orientation: vertical

Excellent. It looks like the mixture model found two populations and separated them out.

- What did the **Gaussian Mixture Model** operation actually do? Just like a threshold gate, it makes two new populations – in this case, they're named **Gauss_1** and **Gauss_2**. You can now view and manipulate them just like you could if you defined the two populations with any other gate. For example, I can view them on separate histograms:

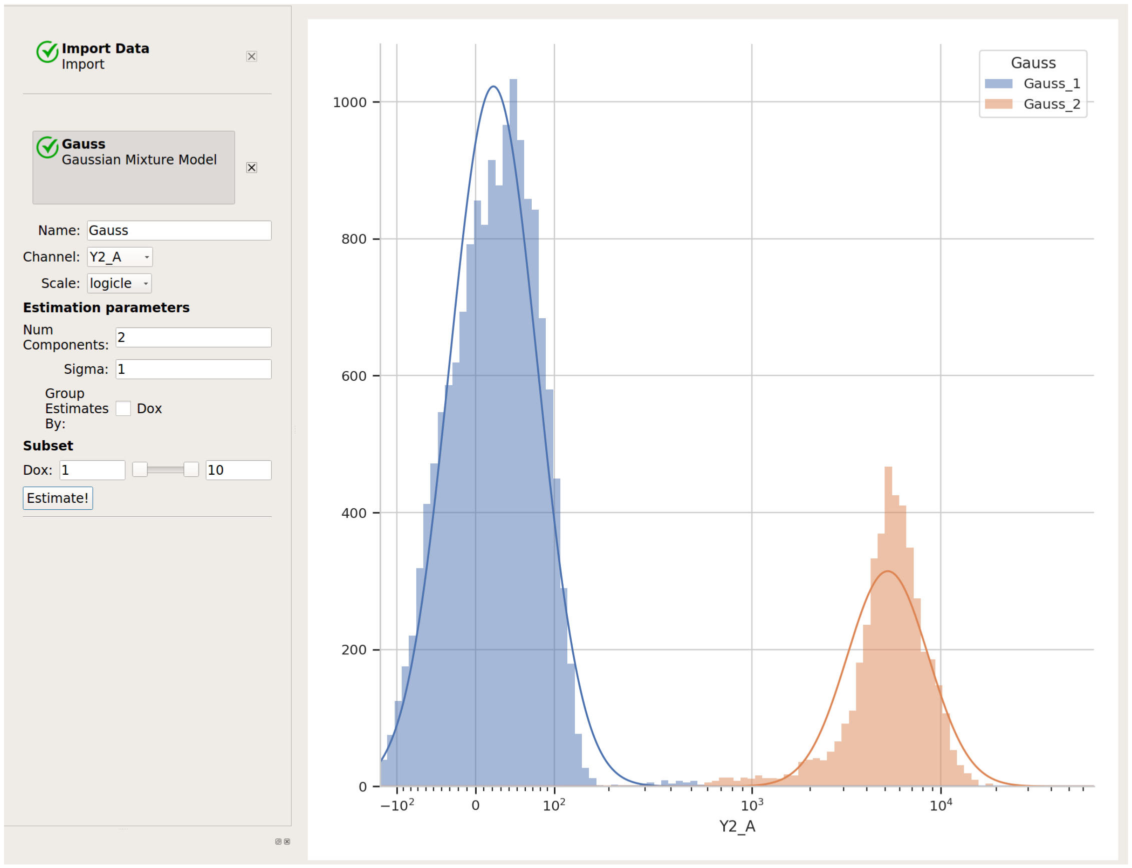


View Properties
Histogram Plot
Channel: Y2_A
Scale: logicle
Horizontal Facet: None
Vertical Facet: Gauss
Color Facet: None
Color Scale: linear
Tab Facet: None
Subset
Dox: 1 10
Gauss: Gauss_1 Gauss_2

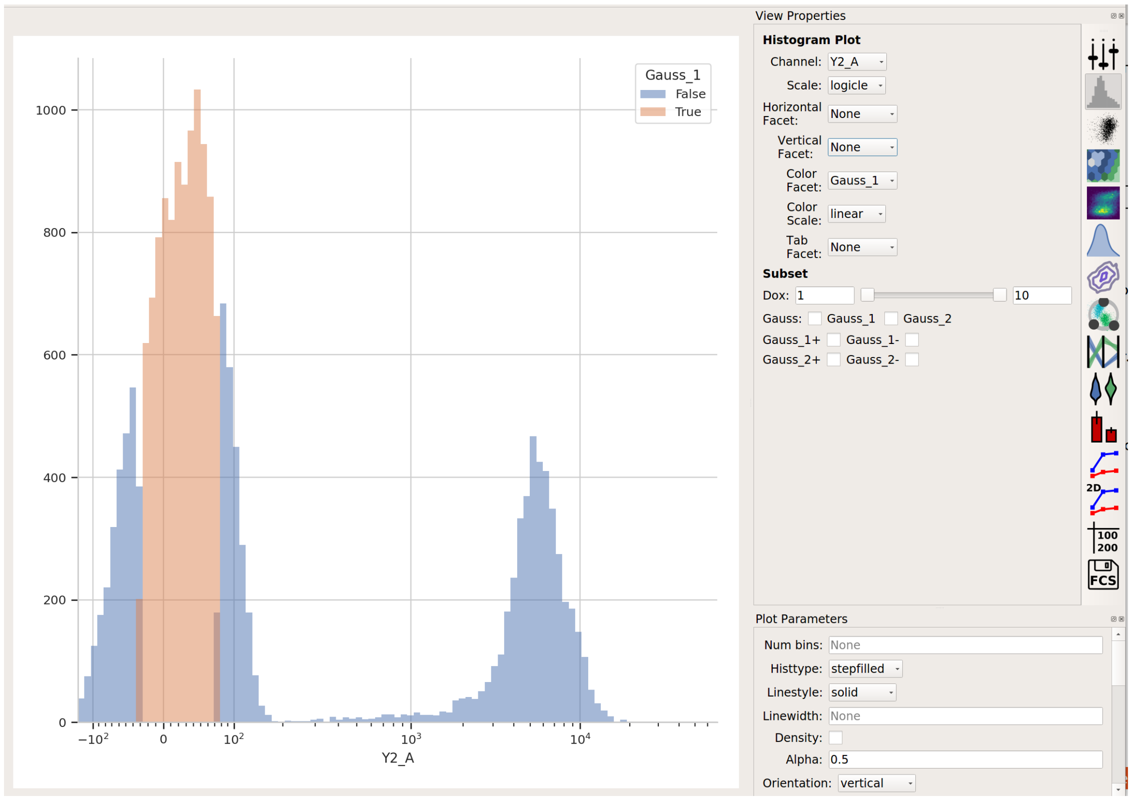
Plot Parameters
Num bins: None
Histtype: stepfilled
Linestyle: solid
Linewidth: None
Density:
Alpha: 0.5
Orientation: vertical

Advanced GMM uses

- Sometimes, data does not separate "cleanly" like this data set does. If that's the case, you can set the **sigma** estimation parameter. This tells the operation to create a new boolean variable for each component in the model, which is **True** if the event is within **sigma** standard deviations of the population mean. For example, if we set **sigma** to 1 and click **Estimate!**, our diagnostic plot remains the same:


Import Data
Import
Gauss
Gaussian Mixture Model
Name: Gauss
Channel: Y2_A
Scale: logicle
Estimation parameters
Num Components: 2
Sigma: 1
Group Estimates By: Dox
Subset
Dox: 1 10
Estimate!

. . . but now we've got two additional variables, **Gauss_1** and **Gauss_2**. For example, I can make a histogram and set the color facet to **Gauss_1**:



View Properties

Histogram Plot

Channel: Y2_A

Scale: logicle

Horizontal Facet: None

Vertical Facet: None

Color Facet: Gauss_1

Color Scale: linear

Tab Facet: None

Subset

Dox: 1 10

Gauss: ☐ Gauss_1 ☐ Gauss_2

Gauss_1+ ☐ Gauss_1- ☐

Gauss_2+ ☐ Gauss_2- ☐

Plot Parameters

Num bins: None

Histtype: stepfilled

Linestyle: solid

Linewidth: None

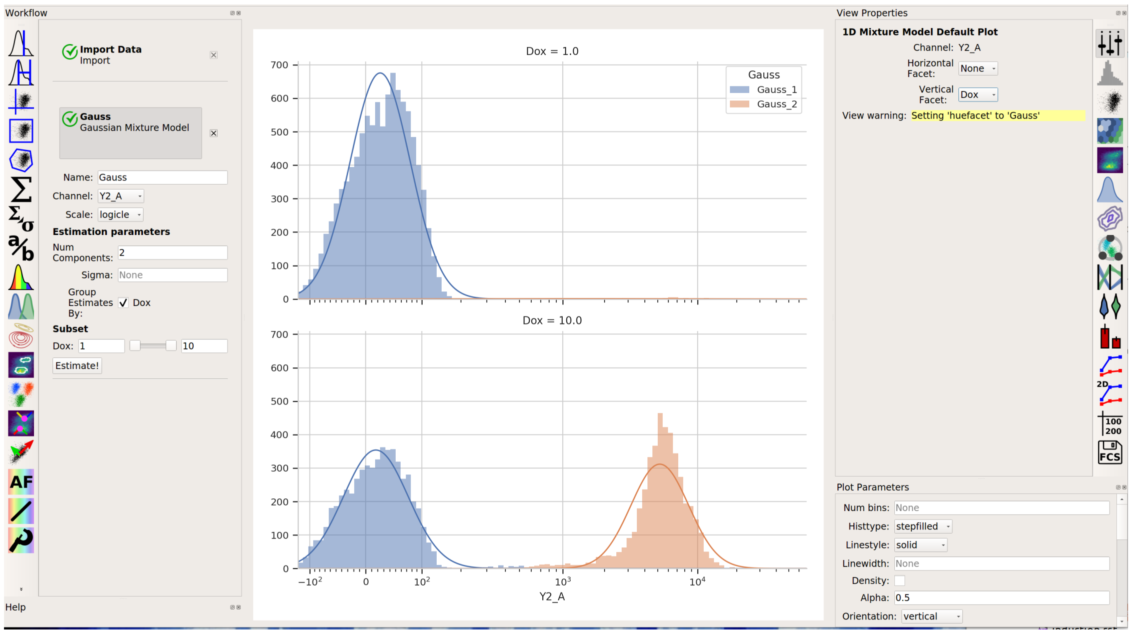
Density: ☐

Alpha: 0.5

Orientation: vertical

and we can see that only the "center" of the first population has the **Gauss_1** variable set to **True**.

- Sometimes, we don't want to use the same model parameters for the entire data set. Instead, sometimes you want to estimate different models for differe subsets. The **Gaussian Mixture Model** (and other data-driven operations) have a **Group Estimates By** parameter that allows you to just that. For example, let's say I want a different model estimated for each value of **Dox** (remember, that's my experimental variable.) I set **Group Estimates By** to **Dox**, and instead of getting one mixture model, now I get one for each different value of **Dox**:



Workflow

Import Data
Import

Gauss
Gaussian Mixture Model

Name: Gauss

Channel: Y2_A

Scale: logicle

Estimation parameters

Num Components: 2

Sigma: None

Group Estimates By: ✓ Dox

Subset

Dox: 1 10

Estimate!

Help

View Properties

1D Mixture Model Default Plot

Channel: Y2_A

Horizontal Facet: None

Vertical Facet: Dox

View warning: Setting 'huefacet' to 'Gauss'

Plot Parameters

Num bins: None

Histtype: stepfilled

Linestyle: solid

Linewidth: None

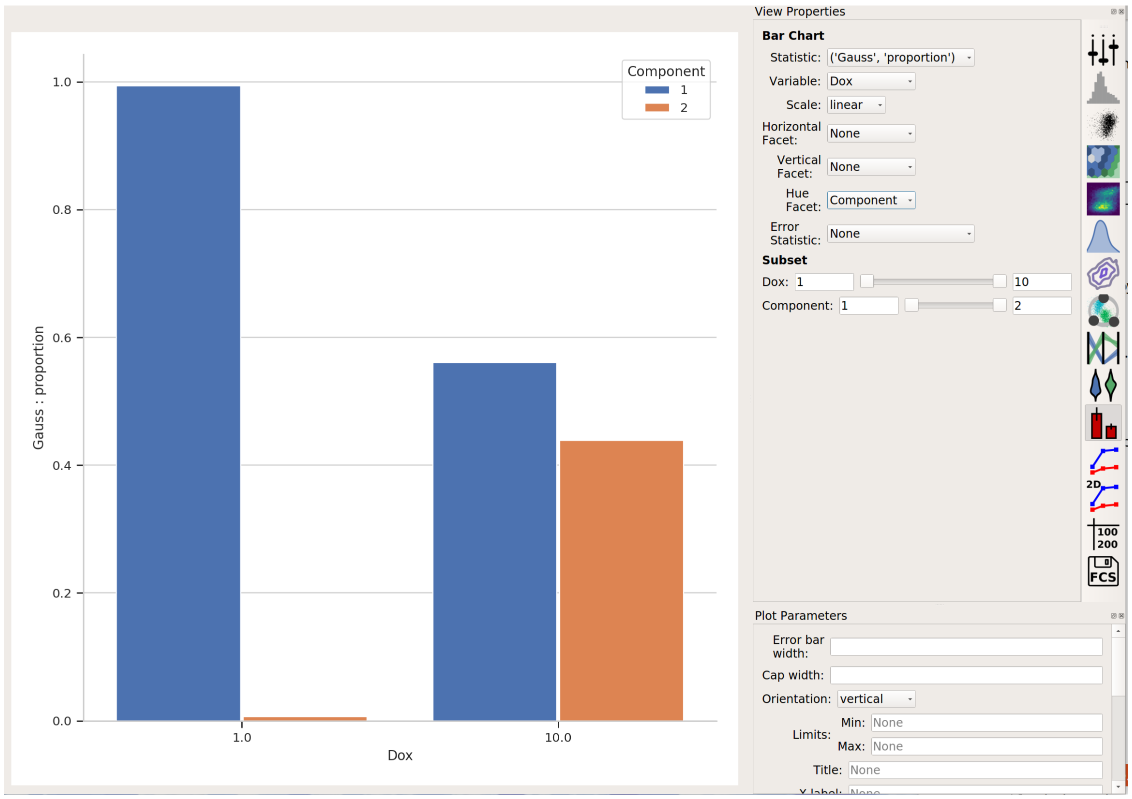
Density: ☐

Alpha: 0.5

Orientation: vertical

Note that there are two models shown, one for each value of **Dox**. (Note that I set **Vertical Facet** on the view to **Dox** to show both of them.

- Finally, in addition to gating events, data-driven modules also often create new statistics as well. For example, the **Gaussian Mixture Model** operation creates two statistics, **mean** and **proportion**, recording the mean of each population and the proportion of events that was in that population. This is particularly powerful when combined with the **Group Estimates By** parameter. For example, in the image above, it's pretty clear that there weren't many events in the **Gauss_2** population for **Dox = 1.0**. Let's look at the actual proportions using a bar graph:



Bar Chart
Statistic: ('Gauss', 'proportion')
Variable: Dox
Scale: linear
Horizontal Facet: None
Vertical Facet: None
Hue Facet: Component
Error Statistic: None
Subset
Dox: 1 10
Component: 1 2

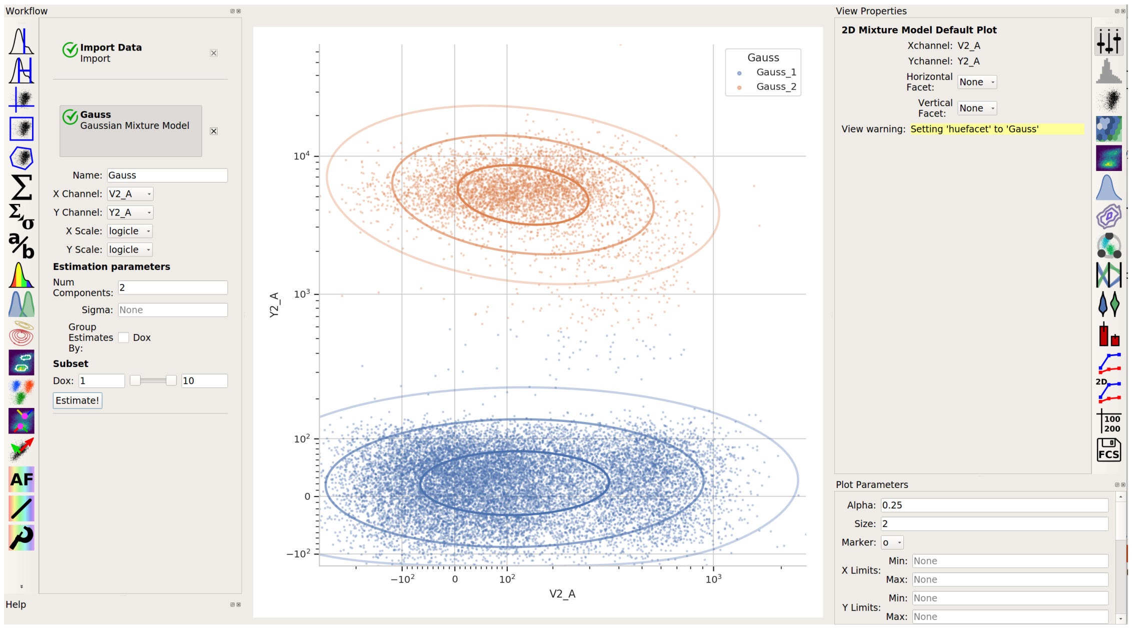
Error bar width:
Cap width:
Orientation: vertical
Limits: Min: None Max: None
Title: None

The *Tutorial: Yeast Multidimensional Induction Timecourse* tutorial gives a non-trival example of this.

More Machine Learning Operations

- **2-dimensional Gaussian Mixture model**

A gaussian mixture model that works in two dimensions (ie, on a scatterplot.)



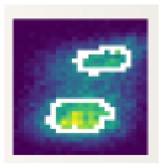
Import Data
Import

Gauss
Gaussian Mixture Model

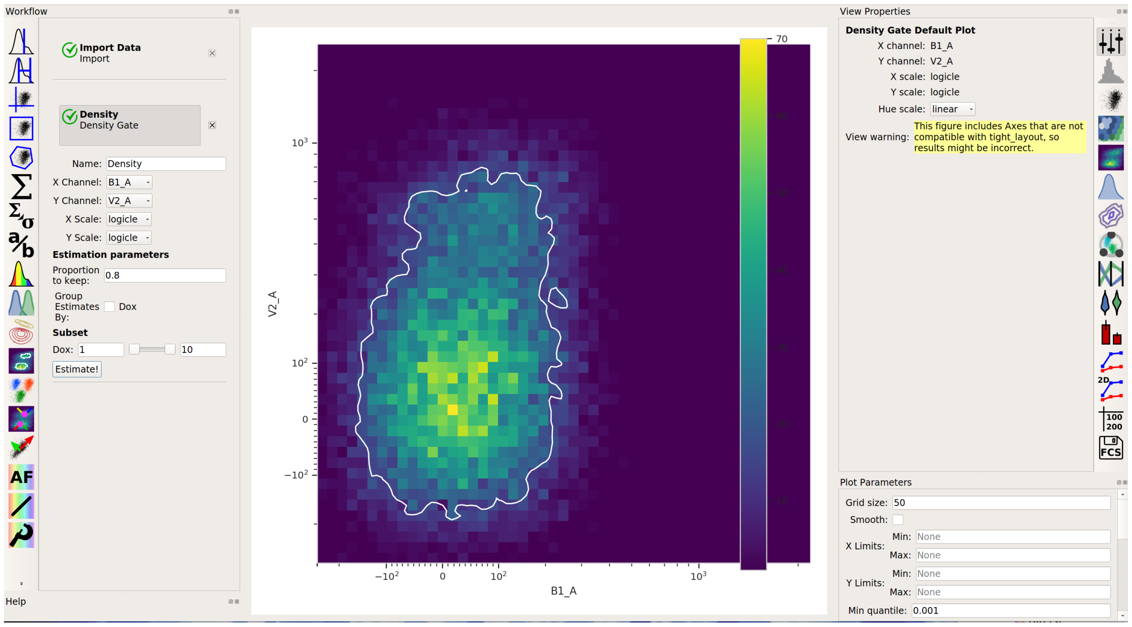
Name: Gauss
X Channel: V2_A
Y Channel: Y2_A
X Scale: logicle
Y Scale: logicle
Estimation parameters
Num Components: 2
Sigma: None
Group Estimates By: Dox
Subset
Dox: 1 10
Estimate!

View Properties
2D Mixture Model Default Plot
Xchannel: V2_A
Ychannel: Y2_A
Horizontal Facet: None
Vertical Facet: None
View warning: Setting 'huefacet' to 'Gauss'

Plot Parameters
Alpha: 0.25
Size: 2
Marker: o
X Limits: Min: None Max: None
Y Limits: Min: None Max: None

-  **Density gate**

Computes a gate based on a 2D density plot. The user chooses what proportion of events to keep, and the module creates a gate that selects the events in the highest-density bins of a 2D histogram.



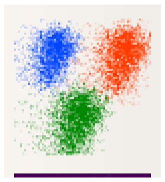
Import Data
Import

Density
Density Gate

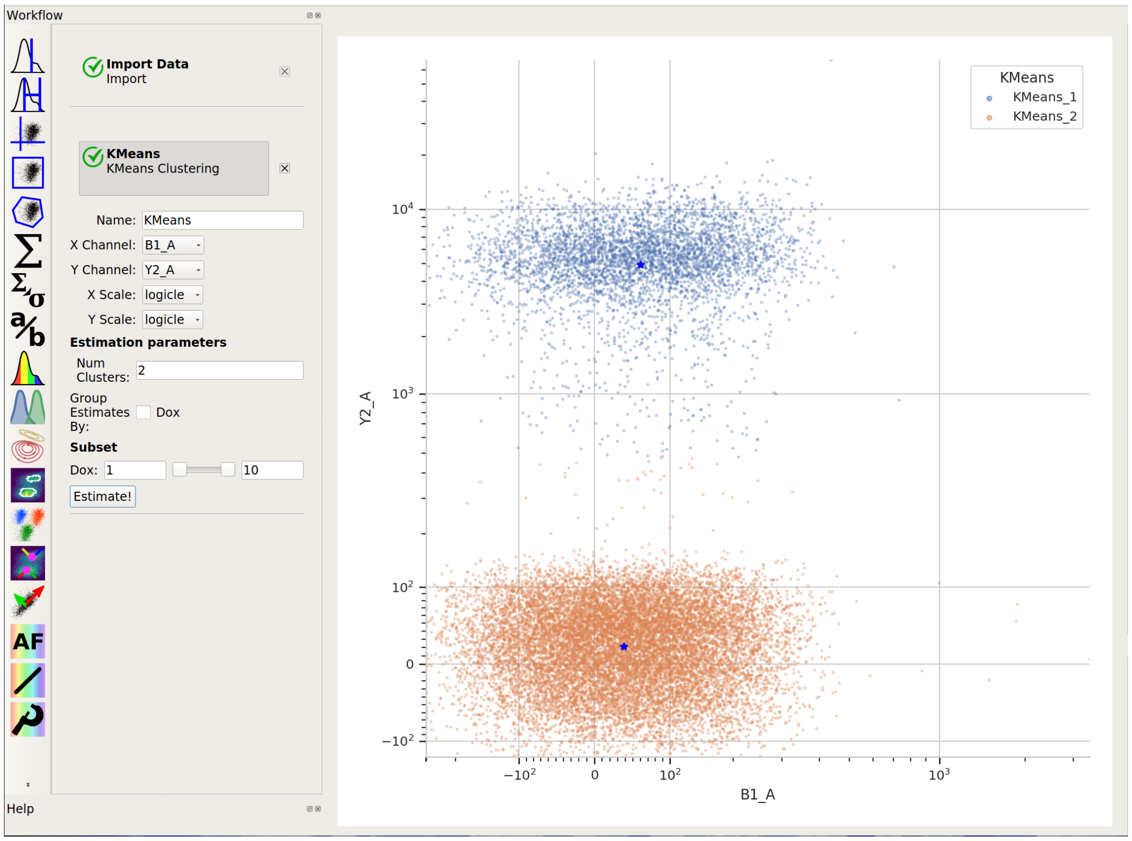
Name: Density
X Channel: B1_A
Y Channel: V2_A
X Scale: logicle
Y Scale: logicle
Estimation parameters
Proportion to keep: 0.8
Group Estimates By: Dox
Subset
Dox: 1 10
Estimate!

View Properties
Density Gate Default Plot
X channel: B1_A
Y channel: V2_A
X scale: logicle
Y scale: logicle
Hue scale: linear
View warning: This figure includes Axes that are not compatible with tight_layout, so results might be incorrect.

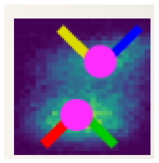
Plot Parameters
Grid size: 50
Smooth:
X Limits: Min: None Max: None
Y Limits: Min: None Max: None
Min quantile: 0.001

-  **K-means clustering**

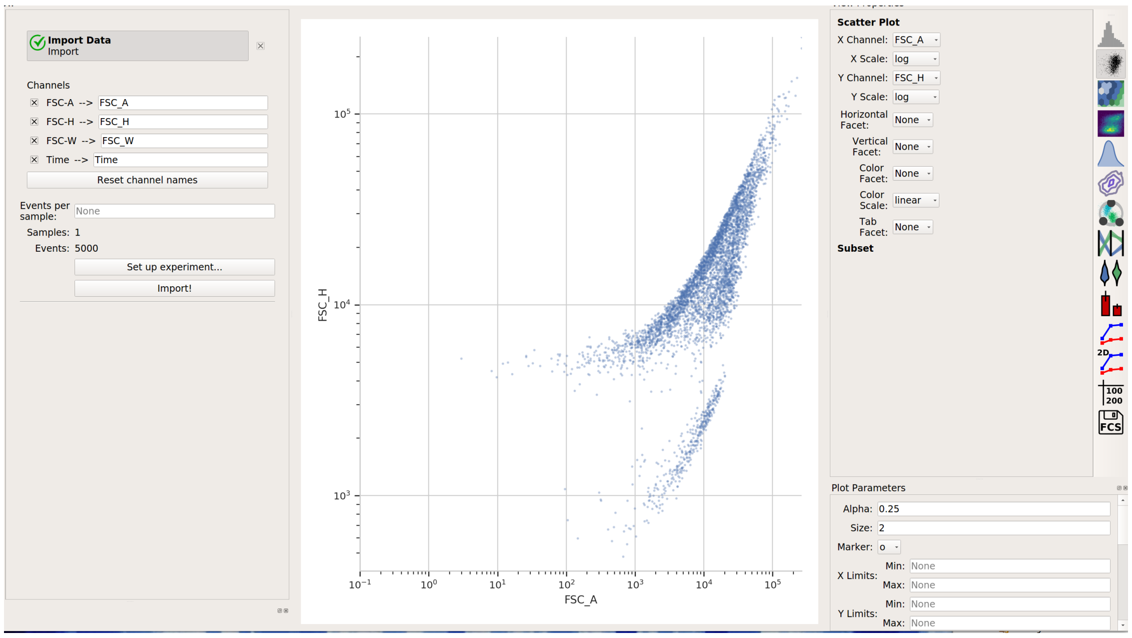
Uses k-means clustering on a scatter-plot. This sometimes works better than a 2D Gaussian mixture model if the populations aren't "normal" (ie, Gaussian- shaped).



Name: KMeans
X Channel: B1_A
Y Channel: Y2_A
X Scale: logicle
Y Scale: logicle
Estimation parameters
Num Clusters: 2
Group Estimates By: Dox
Subset
Dox: 1 10

-  **flowPeaks**

Sometimes, Gaussian mixtures and k-means clustering don't do a great job of clustering flow data. These clustering methods like data that is "compact" – regularly spaced around a "center". Many data sets are not like that. For example, this one, from the `ecoli.fcs` file in `examples-basic/data`:


Import Data
Import

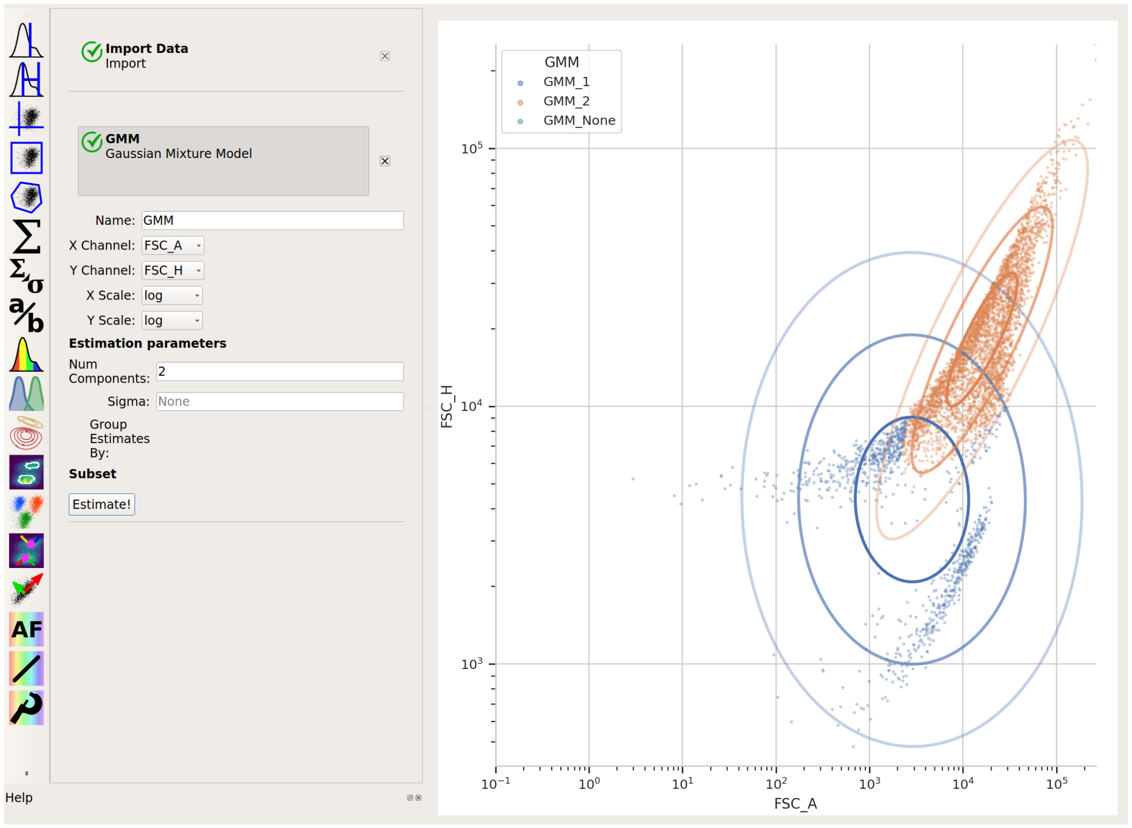
Channels
FSC-A --> FSC_A
FSC-H --> FSC_H
FSC-W --> FSC_W
Time --> Time

Events per sample: None
Samples: 1
Events: 5000

Scatter Plot
X Channel: FSC_A
X Scale: log
Y Channel: FSC_H
Y Scale: log
Horizontal Facet: None
Vertical Facet: None
Color Facet: None
Color Scale: linear
Tab Facet: None
Subset

Plot Parameters
Alpha: 0.25
Size: 2
Marker: o
X Limits: Min: None Max: None
Y Limits: Min: None Max: None

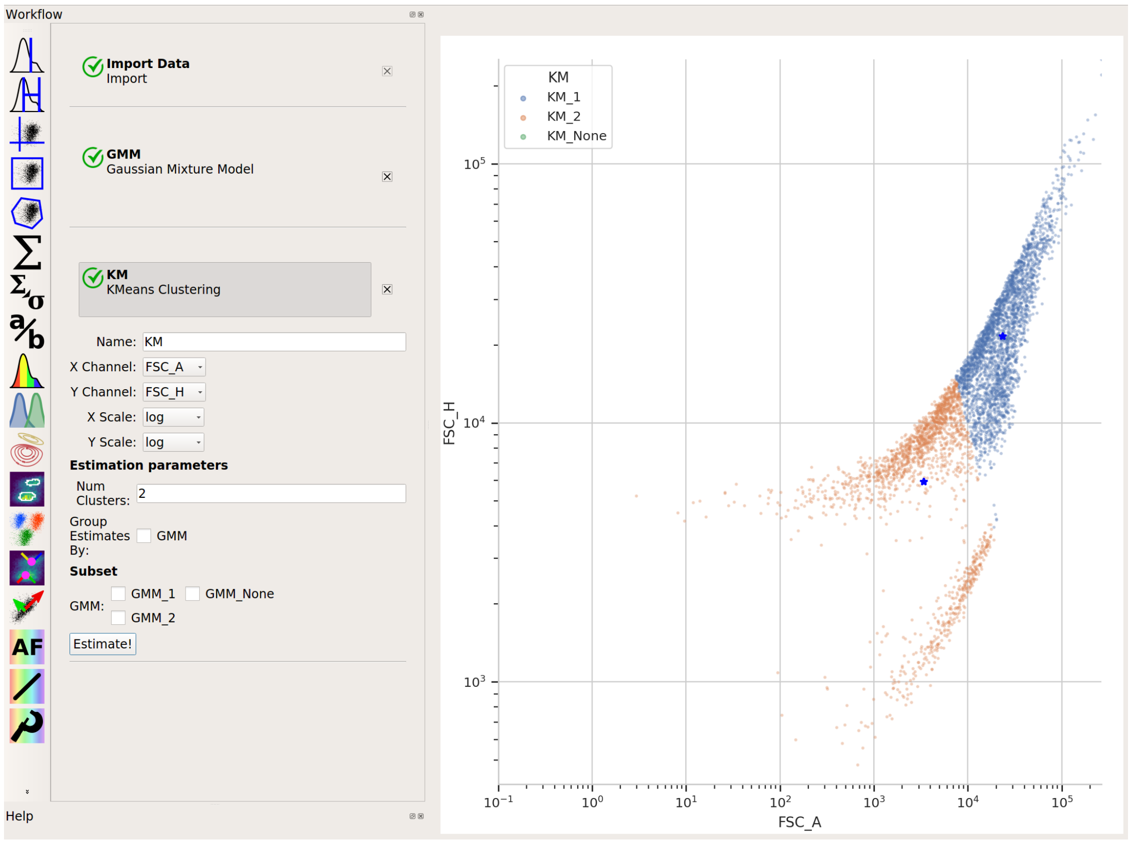
There are clearly two populations, but the Gaussian mixture method isn't effective at separating them:


Import Data
Import

GMM
Gaussian Mixture Model

Name: GMM
X Channel: FSC_A
Y Channel: FSC_H
X Scale: log
Y Scale: log
Estimation parameters
Num Components: 2
Sigma: None
Group Estimates By:
Subset

And neither is k-means:


Workflow

Import Data
Import

GMM
Gaussian Mixture Model

KM
KMeans Clustering

Name: KM
X Channel: FSC_A
Y Channel: FSC_H
X Scale: log
Y Scale: log

Estimation parameters

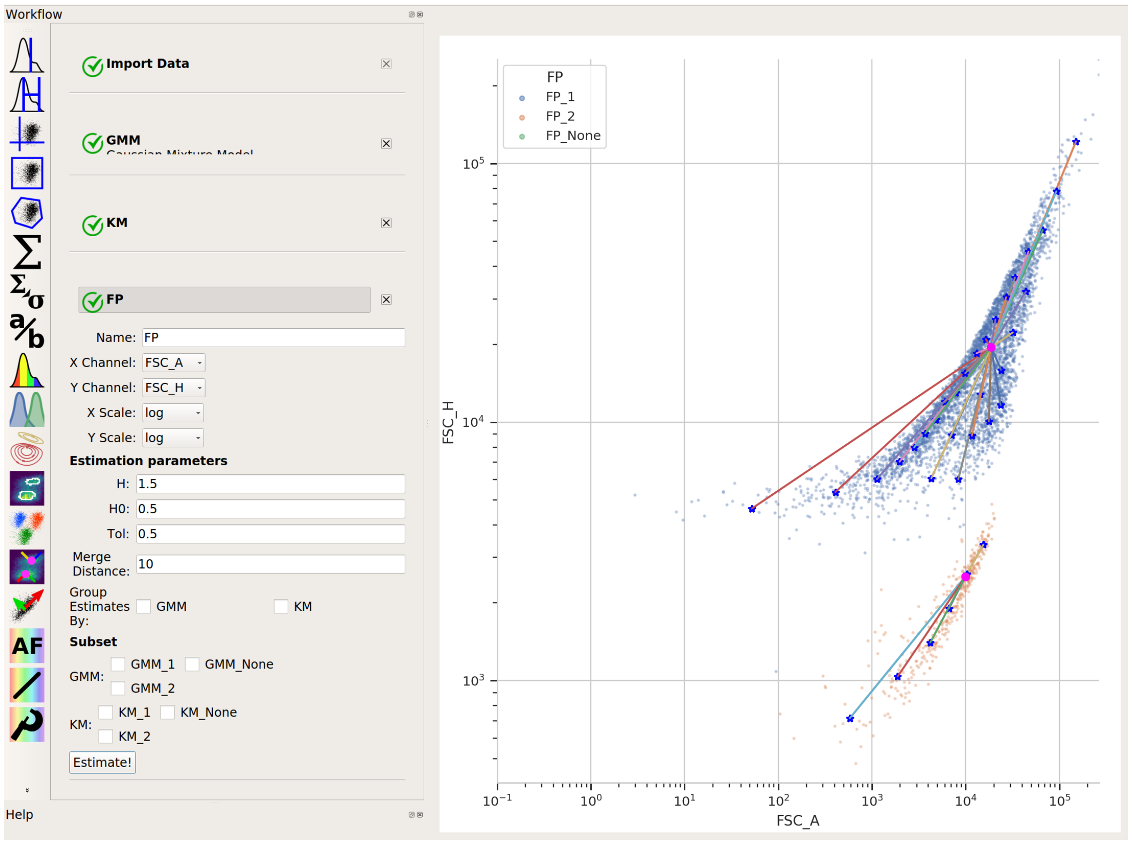
Num Clusters: 2

Group Estimates By: GMM

Subset

GMM: GMM_1 GMM_None GMM_2

In cases like this, a flow cytometry-specific method called `flowPeaks` may work better.



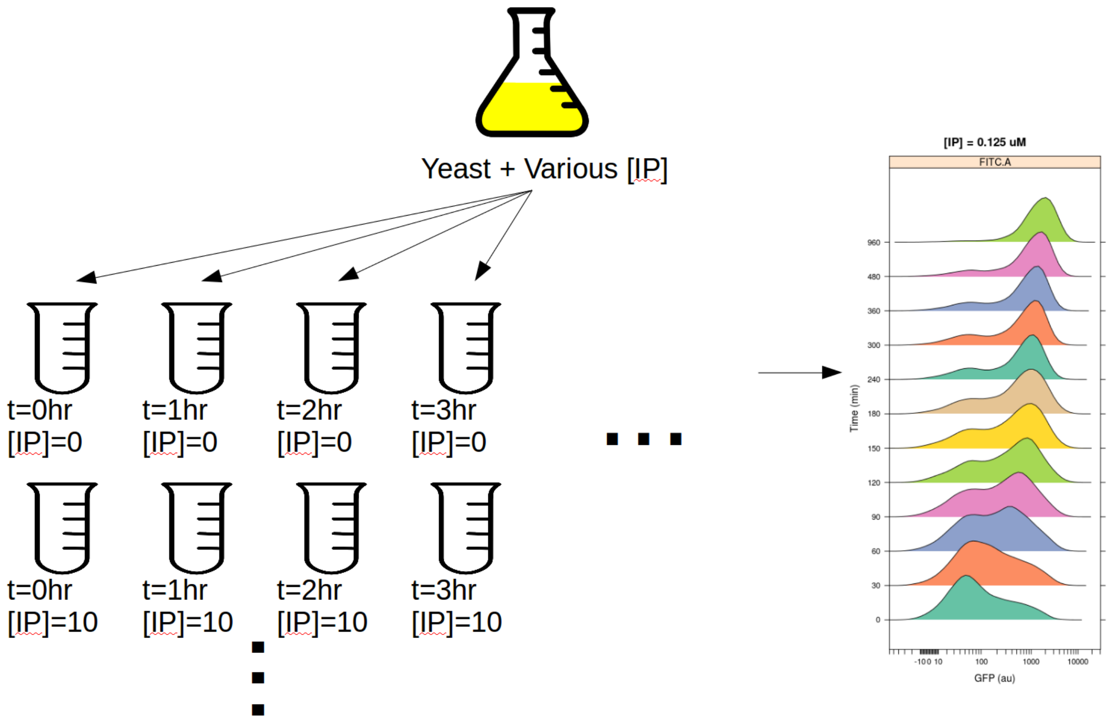
Name: FP
X Channel: FSC_A
Y Channel: FSC_H
X Scale: log
Y Scale: log
Estimation parameters
H: 1.5
H0: 0.5
Tol: 0.5
Merge Distance: 10
Group Estimates By: ☐ GMM ☐ KM
Subset
GMM: ☐ GMM_1 ☐ GMM_None ☐ GMM_2
KM: ☐ KM_1 ☐ KM_None ☐ KM_2
Estimate!

`flowPeaks` is nice in that it can automatically discover the "natural" number of clusters. There are two caveats, though. First, the peak-finding can be quite sensitive to the estimation parameters `h`, `h0`, `t` and `Merge distance`. The defaults are a good place to start, but if you're having trouble getting good performance, try tweaking them (see the documentation for what they mean.) And second, `flowPeaks` is quite computationally expensive. Thus, on large data sets, it can be quite slow.

5.1.4 Tutorial: Yeast Multidimensional Induction Timecourse

This workflow demonstrates a real-world multidimensional analysis. The yeast strain we're studying responds to the small molecule *isopentyladenine* (IP) by expressing a green fluorescent protein, which we can measure using a flow cytometer in the *FITC-A* channel.

This experiment was designed to determine the dynamics of the IP –> GFP response. I induced several yeast cultures with different amounts of IP, then took readings on the cytometer over the course of the day, every 30 minutes for 8 hours. The outline of the experimental setup is below.



If you'd like to follow along, you can do so by downloading one of the **cytoflow-#####-examples-advanced.zip** files from the Cytoflow releases page on GitHub. The files are in the **yeast/** subdirectory.

Warning: This is a pretty big data set; on modest computers, the operations can take quite some time to complete. Be patient!

Import the data

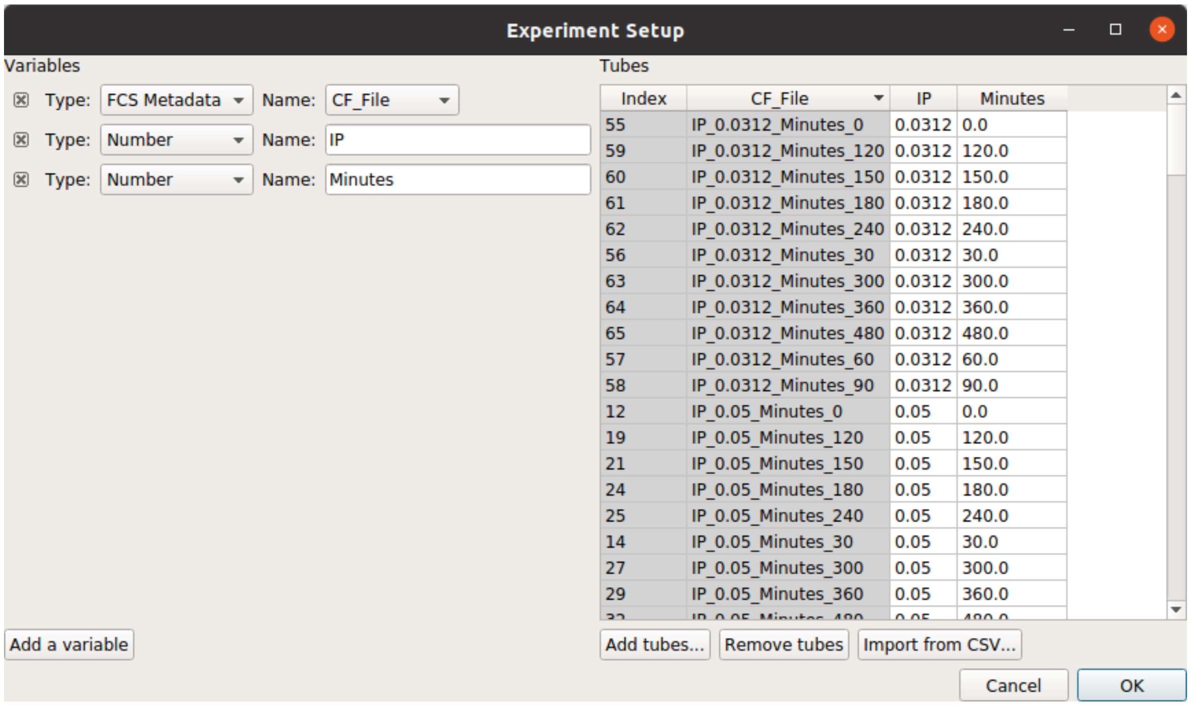
- Start Cytoflow. Under the **Import Data** operation, choose **Set up experiment…**
- Add two variables, *IP* and *Time*. Make both of them *Number* s.
- In this case, I've encoded the amount of IP and the time (in minutes) in the FCS files' file names. For example:

File	IP	Minutes
IP_0.0_Minutes_0	0	0
IP_0.05_Minutes_0	0.05	0
…	…	…
IP_0.0_Minutes_30	0	30
…	…	…

Note: There are a *lot* of rows in this table. Two things can make setting up these kinds of experiments easier. First, if you already have the details in a table, you can import that table by following the instructions at *HOWTO: Import an experiment from a table*. And second, you can select multiple cells in the table to edit at once by holding **Control** or **Command** and clicking multiple cells.

Warning: It is generally not a good idea to name a variable **Time**, because most flow cytometers produce FCS files with a **Time** "channel" and you can't re-use those names!

At the end, your table should look like this:

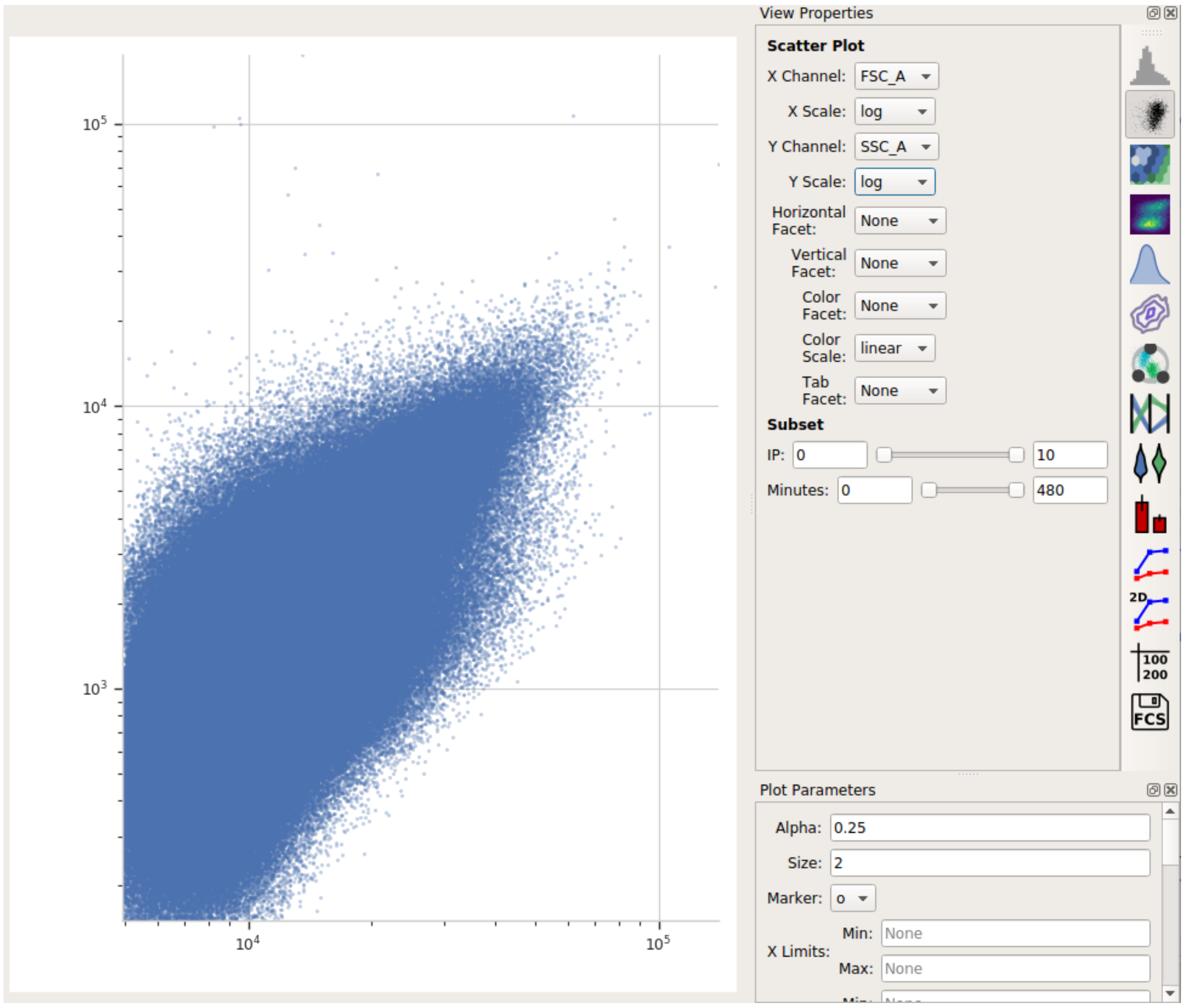


Type:	FCS Metadata	Name:	CF_File
Type:	Number	Name:	IP
Type:	Number	Name:	Minutes

Index	CF_File	IP	Minutes
55	IP_0.0312_Minutes_0	0.0312	0.0
59	IP_0.0312_Minutes_120	0.0312	120.0
60	IP_0.0312_Minutes_150	0.0312	150.0
61	IP_0.0312_Minutes_180	0.0312	180.0
62	IP_0.0312_Minutes_240	0.0312	240.0
56	IP_0.0312_Minutes_30	0.0312	30.0
63	IP_0.0312_Minutes_300	0.0312	300.0
64	IP_0.0312_Minutes_360	0.0312	360.0
65	IP_0.0312_Minutes_480	0.0312	480.0
57	IP_0.0312_Minutes_60	0.0312	60.0
58	IP_0.0312_Minutes_90	0.0312	90.0
12	IP_0.05_Minutes_0	0.05	0.0
19	IP_0.05_Minutes_120	0.05	120.0
21	IP_0.05_Minutes_150	0.05	150.0
24	IP_0.05_Minutes_180	0.05	180.0
25	IP_0.05_Minutes_240	0.05	240.0
14	IP_0.05_Minutes_30	0.05	30.0
27	IP_0.05_Minutes_300	0.05	300.0
29	IP_0.05_Minutes_360	0.05	360.0

Filter out clumps and debris

Let's have a look at the morphological parameters, *FSC-A* and *SSC-A*. There are so many events in this data set that a standard scatterplot isn't a great choice:



View Properties

Scatter Plot

X Channel: FSC_A
X Scale: log
Y Channel: SSC_A
Y Scale: log
Horizontal Facet: None
Vertical Facet: None
Color Facet: None
Color Scale: linear
Tab Facet: None

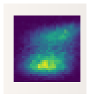
Subset

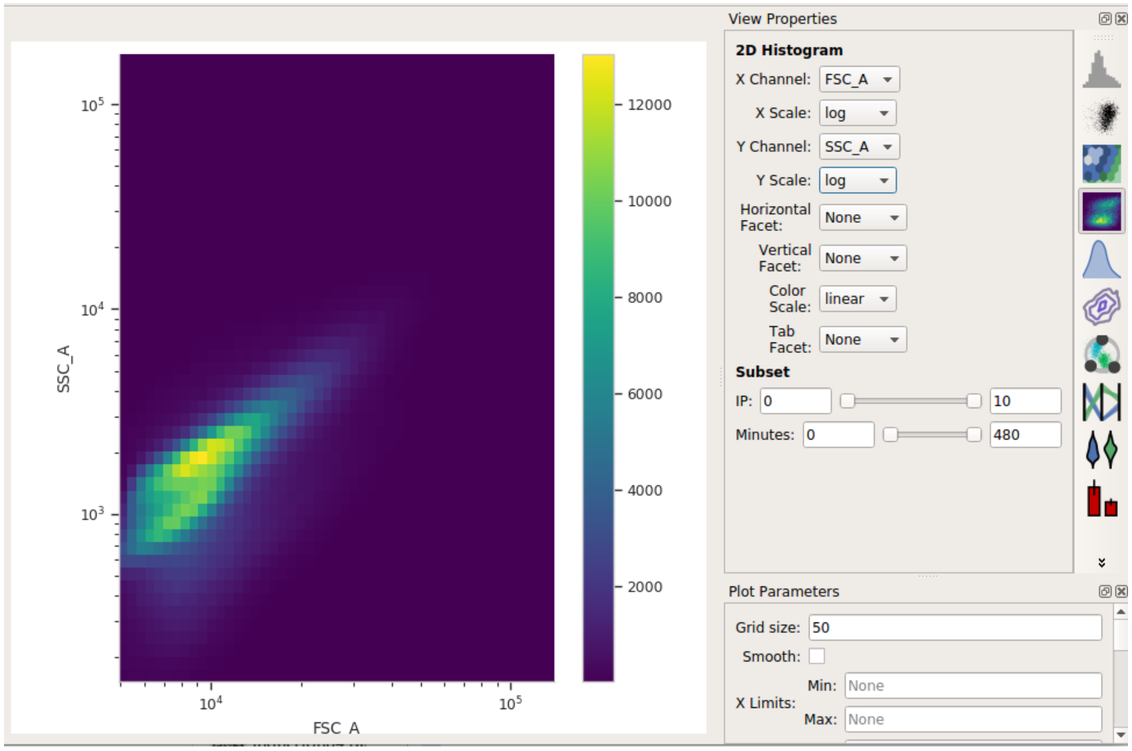
IP: 0 — 10
Minutes: 0 — 480

Plot Parameters

Alpha: 0.25
Size: 2
Marker: o
X Limits: Min: None Max: None

Instead, let's use a density plot.

- Click the density plot button: 
- Set the **X channel** to **FSC_A** and the **Y channel** to **SSC_A**. Change both axis scales to **log**.



View Properties

2D Histogram

X Channel: FSC_A
X Scale: log
Y Channel: SSC_A
Y Scale: log
Horizontal Facet: None
Vertical Facet: None
Color Scale: linear
Tab Facet: None

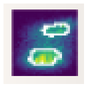
Subset

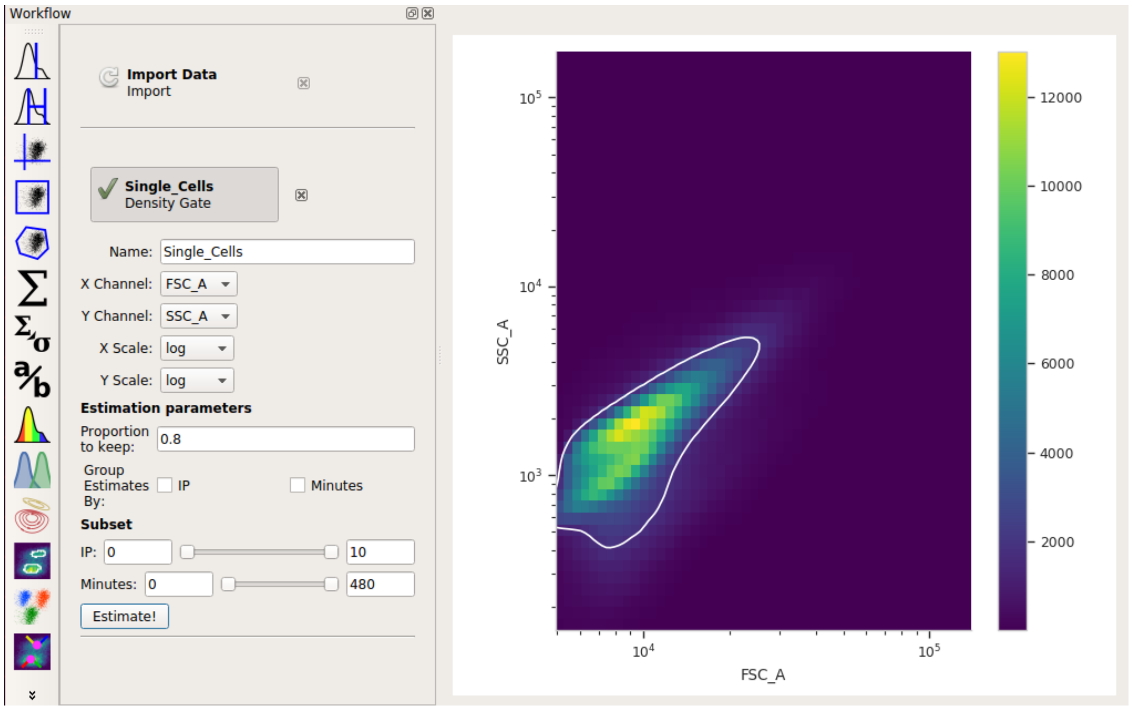
IP: 0 … 10
Minutes: 0 … 480

Plot Parameters

Grid size: 50
Smooth:
X Limits: Min: None Max: None

This distribution looks quite tight. Instead of drawing a polygon, let's use a density gate to keep the 80% of events that are in the "center" of this distribution.

- Add a **Density Gate** operation to your workflow: 
- Set **X channel** to **FSC_A** and **Y Channel** to **SSC_A**. Change both axis scales to **log**. By default, the operation keeps 90% of the events; let's change that to 80% by setting **Proportion to keep** to *0.8*.
- Click **Estimate!** and check the diagnostic plot to make sure that the gate captures the population you want:



Name: Single_Cells
X Channel: FSC_A
Y Channel: SSC_A
X Scale: log
Y Scale: log
Estimation parameters
Proportion to keep: 0.8
Group Estimates By: IP Minutes
Subset
IP: 0 10
Minutes: 0 480
Estimate!

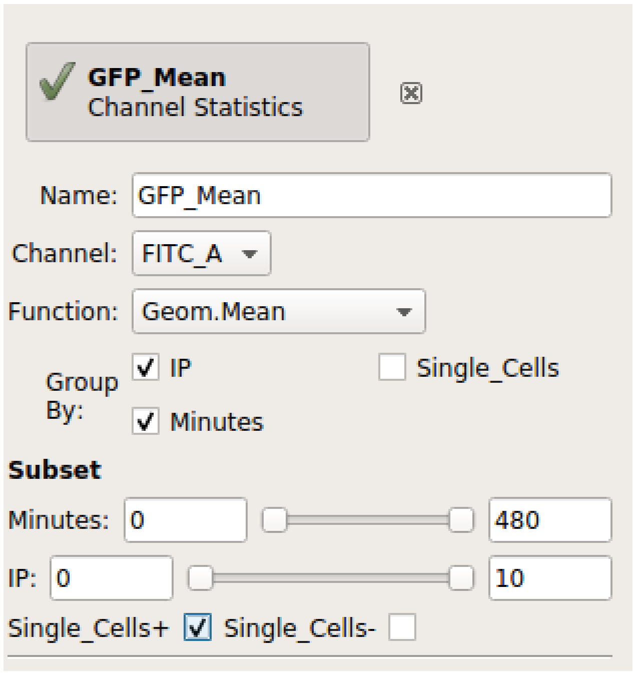
Compute the geometric mean at each different timepoint and IP concentration

Next, we'll create a new **Channel Statistic** to compute the geometric mean of the **FITC_A** channel at each timepoint and IP concentration.

Note: Why the geometric mean? See *Guide: Which measure of center should I use?*

- Add a **Channel Statistics** operation: Σ
- Give the new statistic a name – I called it *GFP_Mean* – and choose the channel we want to analyze (*FITC_A*) and the function we want to apply (*Geom.Mean*)
- Now we need to tell `Cytoflow` which subsets of our data we want to apply the function to. We want the geometric mean computed for every different value of IP *and* timepoint; so set **Group by** to **IP** *and* **Minutes**.
- Again, we only want to analyze the cells in the *Single_Cells* gate – so set **Subset** to *Single_Cells+*.

At the end, your operation should look like this:


GFP_Mean
Channel Statistics

Name: GFP_Mean
Channel: FITC_A
Function: Geom.Mean
Group By: [✓ IP] Single_Cells [✓ Minutes]

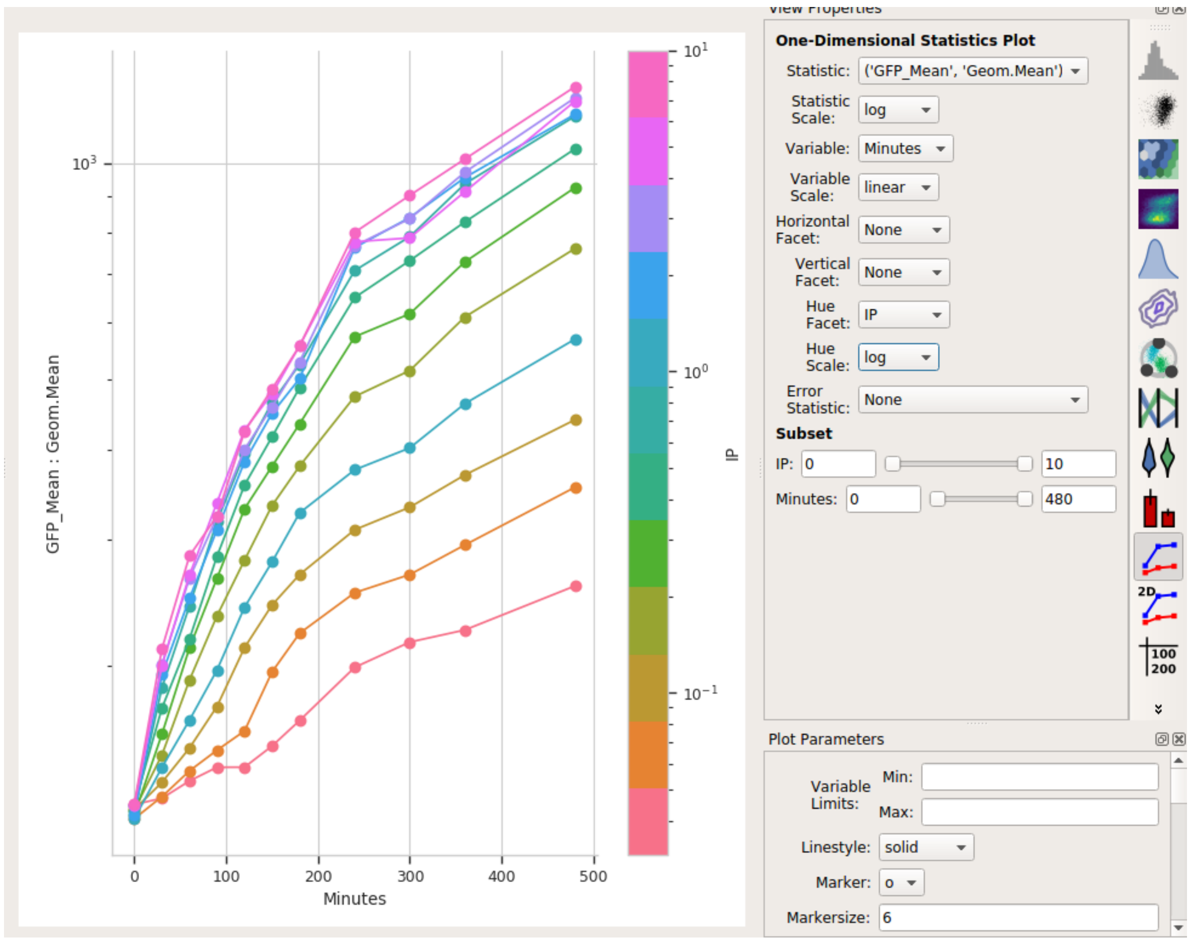
Subset

Minutes: 0 – 480
IP: 0 – 10
[✓ Single_Cells+] Single_Cells-

Now that we've made a new summary statistic, we want to plot it!

- Open the **1D Statistics View**:
- Set **Statistic** to the name of the statistic we just created: *('GFP_Mean', 'Geom.Mean')* (note that it shows us both the name of the operation that created the statistic, and the function that we used.)
- Set the **Statistic Scale** to **log**. This is how the plot will scale the Y axis.
- Set **Variable** to the variable we want on the X axis – in this case, *Minutes*.
- Set **Hue Facet** to the variable we want plotted in different colors – in this case, *IP*.
- The IP concentrations were a standard dilution series, so change the **Hue scale** to **log**.

Et voila, a scatter plot:



One-Dimensional Statistics Plot

Statistic: ('GFP_Mean', 'Geom.Mean')

Statistic Scale: log

Variable: Minutes

Variable Scale: linear

Horizontal Facet: None

Vertical Facet: None

Hue Facet: IP

Hue Scale: log

Error Statistic: None

Subset

IP: 0 10

Minutes: 0 480

Variable Limits: Min: Max:

Linestyle: solid

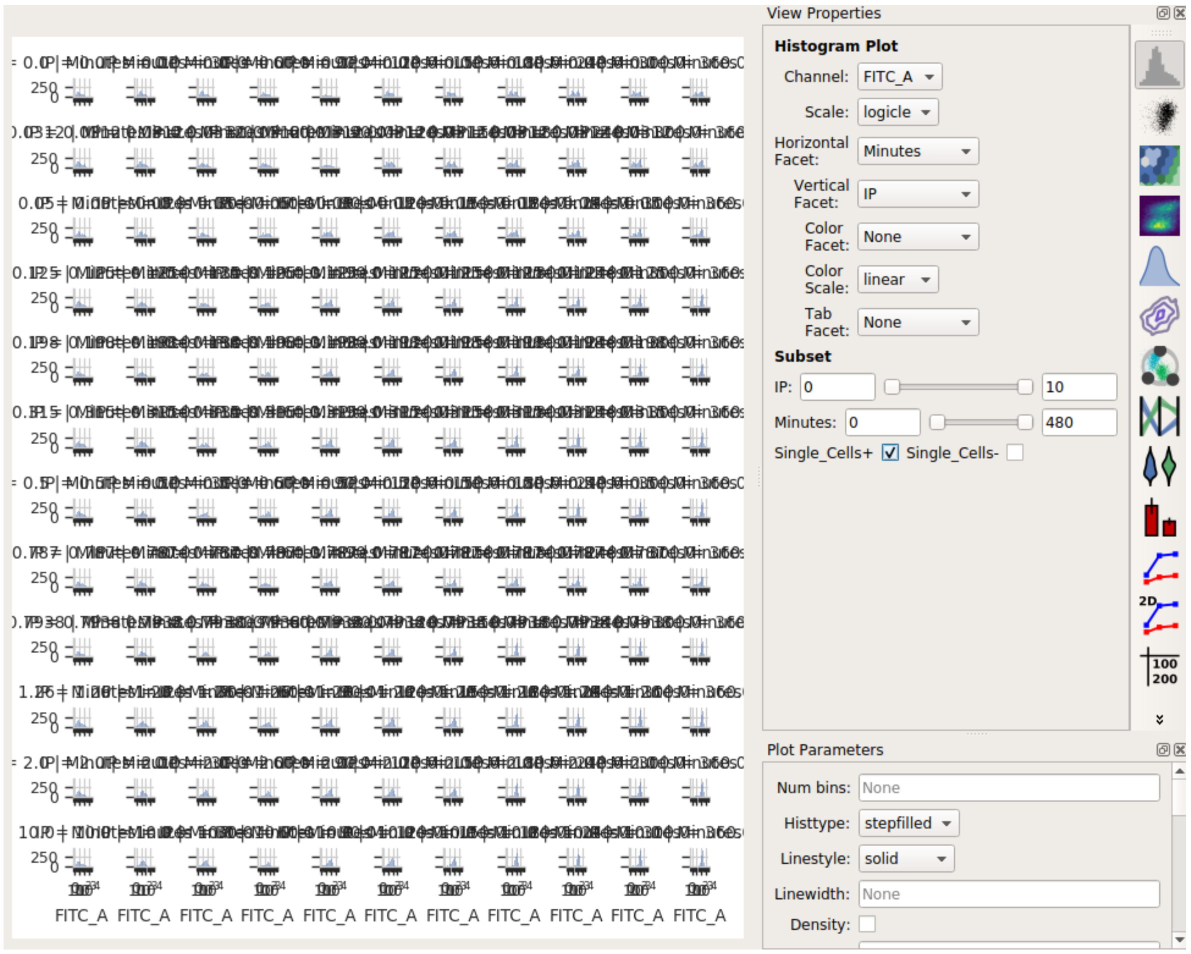
Marker: o

Markersize: 6

Is a geometric mean an appropriate summary statistic?

A geometric mean is only an appropriate summary statistic if the unimodal in log space. Is this actually true? Let's look at the histogram of each IP/time combination to find out.

- Choose the histogram view:
- Set the **Channel** to **FITC_A**, the **Scale** to *logicle*, the **Horizontal facet** to *Minutes* and the **Vertical facet** to *IP*.
- Set **Subset** to *Single_Cells+*



Histogram Plot

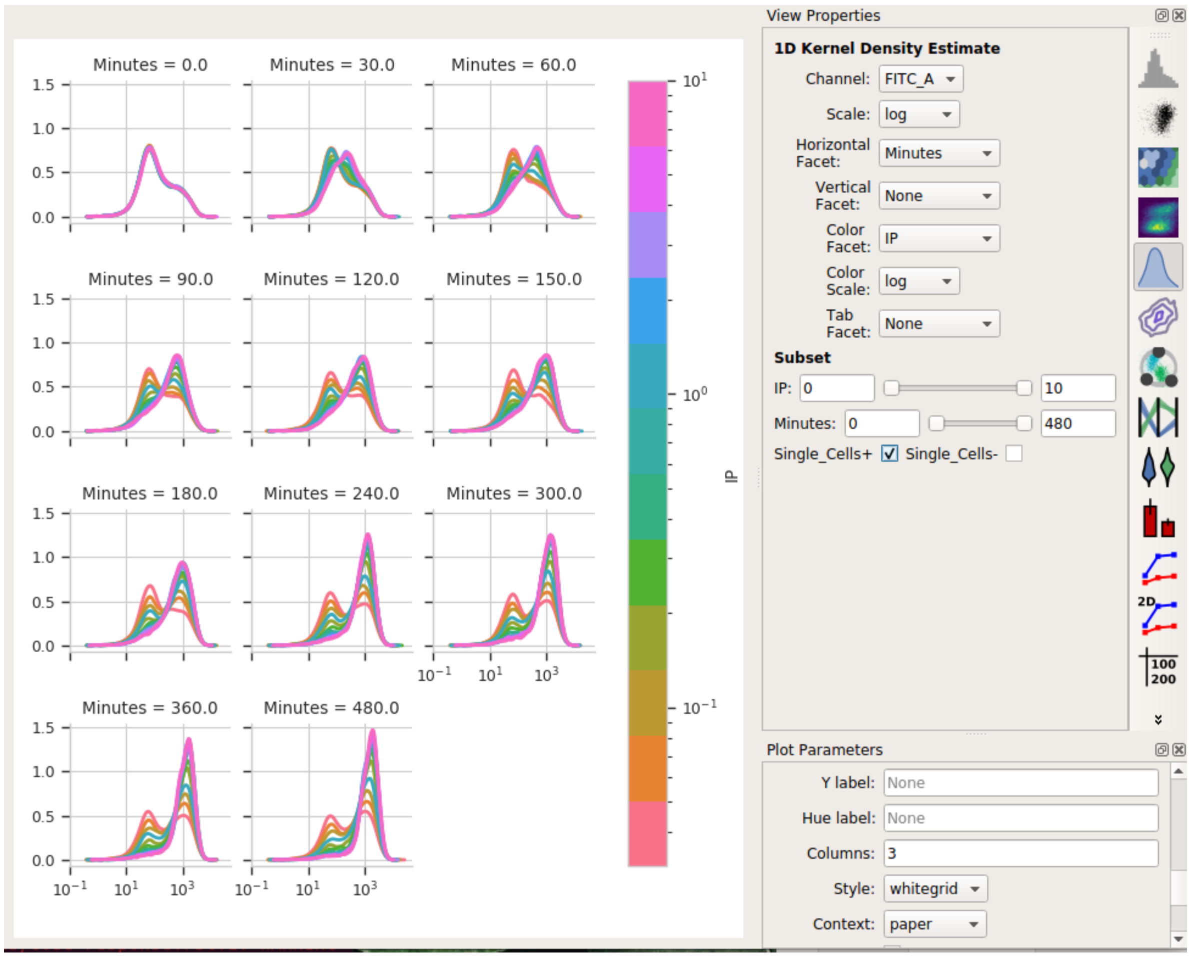
Channel: FITC_A
Scale: logicle
Horizontal Facet: Minutes
Vertical Facet: IP
Color Facet: None
Color Scale: linear
Tab Facet: None

Subset

IP: 0 10
Minutes: 0 480
Single_Cells+ [✓] Single_Cells- [ ]

Num bins: None
Histtype: stepfilled
Linestyle: solid
Linewidth: None
Density: [ ]

Eeep, that's impossible to read! Instead, let's put the *IP* variable on the *Hue* axis, and then use the **Columns** parameter to give us a table of plots. We'll also change to a **1D Kernel Density Estimate**, which will give us smoothed lines instead of jagged histograms.



1D Kernel Density Estimate

Channel: FITC_A

Scale: log

Horizontal Facet: Minutes

Vertical Facet: None

Color Facet: IP

Color Scale: log

Tab Facet: None

Subset

IP: 0 10

Minutes: 0 480

Single_Cells+ ☑ Single_Cells- ☐

Y label: None

Hue label: None

Columns: 3

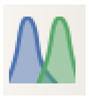
Style: whitegrid

Context: paper

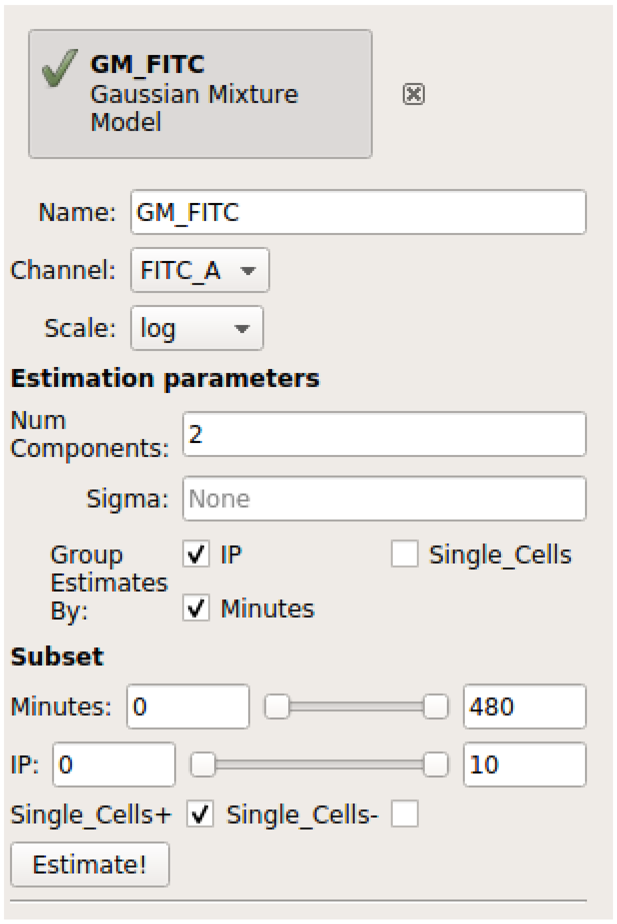
Okay, now *this* is interesting. Many of these distributions are *not* unimodal. Instead, there's significant additional structure. It's almost like there are two populations of cells in each tube – on that's "off" and one that's "on" – and different amounts of IP and time change the proportion of cells in each population.

Model the data as a mixture of gaussians

It turns out that this "mixture of Gaussians" thing is sufficiently common in cytometery that `Cytoflow` has a module that can handle it explicitly. Let's have `Cytoflow` model each IP/time subset as a mixture of two gaussians and see if that's more informative than the simple dose-response curve.

- Add a **1D Mixture Model** to your workflow: 
- Set the name to something reasonable – I chose *GM_FITC* – and the channel to *FITC_A* and the scale to *log*.
- We want a model with two components, so set **Num components** to 2.
- We want a *separate* model fit to each subset of data with unique values of *IP* and *Minutes*. So, set **Group estimates by** to **IP** *and* **Minutes**.
- We only want to estimate the model from the cells in the *Single_Cells* gate – so set **Subset** to **Single_Cells+**.

  Your operation should look like this:



GM_FITC
Gaussian Mixture Model

Name: GM_FITC

Channel: FITC_A

Scale: log

Estimation parameters

Num Components: 2

Sigma: None

Group Estimates By: [✓ IP] [✓ Minutes] Single_Cells

Subset

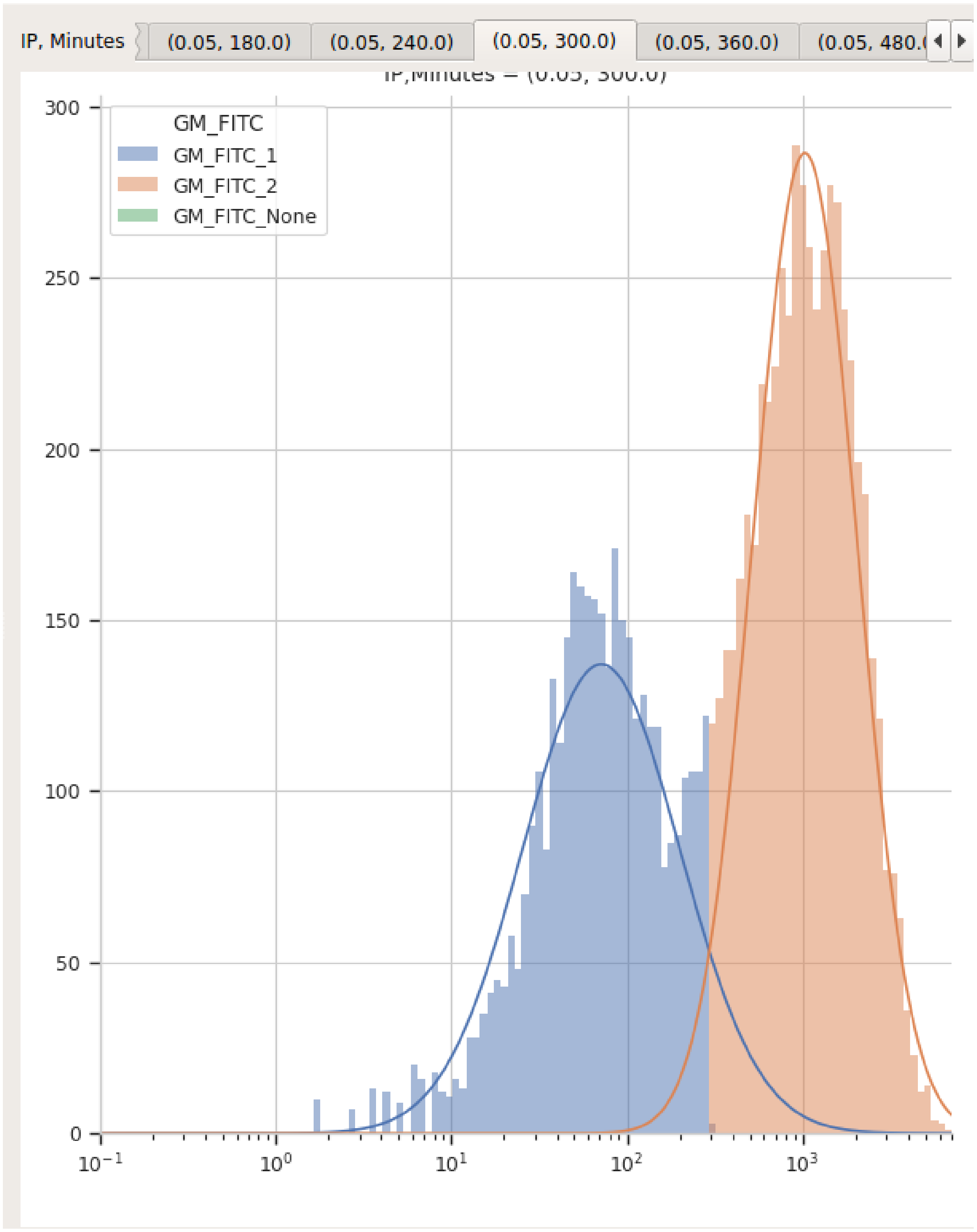
Minutes: 0 – 480

IP: 0 – 10

[✓ Single_Cells+] Single_Cells-

- Click **Estimate!**

You can page through the tabs on the plot to look at the various models that were fit. For example, here's the IP=0.05, Minutes=300 tab:

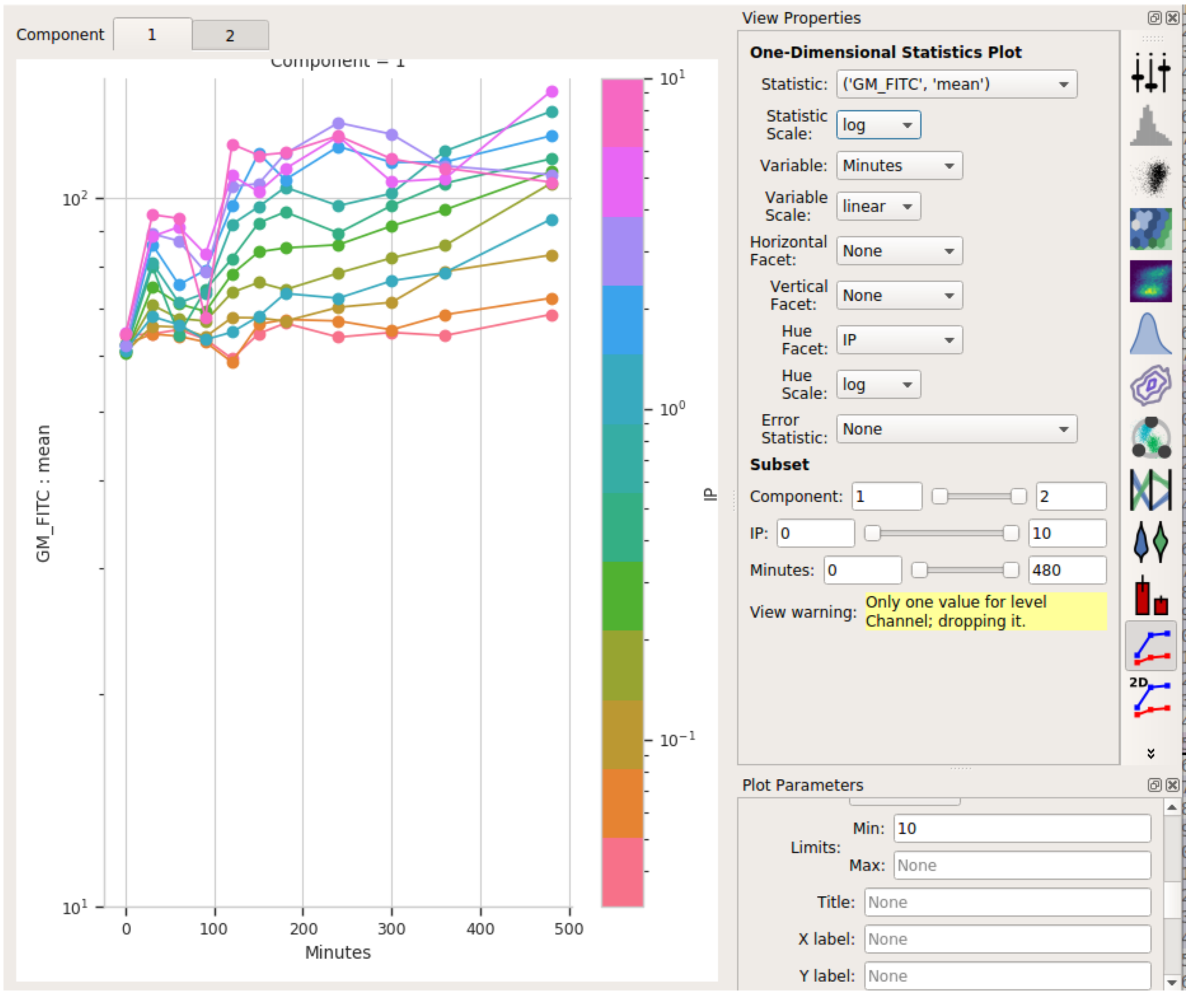


I'd say that's a pretty good fit!

It's important to note that *most data-driven operations* **also** *add statistics* that contain information about the models they fit. In this case, the **1D Mixture Model** operation creates statistics named *mean* and *proportion*, containing the mean and proportion for each component for each data subset.

First, let's see if the means actually do stay the same for the two components:

- Select the **1D Statistics View**
- Set **Statistic** to *('GM_FITC', 'mean')* and the **Statistic scale** to *log*.
- Set **Variable** to *Minutes*. Leave the **Variable Scale** as *linear*.
- Set the **Hue facet** to *IP* and change the **Hue scale** to *log*.
- The tabs at the top of the plot window will show you the results for the different components. (Note that I also set the Y axis minimum to "10").



One-Dimensional Statistics Plot

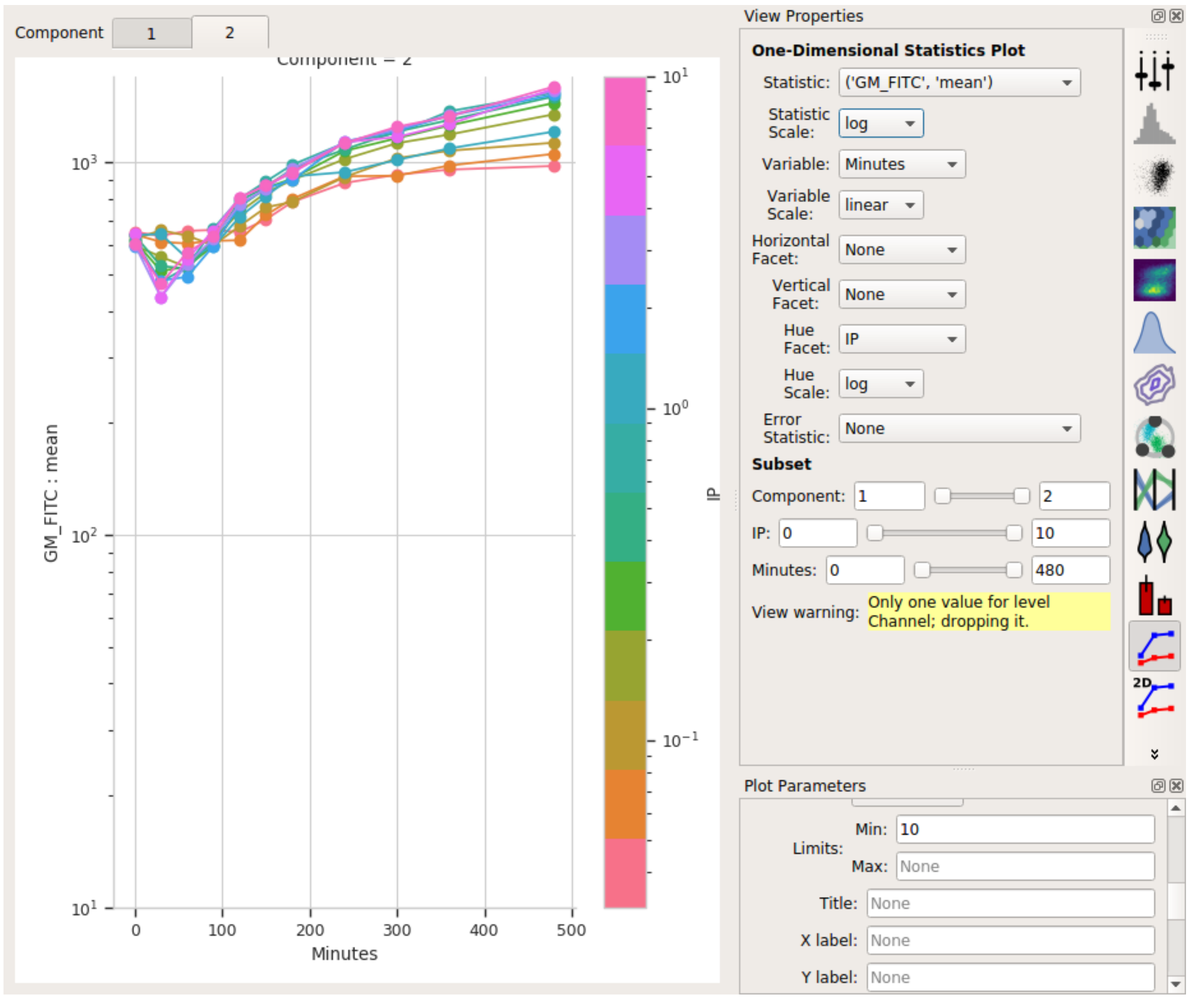
Statistic: ('GM_FITC', 'mean')
Statistic Scale: log
Variable: Minutes
Variable Scale: linear
Horizontal Facet: None
Vertical Facet: None
Hue Facet: IP
Hue Scale: log
Error Statistic: None

Subset

Component: 1 – 2
IP: 0 – 10
Minutes: 0 – 480

View warning: Only one value for level Channel; dropping it.

Limits: Min: 10 Max: None
Title: None
X label: None
Y label: None



One-Dimensional Statistics Plot

Statistic: ('GM_FITC', 'mean')

Statistic Scale: log

Variable: Minutes

Variable Scale: linear

Horizontal Facet: None

Vertical Facet: None

Hue Facet: IP

Hue Scale: log

Error Statistic: None

Subset

Component: 1 2

IP: 0 10

Minutes: 0 480

View warning: Only one value for level Channel; dropping it.

Limits: Min: 10 Max: None

Title: None

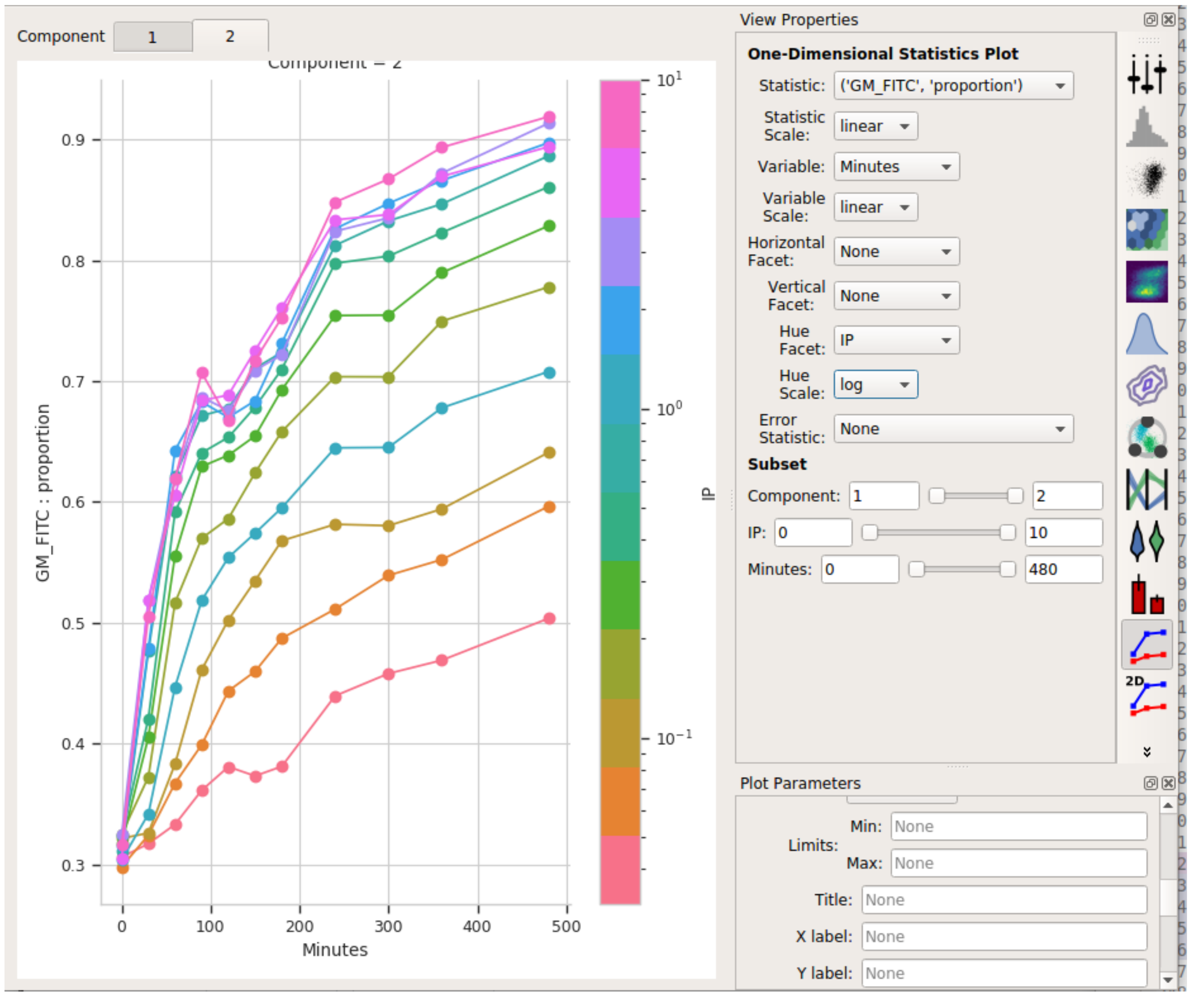
X label: None

Y label: None

So the means stay pretty constant? They change a lot less than the geometric mean does, at least. A little increase over time – about 5-fold – for the "high" population, and a more-chaotic but still some increase over time for the "low" population.

Second, let's see if the proportion in the "high" component changes:

- Set **Statistic** to *('GM_FITC', 'proportion')*
- Change the **Statistic scale** back to *linear*.
- Leave the **Variable** set to *Minutes*, the **Variable scale** on *linear**, the **Hue facet** on *IP* and the **Hue scale** on *log*.
- If you changed the Y axis minimum, reset it to nothing (default).
- Select Component *2* in the tabs at the top of the plot window.



One-Dimensional Statistics Plot

Statistic: ('GM_FITC', 'proportion')
Statistic Scale: linear
Variable: Minutes
Variable Scale: linear
Horizontal Facet: None
Vertical Facet: None
Hue Facet: IP
Hue Scale: log
Error Statistic: None

Subset

Component: 1 — 2
IP: 0 — 10
Minutes: 0 — 480

Limits: Min: None Max: None
Title: None
X label: None
Y label: None

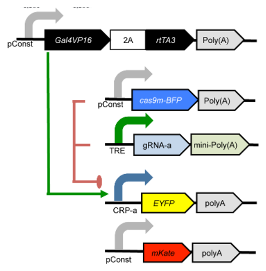
I think those dynamics look significantly different. For one thing, the mixture model "saturates" much more quickly – both in time and in IP. The geometric mean model indicates saturation at about 5 uM, while the mixture model seems to saturate one or two steps earlier. Things also stop changing quite as dramatically by about 240 minutes, whereas the geometric mean hasn't reached anything like a steady state by 480 minutes (the end of the experiment.)

I hope this has demonstrated a non-trivial insight into the dynamics of this biological system that are gained by looking at it through a quantitative lens, with some machine learning thrown in there as well.

5.1.5 Tutorial: Synthetic Gene Logic Network

This example reproduces Figure 2, part (a), from Kiani et al, Nature Methods 11: 723 (2014). This experiment uses a dCas9 fusion to repress the output of a yellow fluorescent reporter. The dCas9 is directed to the repressible promoter by a guide RNA under the control of rtTA3, a transcriptional activator controlled with the small molecule inducer doxycycline (Dox).

The plasmids that were co-transfected are shown below (reproduced from the above publication's Supplementary Figure 6a.)

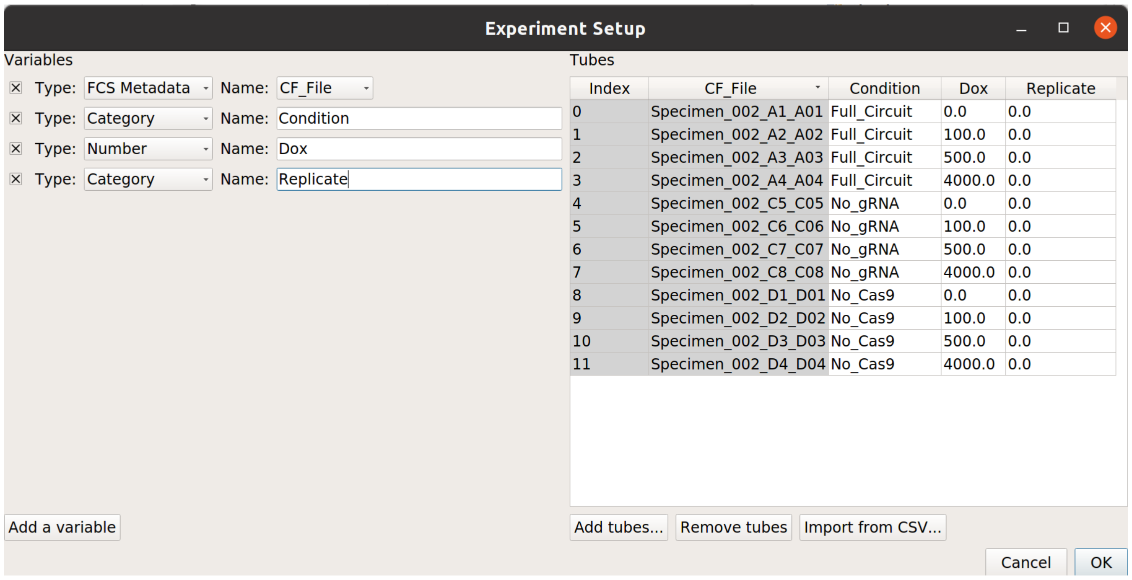


If you'd like to follow along, you can do so by downloading one of the **cytoflow-#####-examples-advanced.zip** files from the Cytoflow releases page on GitHub. The files are in the **kiani/** subdirectory.

Warning: This is a pretty big data set; on modest computers, the operations can take quite some time to complete. Be patient!

Import the data

- Start Cytoflow. Under the **Import Data** operation, choose **Set up experiment…**
- Add three variables, *Condition* and *Dox*, and *Replicate*. Make *Condition* and *Replicate Category*s, and make *Dox* a *Number*.
- Each replicate is in a separate subdirectory, with identical filenames. Here's the mapping from filename to conditions for one replicate:


Experiment Setup

Variables

☒	Type: FCS Metadata	Name: CF_File
☒	Type: Category	Name: Condition
☒	Type: Number	Name: Dox
☒	Type: Category	Name: Replicate

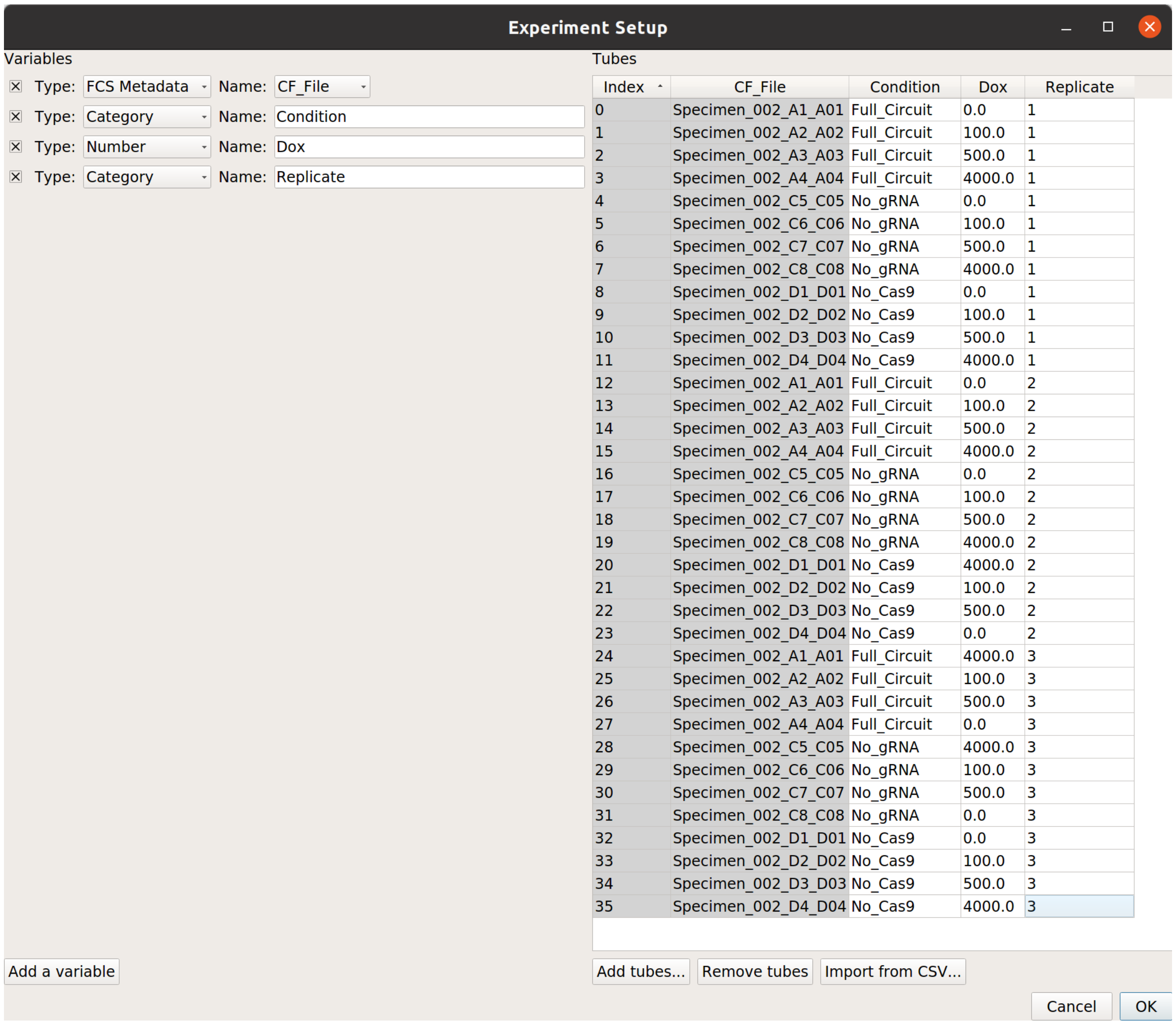
Tubes

Index	CF_File	Condition	Dox	Replicate
0	Specimen_002_A1_A01	Full_Circuit	0.0	0.0
1	Specimen_002_A2_A02	Full_Circuit	100.0	0.0
2	Specimen_002_A3_A03	Full_Circuit	500.0	0.0
3	Specimen_002_A4_A04	Full_Circuit	4000.0	0.0
4	Specimen_002_C5_C05	No_gRNA	0.0	0.0
5	Specimen_002_C6_C06	No_gRNA	100.0	0.0
6	Specimen_002_C7_C07	No_gRNA	500.0	0.0
7	Specimen_002_C8_C08	No_gRNA	4000.0	0.0
8	Specimen_002_D1_D01	No_Cas9	0.0	0.0
9	Specimen_002_D2_D02	No_Cas9	100.0	0.0
10	Specimen_002_D3_D03	No_Cas9	500.0	0.0
11	Specimen_002_D4_D04	No_Cas9	4000.0	0.0

Note: There are a *lot* of rows in this table. Two things can make setting up these kinds of experiments easier. First, if you already have the details in a table, you can import that table by following the instructions at *HOWTO: Import an experiment from a table*. And second, you can select multiple cells in the table to edit at once by holding **Control** or **Command** and clicking multiple cells.

Warning: It is generally not a good idea to name a variable **Time**, because most flow cytometers produce FCS files with a **Time** "channel" and you can't re-use those names!

At the end, your table should look like this:

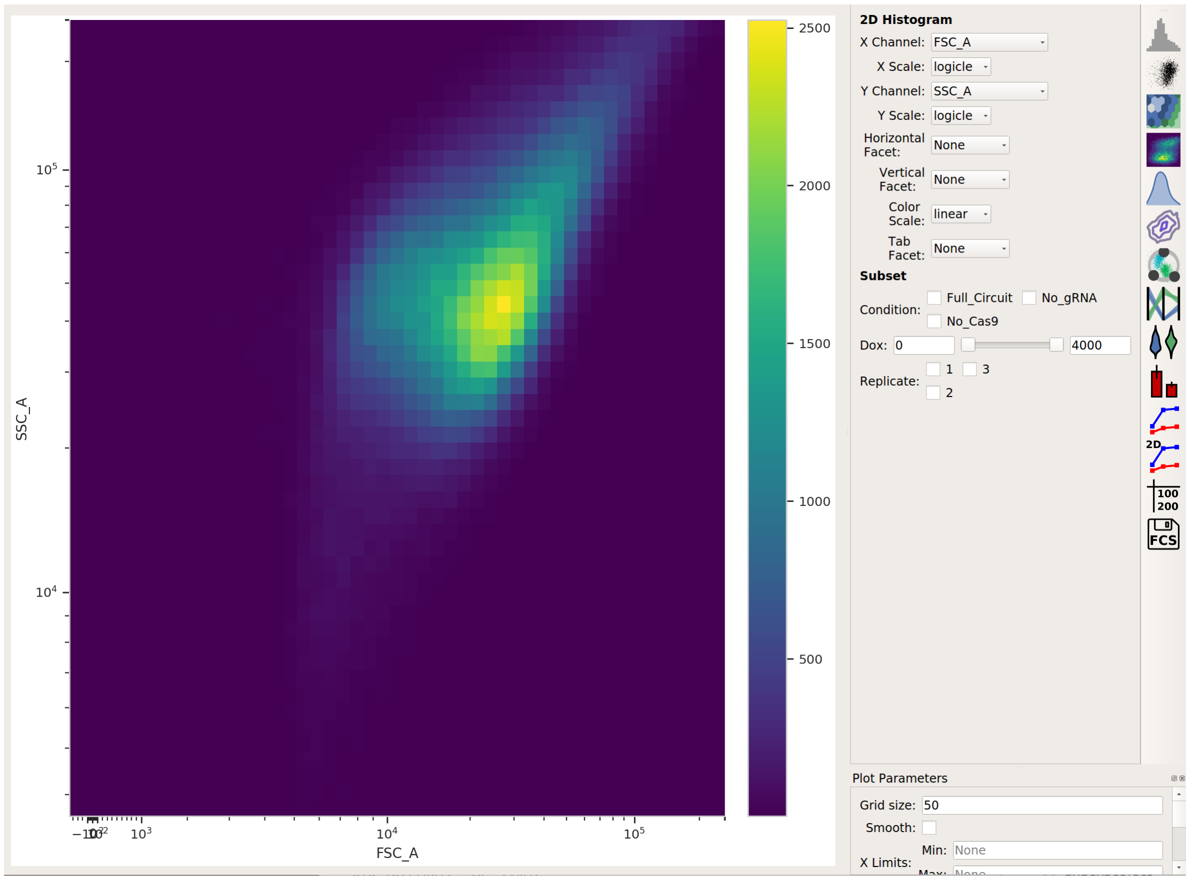


Type:	FCS Metadata	Name:	CF_File
Type:	Category	Name:	Condition
Type:	Number	Name:	Dox
Type:	Category	Name:	Replicate

Index	CF_File	Condition	Dox	Replicate
0	Specimen_002_A1_A01	Full_Circuit	0.0	1
1	Specimen_002_A2_A02	Full_Circuit	100.0	1
2	Specimen_002_A3_A03	Full_Circuit	500.0	1
3	Specimen_002_A4_A04	Full_Circuit	4000.0	1
4	Specimen_002_C5_C05	No_gRNA	0.0	1
5	Specimen_002_C6_C06	No_gRNA	100.0	1
6	Specimen_002_C7_C07	No_gRNA	500.0	1
7	Specimen_002_C8_C08	No_gRNA	4000.0	1
8	Specimen_002_D1_D01	No_Cas9	0.0	1
9	Specimen_002_D2_D02	No_Cas9	100.0	1
10	Specimen_002_D3_D03	No_Cas9	500.0	1
11	Specimen_002_D4_D04	No_Cas9	4000.0	1
12	Specimen_002_A1_A01	Full_Circuit	0.0	2
13	Specimen_002_A2_A02	Full_Circuit	100.0	2
14	Specimen_002_A3_A03	Full_Circuit	500.0	2
15	Specimen_002_A4_A04	Full_Circuit	4000.0	2
16	Specimen_002_C5_C05	No_gRNA	0.0	2
17	Specimen_002_C6_C06	No_gRNA	100.0	2
18	Specimen_002_C7_C07	No_gRNA	500.0	2
19	Specimen_002_C8_C08	No_gRNA	4000.0	2
20	Specimen_002_D1_D01	No_Cas9	4000.0	2
21	Specimen_002_D2_D02	No_Cas9	100.0	2
22	Specimen_002_D3_D03	No_Cas9	500.0	2
23	Specimen_002_D4_D04	No_Cas9	0.0	2
24	Specimen_002_A1_A01	Full_Circuit	4000.0	3
25	Specimen_002_A2_A02	Full_Circuit	100.0	3
26	Specimen_002_A3_A03	Full_Circuit	500.0	3
27	Specimen_002_A4_A04	Full_Circuit	0.0	3
28	Specimen_002_C5_C05	No_gRNA	4000.0	3
29	Specimen_002_C6_C06	No_gRNA	100.0	3
30	Specimen_002_C7_C07	No_gRNA	500.0	3
31	Specimen_002_C8_C08	No_gRNA	0.0	3
32	Specimen_002_D1_D01	No_Cas9	0.0	3
33	Specimen_002_D2_D02	No_Cas9	100.0	3
34	Specimen_002_D3_D03	No_Cas9	500.0	3
35	Specimen_002_D4_D04	No_Cas9	4000.0	3

Gate out debris

There's a lot of data here; let's use a **Density View** to look at the **FSC-A** and **SSC-A** channels:



2D Histogram
X Channel: FSC_A
X Scale: logicle
Y Channel: SSC_A
Y Scale: logicle
Horizontal Facet: None
Vertical Facet: None
Color Scale: linear
Tab Facet: None

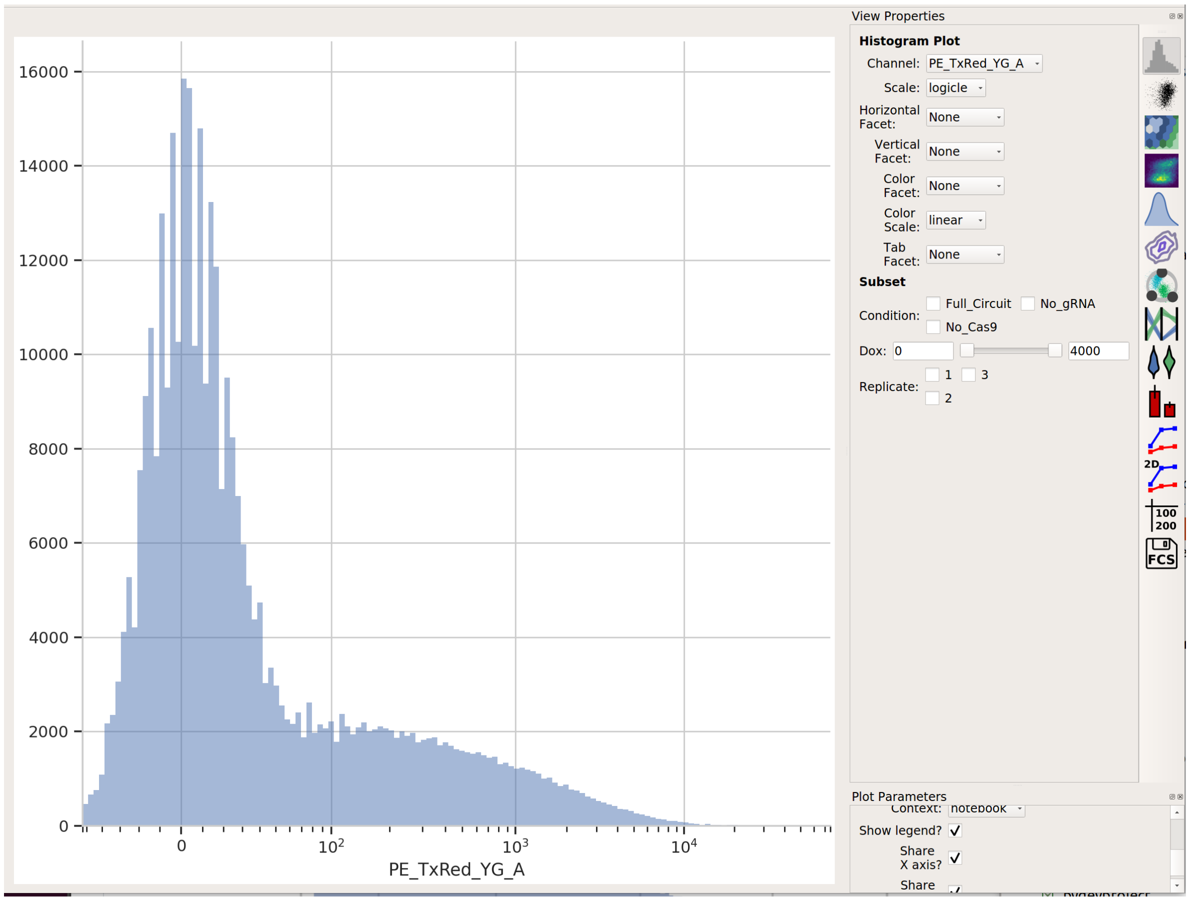
Subset
Condition: Full_Circuit No_gRNA No_Cas9
Dox: 0 4000
Replicate: 1 3 2

Plot Parameters
Grid size: 50
Smooth:
X Limits: Min: None

This looks like it's been pre-gated (ie, there's not a mixture of populations.) It's also pushed up against the top axes in both SSC-A and FSC-A, which is a little concerning, but shouldn't affect our analyses too much.

Select transfected cells

The next thing we usually do is select for positively transfected cells. mKate is the transfection marker, so look at the red (`PE_TxRed_YG_A`) channel:



View Properties

Histogram Plot

Channel: PE_TxRed_YG_A

Scale: logicle

Horizontal Facet: None

Vertical Facet: None

Color Facet: None

Color Scale: linear

Tab Facet: None

Subset

Condition: Full_Circuit No_gRNA No_Cas9

Dox: 0 4000

Replicate: 1 3 2

Plot Parameters

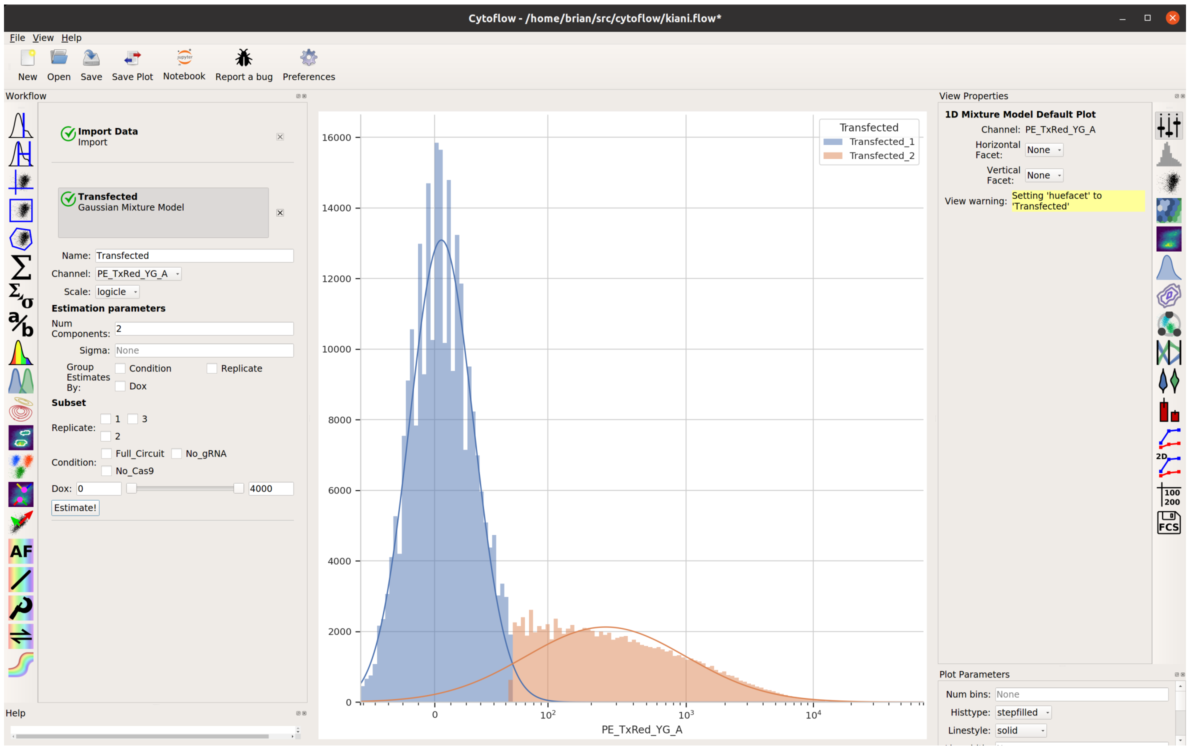
Context: notebook

Show legend? ✓

Share X axis? ✓

Share

Let's fit a mixture-of-gaussians, for a nice principled way of separating the transfected population from the untransfected population.



Import Data
Import

Transfected
Gaussian Mixture Model

Name: Transfected
Channel: PE_TxRed_YG_A
Scale: logicle

Estimation parameters

Num Components: 2
Sigma: None
Group Estimates By: Condition, Replicate, Dox

Subset

Replicate: 1, 3, 2
Condition: Full_Circuit, No_gRNA, No_Cas9
Dox: 0 – 4000

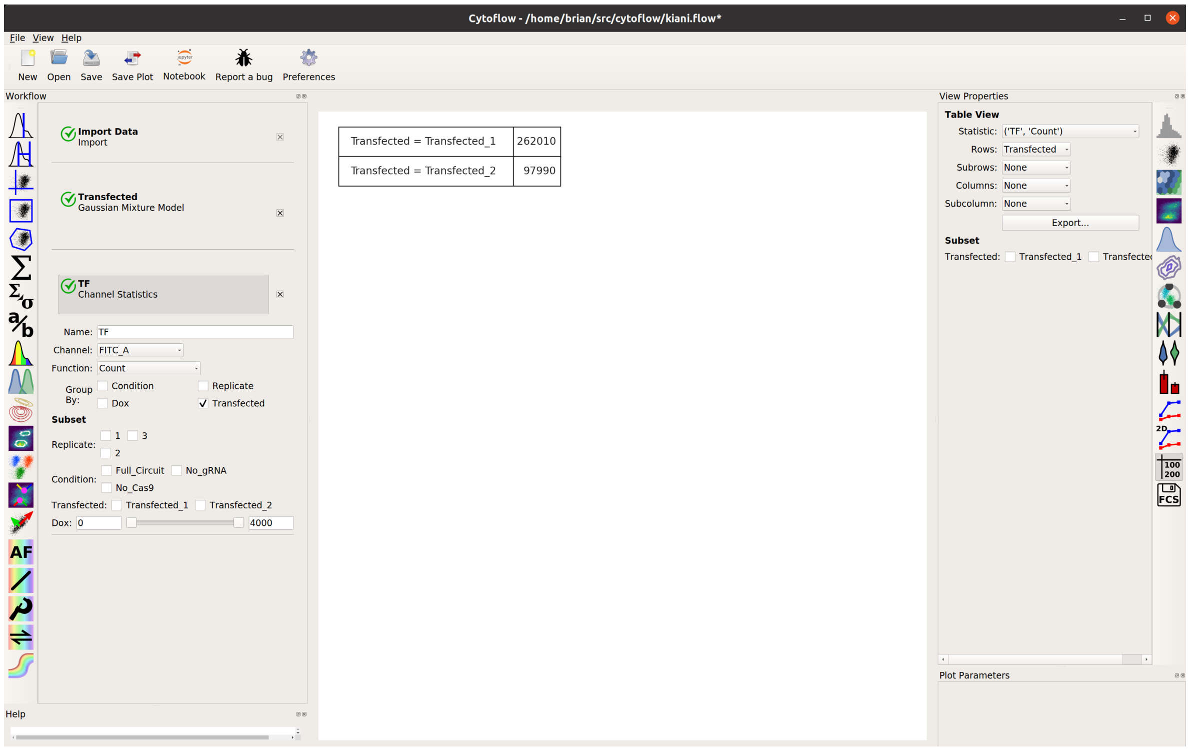
Estimate!

1D Mixture Model Default Plot

Channel: PE_TxRed_YG_A
Horizontal Facet: None
Vertical Facet: None
View warning: Setting 'huefacet' to 'Transfected'

Num bins: None
Histtype: stepfilled
Linestyle: solid

Looks good: the events with `Transfected_2 == True` are the cells in the transfected population. Let's make a statistic to see how many events are in each population:



Import Data
Import

Transfected
Gaussian Mixture Model

TF
Channel Statistics

Name: TF
Channel: FITC_A
Function: Count
Group By: Condition, Replicate, Dox, ✓ Transfected

Subset

Replicate: 1, 3, 2
Condition: Full_Circuit, No_gRNA, No_Cas9
Transfected: Transfected_1, Transfected_2
Dox: 0 – 4000

Transfected = Transfected_1	262010
Transfected = Transfected_2	97990

Table View

Statistic: ('TF', 'Count')
Rows: Transfected
Subrows: None
Columns: None
Subcolumn: None

Subset

Transfected: Transfected_1, Transfecte

Examine the function of the gene circuit

Now, we can reproduce the bar chart in the publication by taking the output (EYFP, in the FITC-A channel) geometric mean of the positively transfected cells, split out by condition and `Dox`. Don't forget to look at just the transfected cells (using subset). We'll compute the geometric mean across circuit and `Dox`, and then split it out by replicate so we can compute an SEM.

Please note: This is a terrible place to use error bars. See:

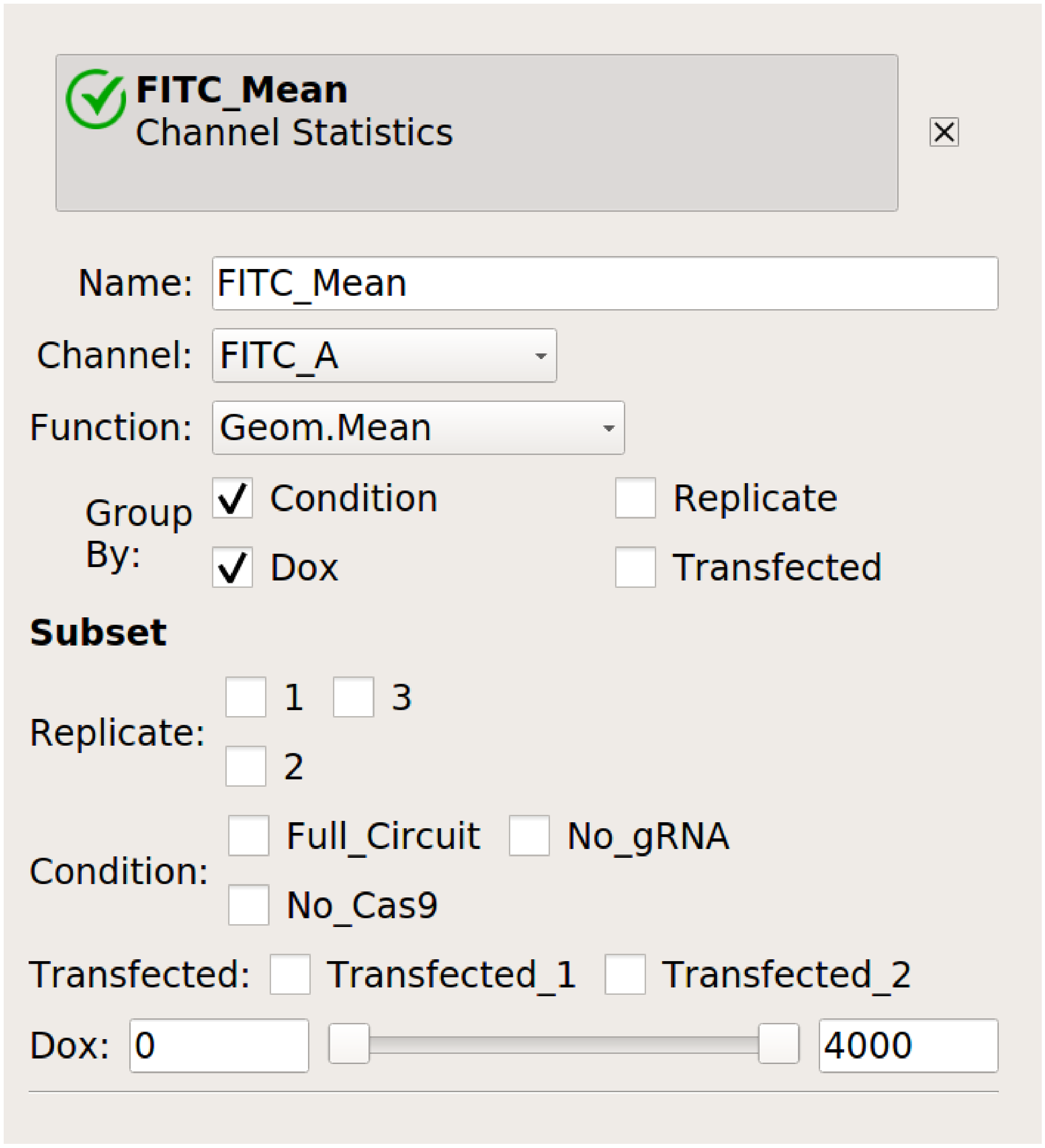
https://www.nature.com/nature/journal/v492/n7428/full/492180a.html

and

http://jcb.rupress.org/content/177/1/7

for the reason why. I'm using them here to demonstrate the capability, rather than argue that you should perform your analysis this way.

- First, make a statistic with the overall geometric mean (by condition and Dox):



FITC_Mean
Channel Statistics

Name: FITC_Mean

Channel: FITC_A

Function: Geom.Mean

Group By: [✓ Condition] Replicate [✓ Dox] Transfected

Subset

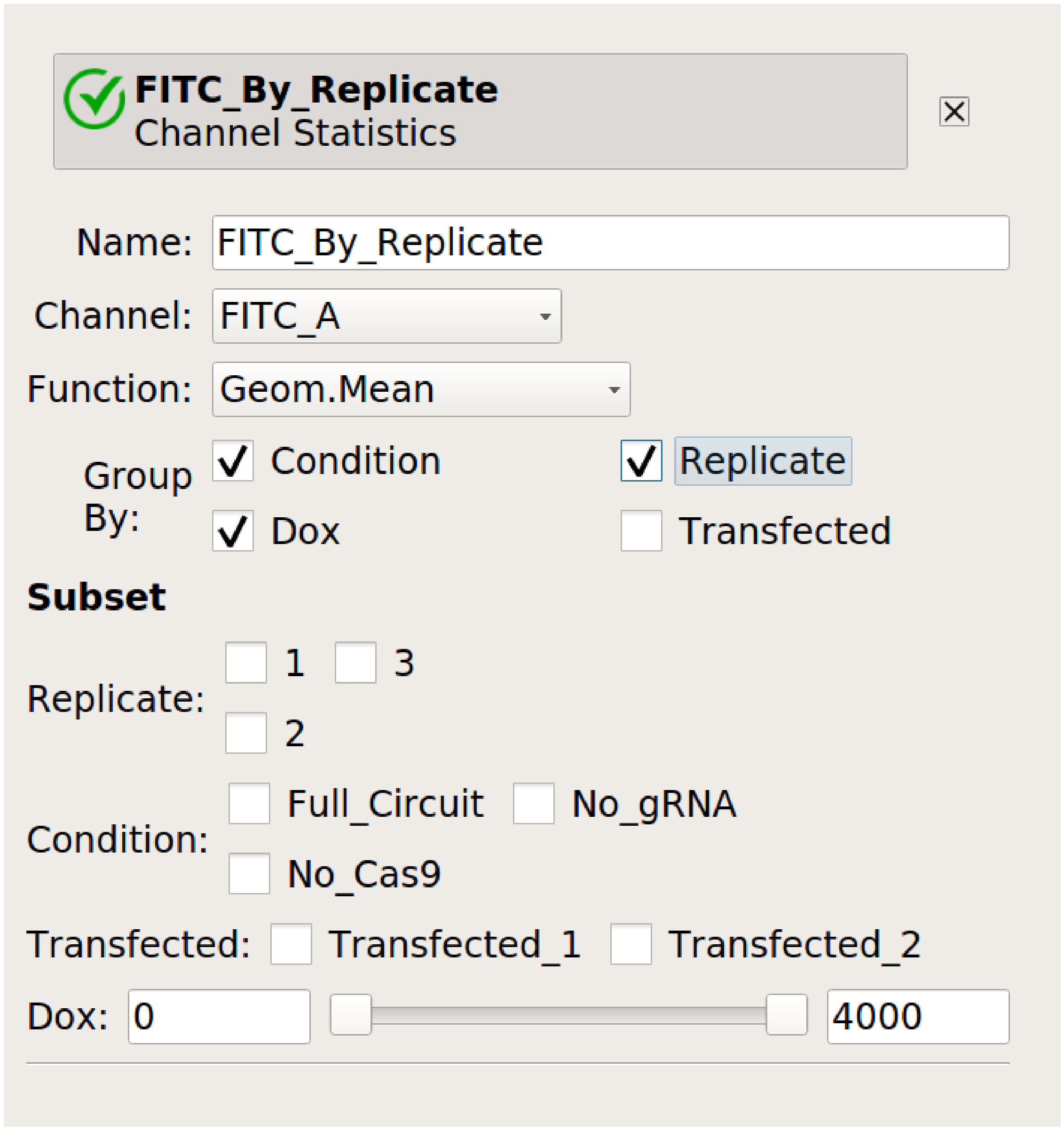
Replicate: 1 3 2

Condition: Full_Circuit No_gRNA No_Cas9

Transfected: Transfected_1 Transfected_2

Dox: 0 4000

- Next, make a statistic with the geometric mean broken out by condition, Dox *and replicate*.


FITC_By_Replicate
Channel Statistics

Name: FITC_By_Replicate

Channel: FITC_A

Function: Geom.Mean

Group By: [✓ Condition] [✓ Replicate] [✓ Dox] Transfected

Subset

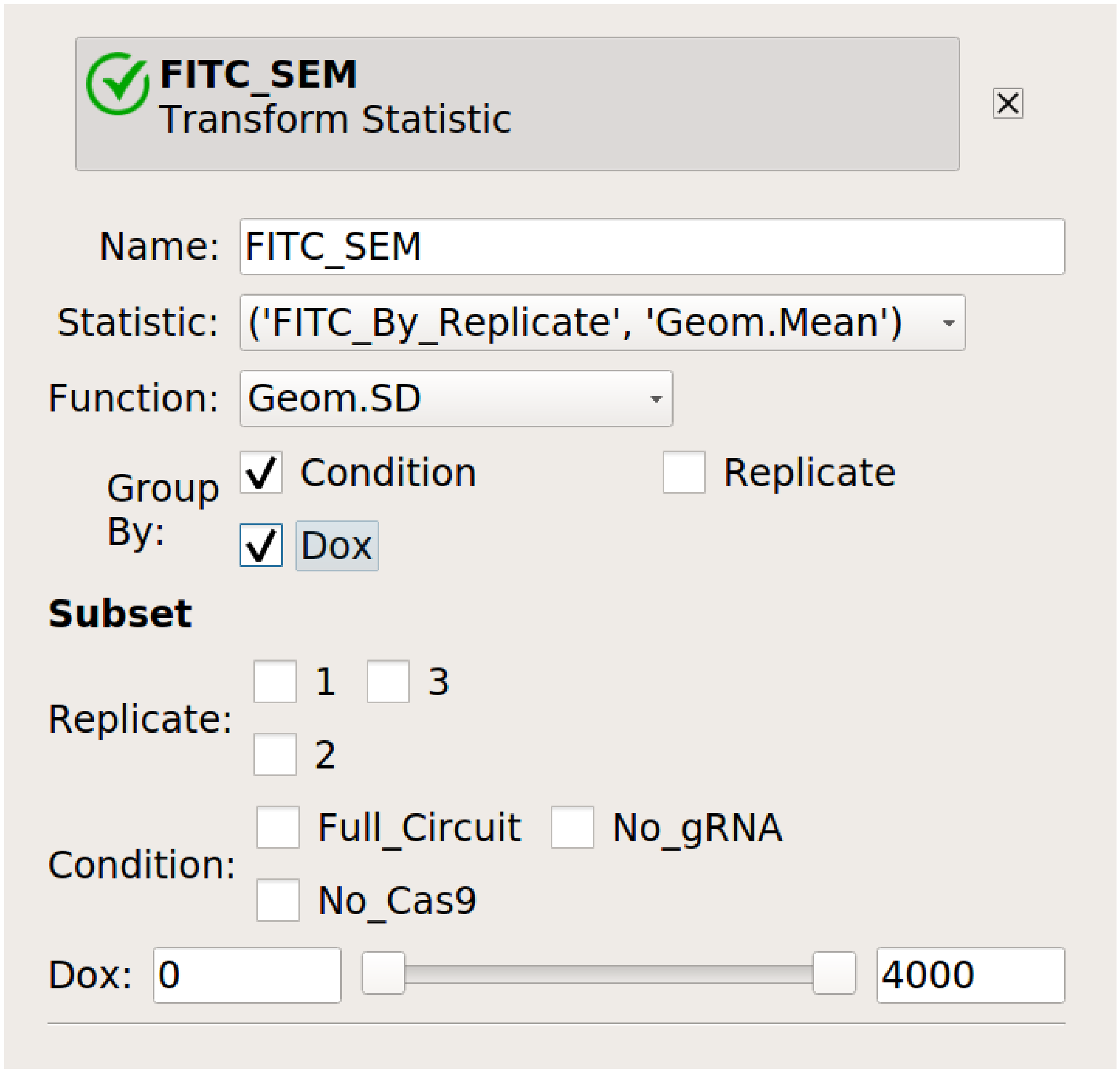
Replicate: 1 3 2

Condition: Full_Circuit No_gRNA No_Cas9

Transfected: Transfected_1 Transfected_2

Dox: 0 4000

- Finally, compute the geometric standard deviation of the mean:


FITC_SEM
Transform Statistic

Name: FITC_SEM

Statistic: ('FITC_By_Replicate', 'Geom.Mean')

Function: Geom.SD

Group By: [✓ Condition] Replicate [✓ Dox]

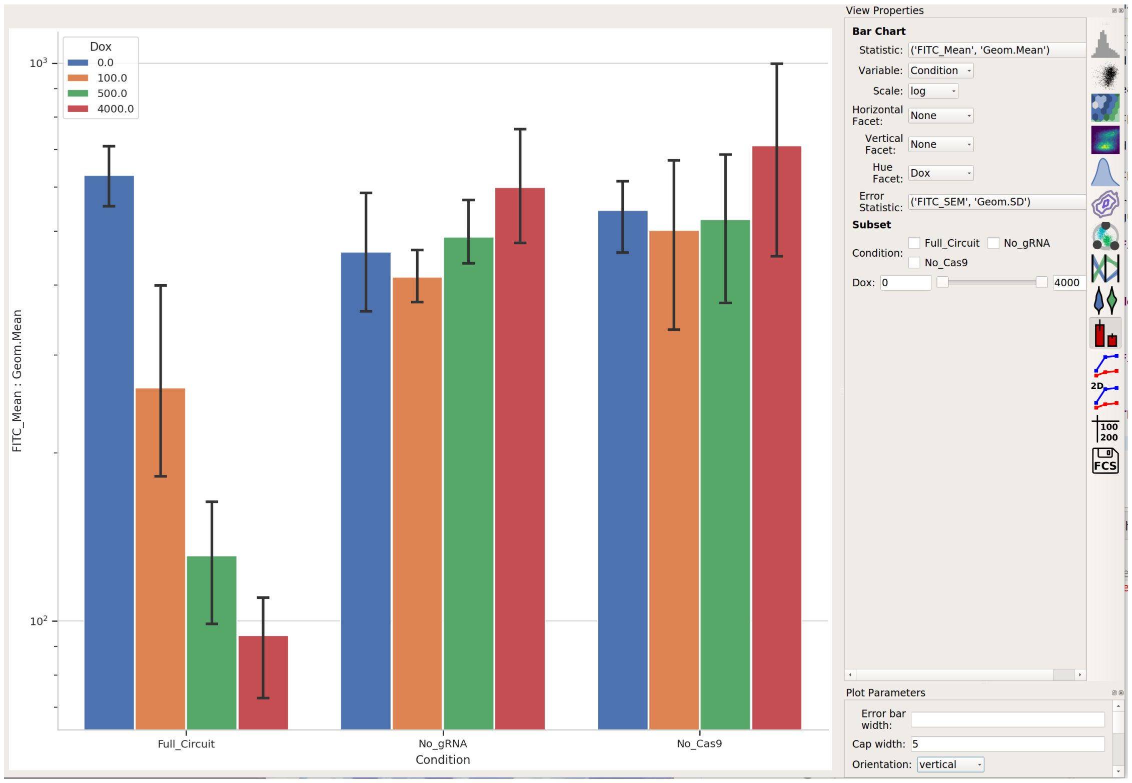
Subset

Replicate: 1 3 2

Condition: Full_Circuit No_gRNA No_Cas9

Dox: 0 4000

- Then plot them together:



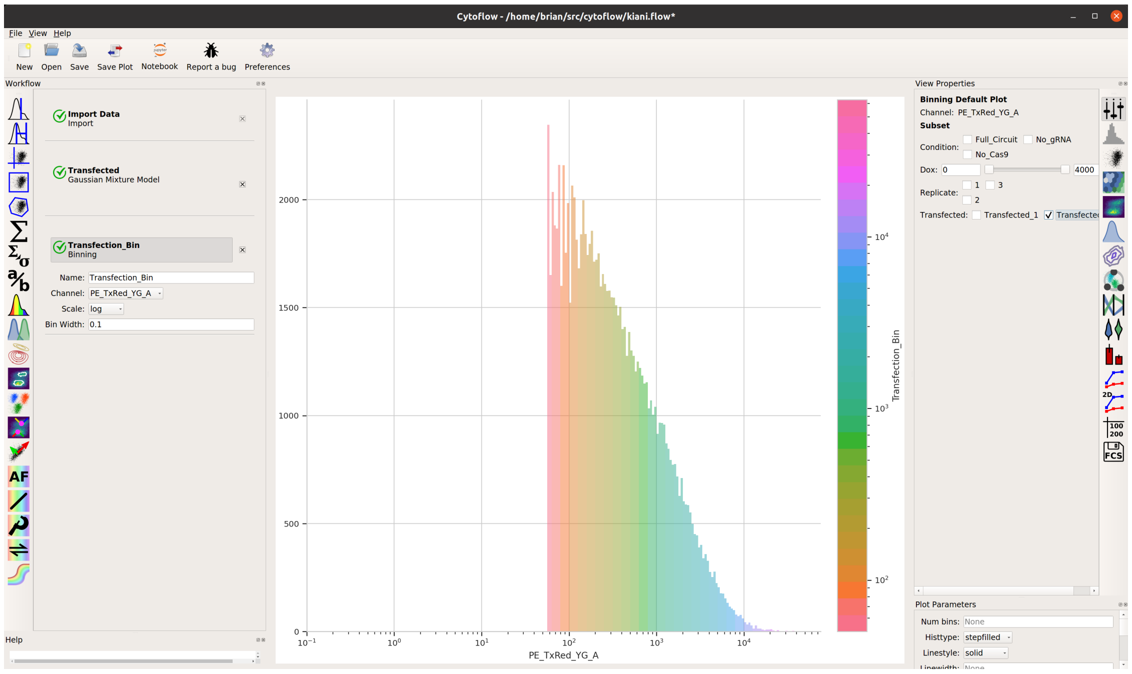
Bar Chart
Statistic: ('FITC_Mean', 'Geom.Mean')
Variable: Condition
Scale: log
Horizontal Facet: None
Vertical Facet: None
Hue Facet: Dox
Error Statistic: ('FITC_SEM', 'Geom.SD')
Subset
Condition: Full_Circuit No_gRNA No_Cas9
Dox: 0 4000

Error bar width:
Cap width: 5
Orientation: vertical

Further exploration

So that's useful, but maybe there's more in this data. We've noticed in our lab that gene circuit behavior frequently changes as copy number changes. Is this the case here? We can bin the data by transfection level, and see if the behavior changes as the bin number increases.

- Add a binning operation:



Import Data
Import

Transfected
Gaussian Mixture Model

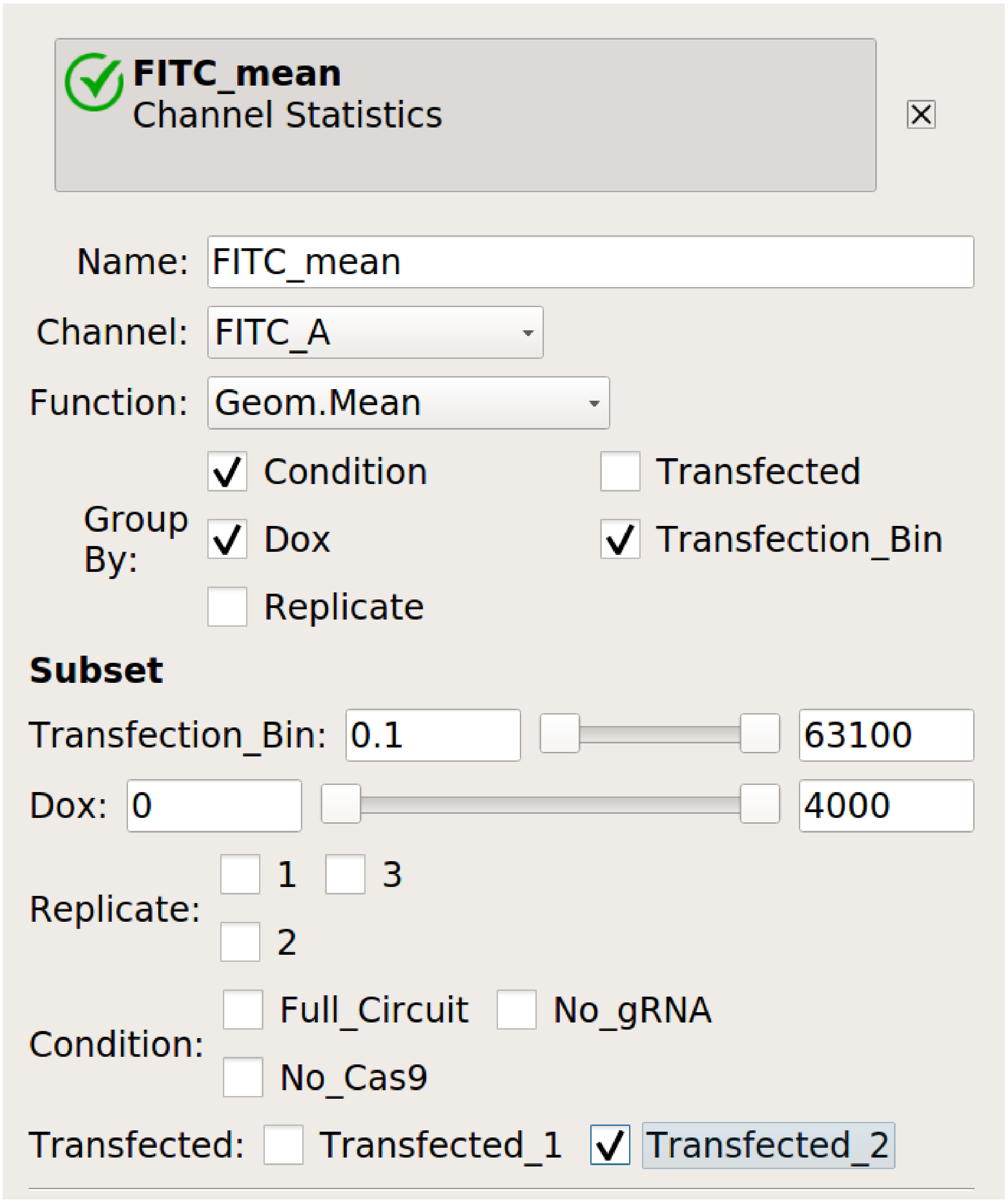
Transfection_Bin
Binning

Name: Transfection_Bin
Channel: PE_TxRed_YG_A
Scale: log
Bin Width: 0.1

Binning Default Plot
Channel: PE_TxRed_YG_A
Subset
Condition: Full_Circuit No_gRNA No_Cas9
Dox: 0 4000
Replicate: 1 3 2
Transfected: Transfected_1 ✓ Transfecte

Num bins: None
Histtype: stepfilled
Linestyle: solid
Linewidth: None

- Make a statistic that computes the mean FITC signal in each bin:



FITC_mean
Channel Statistics

Name: FITC_mean

Channel: FITC_A

Function: Geom.Mean

Group By: [✓ Condition] [ ] Transfected [✓ Dox] [✓ Transfection_Bin] [ ] Replicate

Subset

Transfection_Bin: 0.1 – 63100

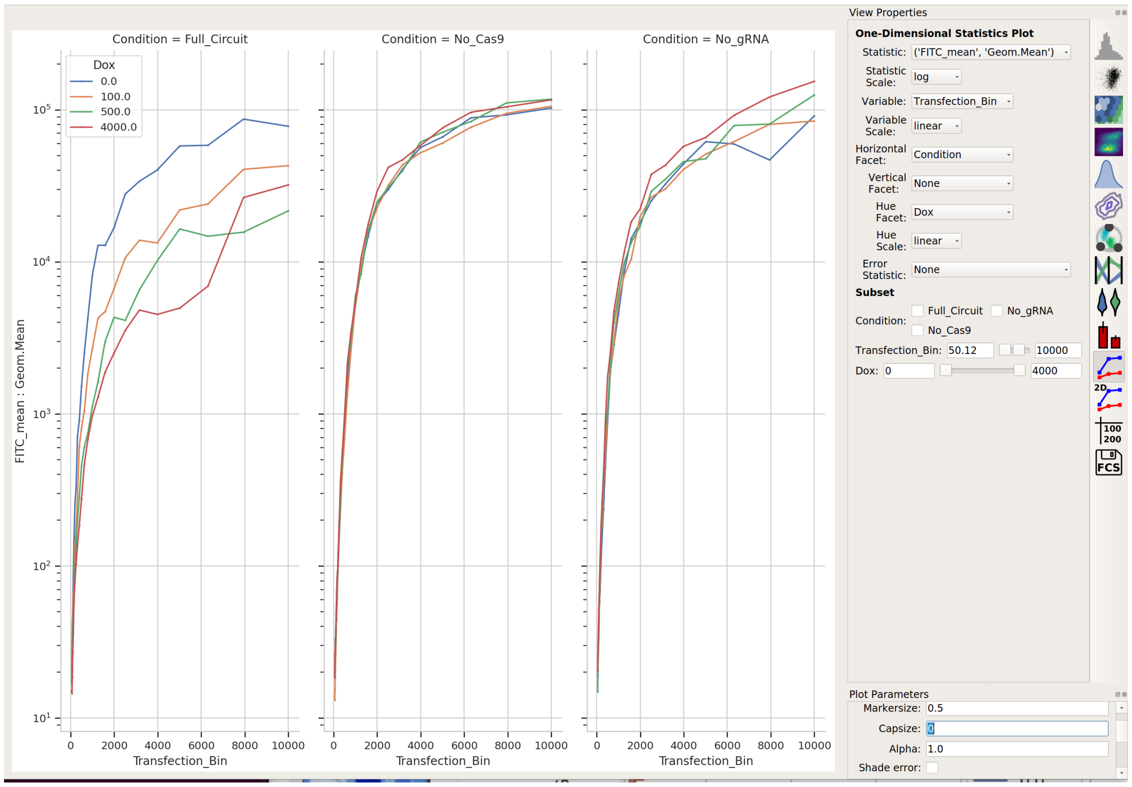
Dox: 0 – 4000

Replicate: [ ] 1 [ ] 3 [ ] 2

Condition: [ ] Full_Circuit [ ] No_gRNA [ ] No_Cas9

Transfected: [ ] Transfected_1 [✓ Transfected_2]

- Does it change as the bin number increases?



View Properties

One-Dimensional Statistics Plot

Statistic: ('FITC_mean', 'Geom.Mean')
Statistic Scale: log
Variable: Transfection_Bin
Variable Scale: linear
Horizontal Facet: Condition
Vertical Facet: None
Hue Facet: Dox
Hue Scale: linear
Error Statistic: None

Subset

Condition: Full_Circuit No_gRNA No_Cas9
Transfection_Bin: 50.12 10000
Dox: 0 4000

Plot Parameters

Markersize: 0.5
Capsize: 0
Alpha: 1.0
Shade error:

I would say it does!

2. *HOWTOs* are "recipes" – step-by-step guides to accomplish a particular task. They are a little higher-level than the tutorials.

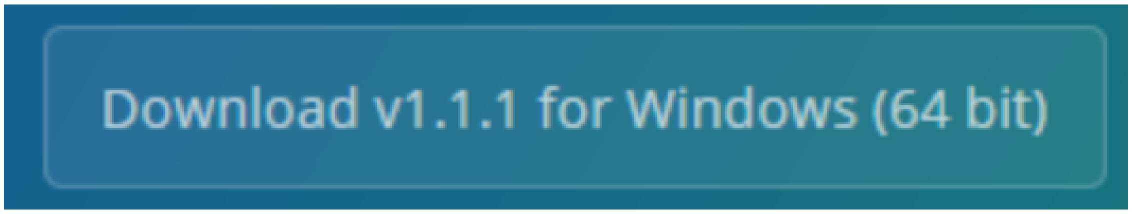
5.1.6 HOWTO: Install Cytoflow

Windows

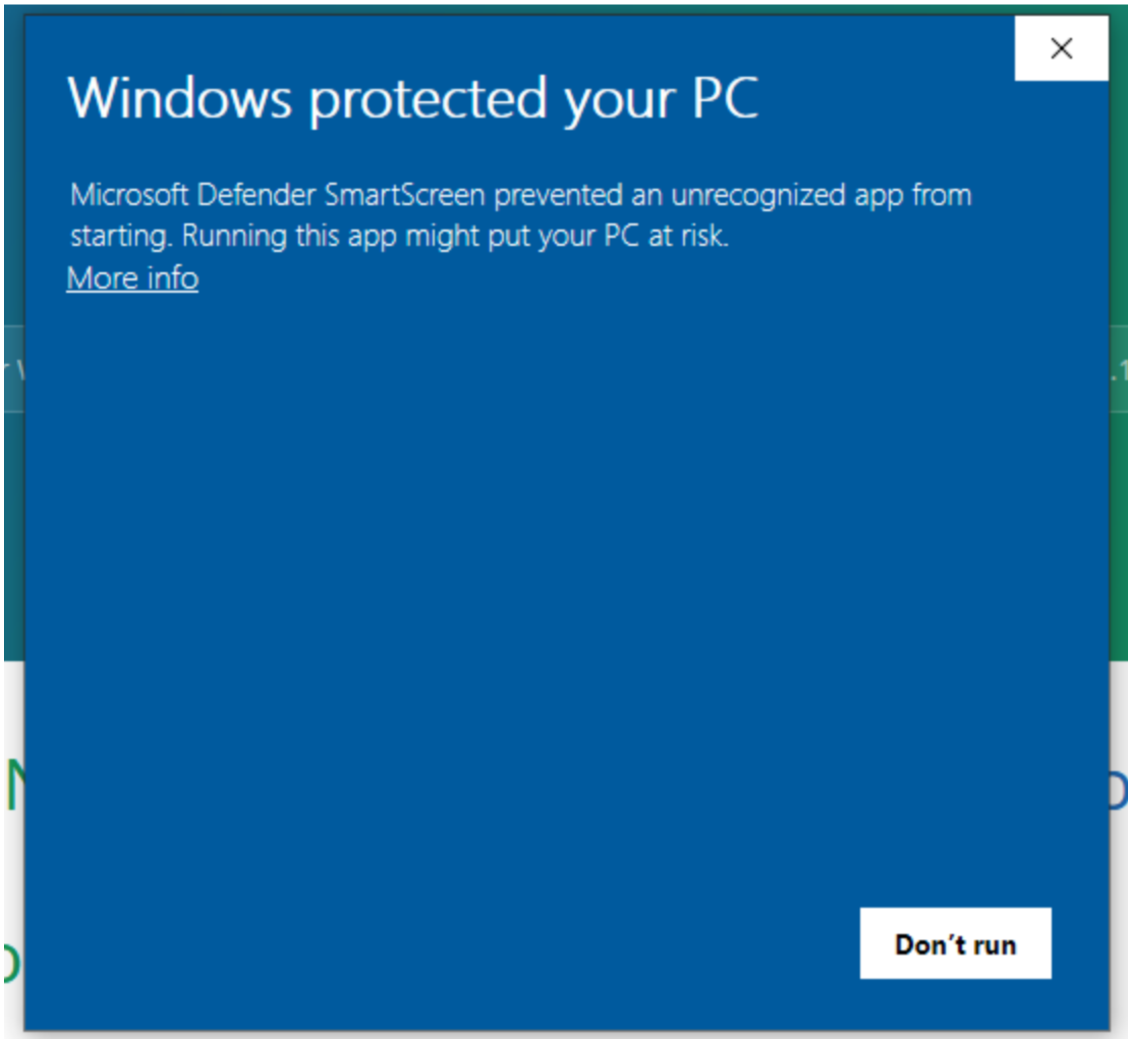
On Windows, you can use a graphical to install Cytoflow.

(a) Browse to https://cytoflow.github.io/

(b) Download the Windows binaries by clicking the button at the top of the page.



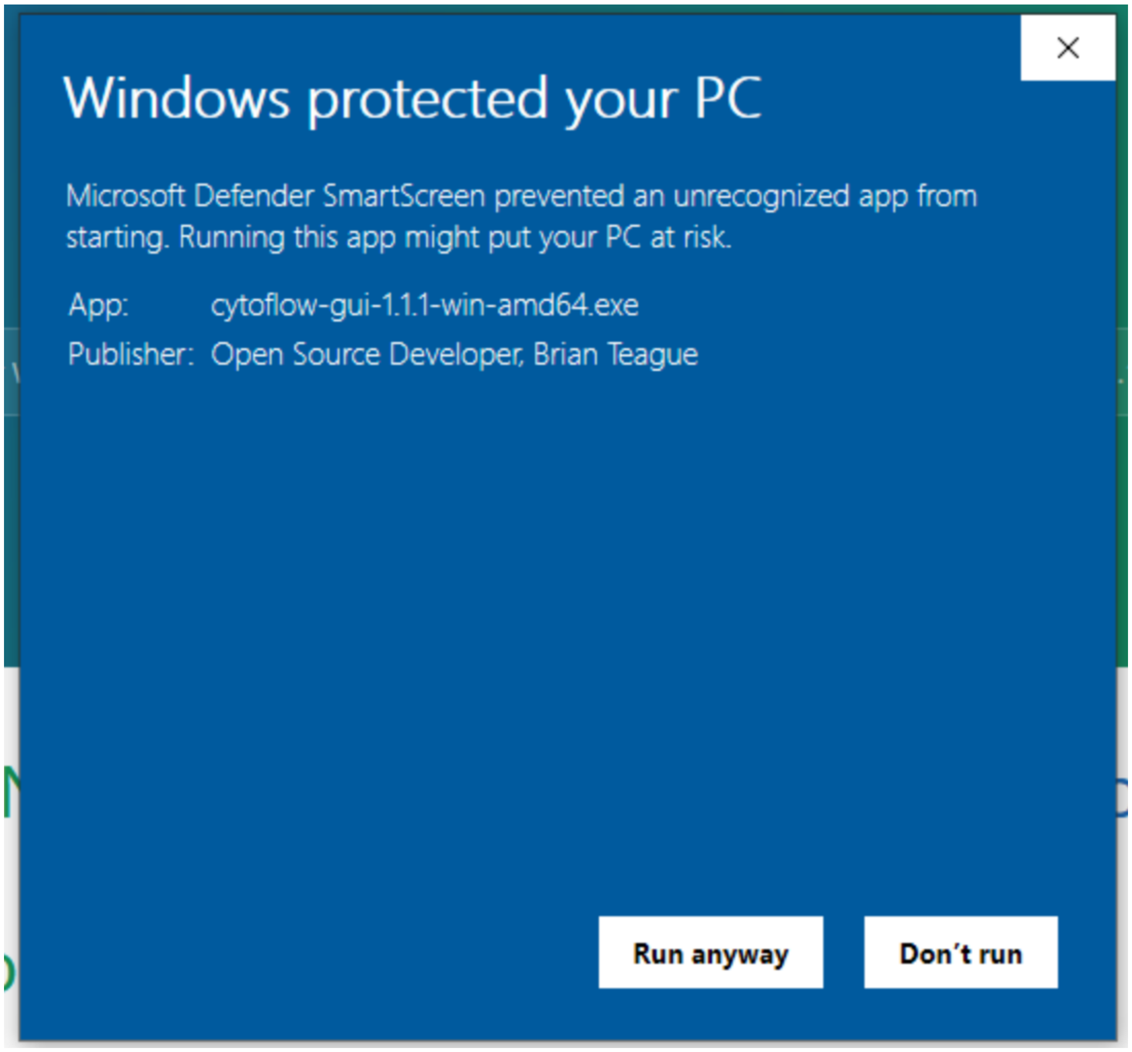
(c) Even though I've signed the Windows app, Windows doesn't recognize me as a "certified" developer. (It's EXPENSIVE – even more so than becoming an Apple developer.) So you'll get a `Windows protected your PC` message:



Windows protected your PC

Microsoft Defender SmartScreen prevented an unrecognized app from starting. Running this app might put your PC at risk.

However, if you click "More information", then you can verify that the installer is signed "Open Source Developer, Brian Teague" (that's me!) and then click "Run anyway."



Windows protected your PC

Microsoft Defender SmartScreen prevented an unrecognized app from starting. Running this app might put your PC at risk.

App: cytoflow-gui-1.1.1-win-amd64.exe

Publisher: Open Source Developer, Brian Teague

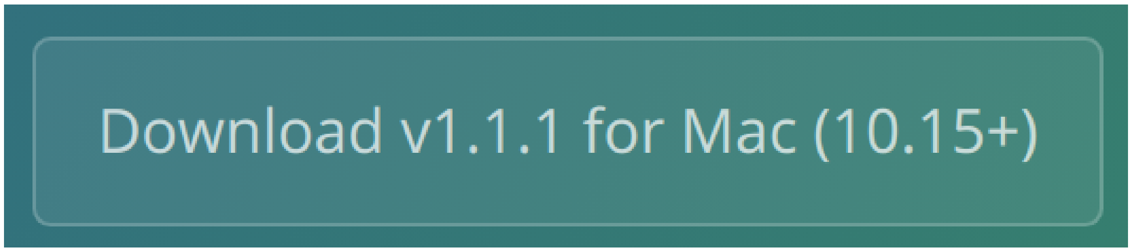
(d) Follow the instructions in the installer. Once you've completed the installation, you should be able to find `Cytoflow` in your Start menu.

MacOS

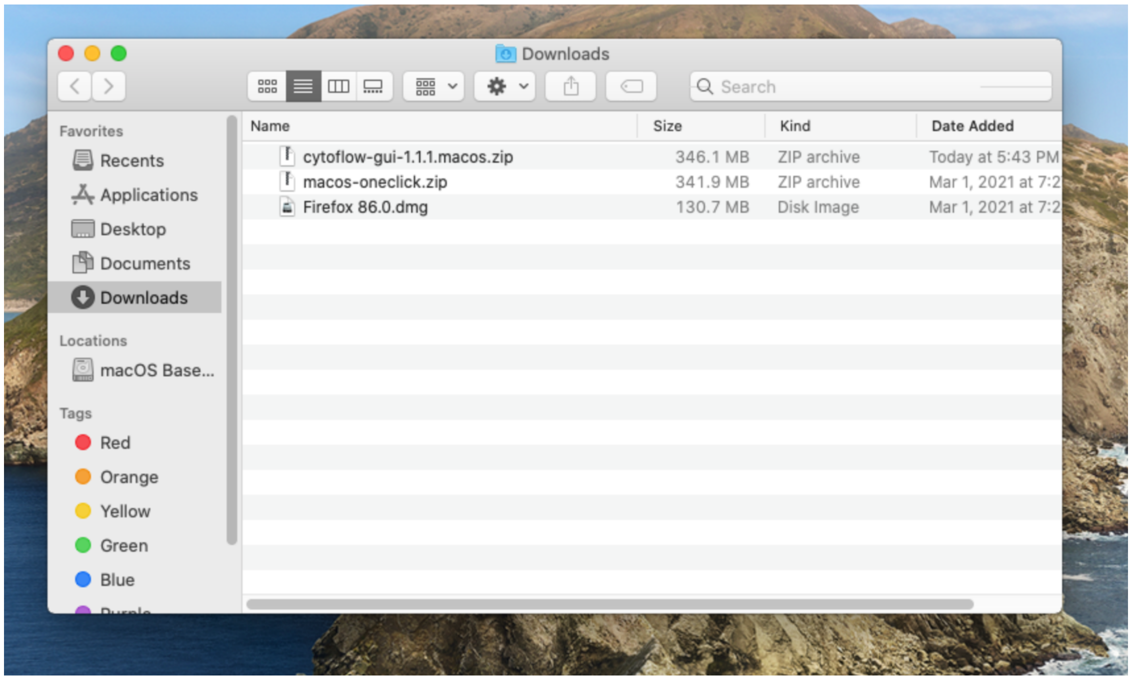
These instructions were developed using OSX Catalina. I don't own a Mac or use one on a regular basis, so if these instructions could be improved, please let me know. Also, these binaries were almost certainly built on an Intel Mac, so they may not work on an A1 mac.

(a) Browse to https://cytoflow.github.io/

(b) Download the MacOS binaries by clicking the button at the top of the page.

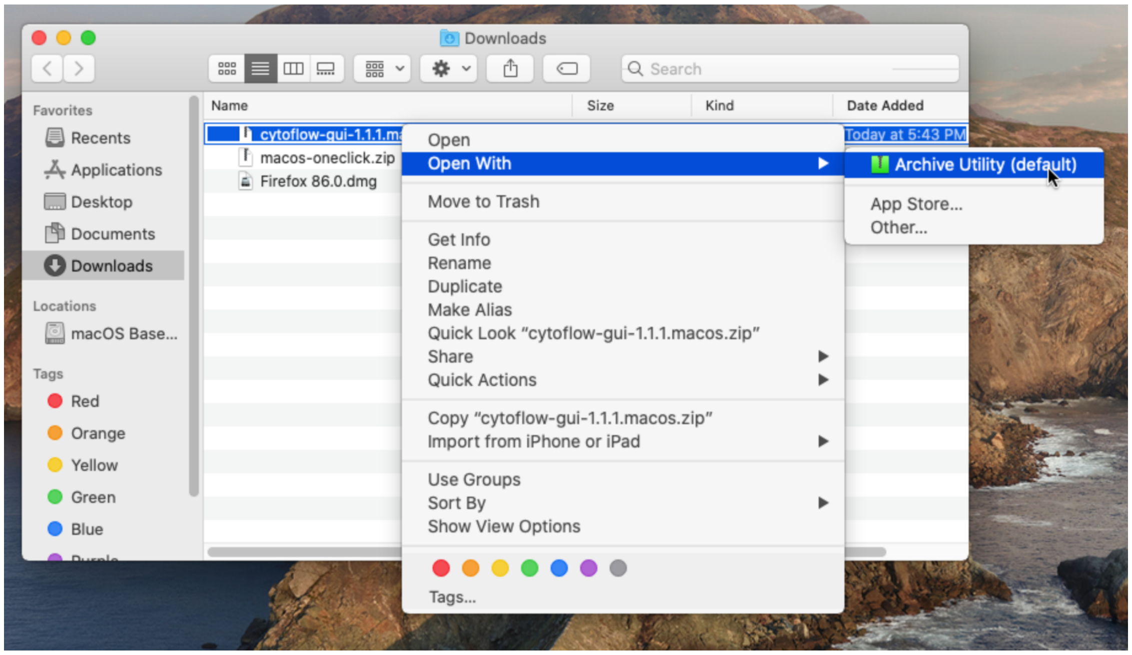


(c) Using Finder, browse to your Downloads folder.



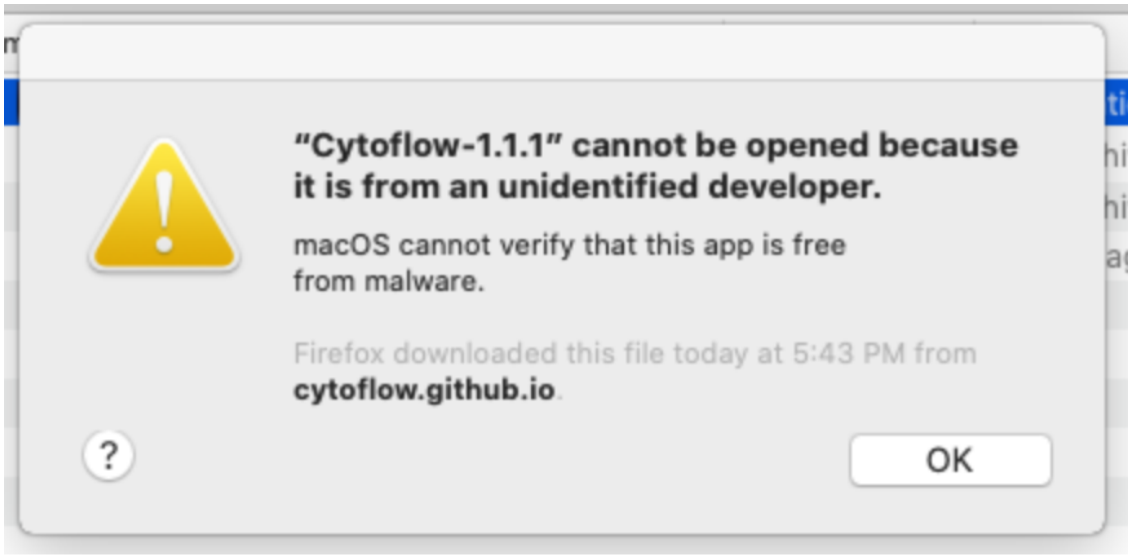
Name	Size	Kind	Date Added
cytoflow-gui-1.1.1.macos.zip	346.1 MB	ZIP archive	Today at 5:43 PM
macos-oneclick.zip	341.9 MB	ZIP archive	Mar 1, 2021 at 7:2
Firefox 86.0.dmg	130.7 MB	Disk Image	Mar 1, 2021 at 7:2

(d) Right-click on the .ZIP file you downloaded and choose “Open With –> Archive Utility.”



This will extract the application from the archive.

(e) Double-click the new application. Unfortuantely, I am not an Apple Developer, and because it costs like $100 a year, I likely never will be. So, the first time you run the application, you will be presented with a screen like the following:

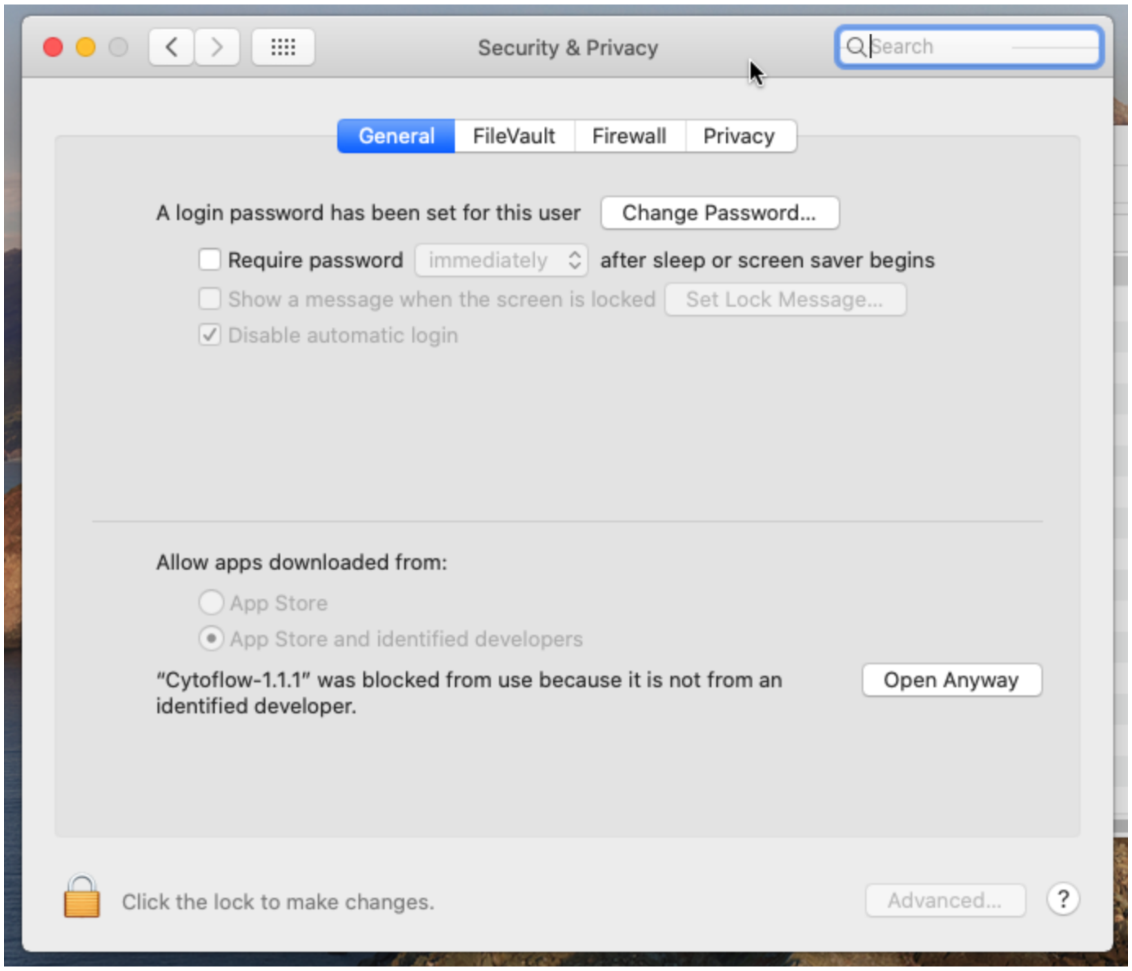


"Cytoflow-1.1.1" cannot be opened because it is from an unidentified developer.

macOS cannot verify that this app is free from malware.

Firefox downloaded this file today at 5:43 PM from cytoflow.github.io.

(f) To launch Cytoflow anyway, open `System Preferences` and select the `Security and Privacy` pane:



A login password has been set for this user [Change Password...]

☐ Require password [immediately] after sleep or screen saver begins

☐ Show a message when the screen is locked [Set Lock Message...]

☑ Disable automatic login

Allow apps downloaded from:

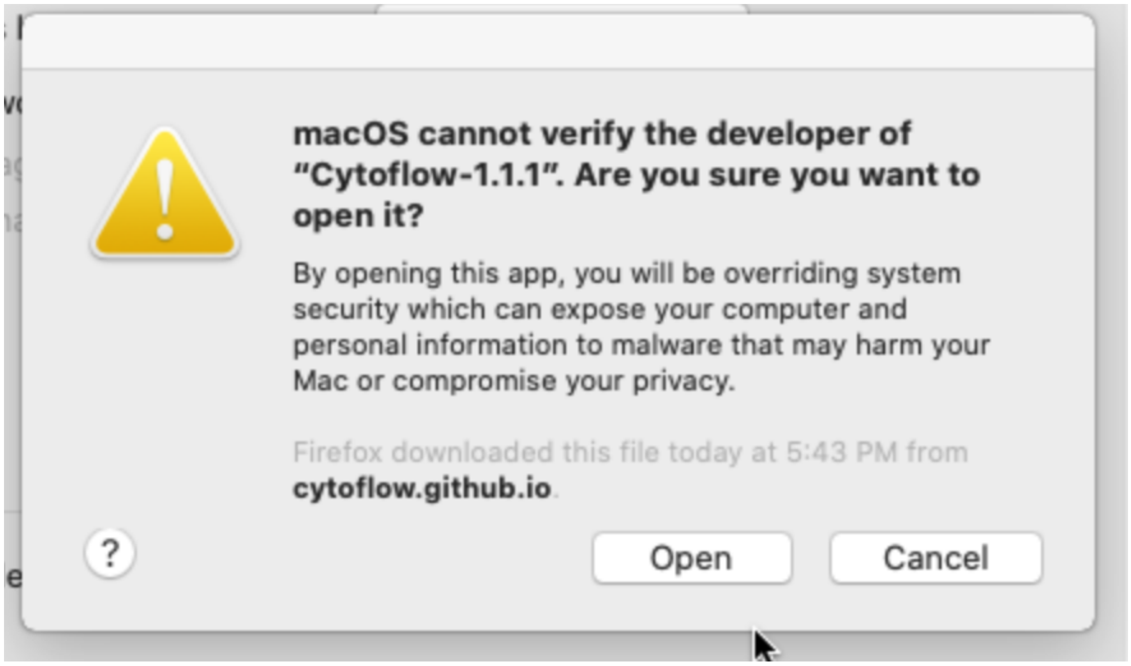
○ App Store

● App Store and identified developers

"Cytoflow-1.1.1" was blocked from use because it is not from an identified developer. [Open Anyway]

Click the lock to make changes.

(g) Click the **Open Anyway** button. Because MacOS wants you to be EXTRA SURE, you will be asked one more time if you're sure you know what you're doing:


macOS cannot verify the developer of "Cytoflow-1.1.1". Are you sure you want to open it?

By opening this app, you will be overriding system security which can expose your computer and personal information to malware that may harm your Mac or compromise your privacy.

Firefox downloaded this file today at 5:43 PM from **cytoflow.github.io**.

Click **Open** and after a moment, `Cytoflow` should launch.

(h) (Optional) Move the Cytoflow application to somewhere else "permanent", like the desktop or your Applications folder.

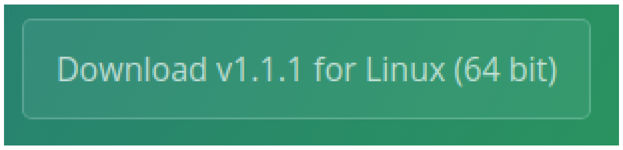
Linux

These instructions were developed on Ubuntu 20.04 – they should work on any modern Linux desktop system. However, they do require some comfort with the command line. If they don't work for you, or you are desparate for a point-and-click installer, please file a bug (or better, a patch or pull-request.)

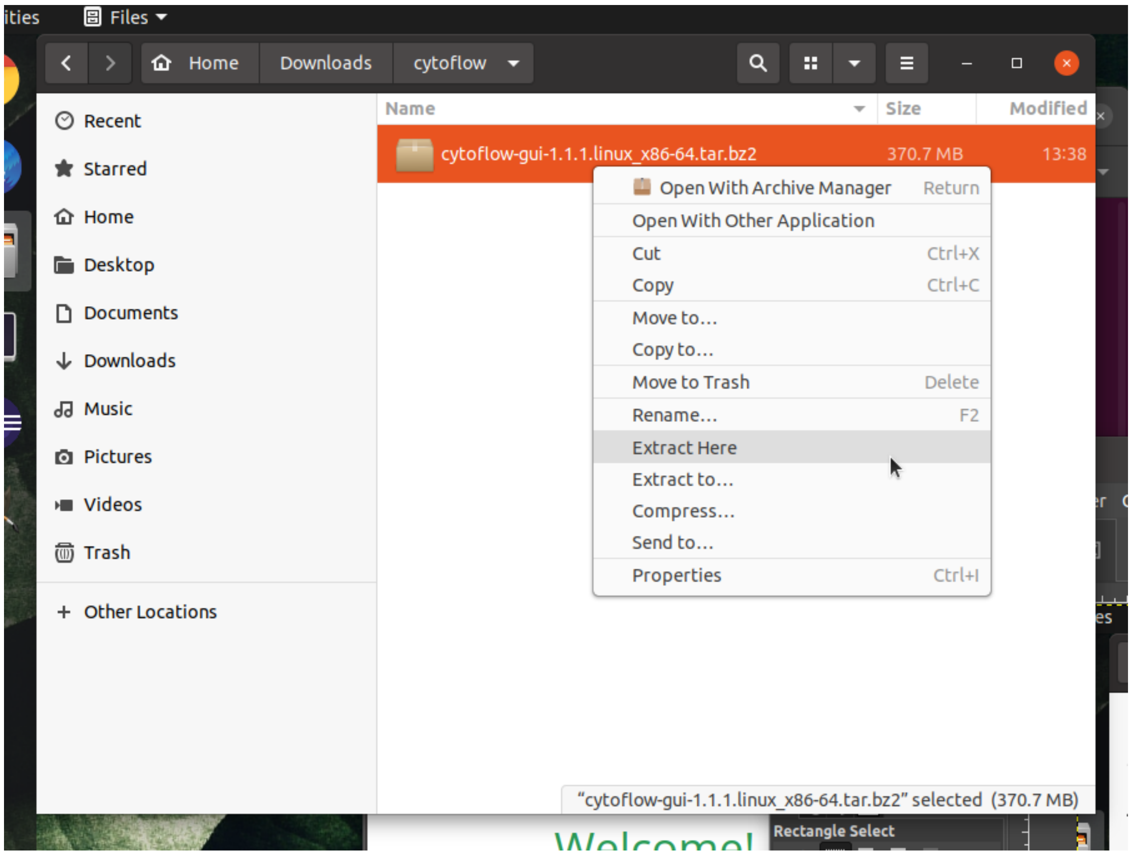
This results in a program that you can launch from your desktop launcher – the "Programs" menu or similar.

(a) Browse to https://cytoflow.github.io/

(b) Download the Linux binaries by clicking the button at the top of the page.



(c) Extract the archive:



(d) Move the resulting directory to a "permanent" home. I like to drop such things in `~/.local/lib`, but you may prefer to put it elsewhere.

The remaining steps should be completed from the command line, starting in the directory containing the extracted files.

If you would like to launch `Cytoflow` from the command line, you can do so by navigating to this directory and running the executable `cytoflow`.

(e) Update the location of the icon in the `.desktop` file by calling the `set_launcher_icon` script:

```
$ ./set_launcher_icon
```

(f) Link `cytoflow.desktop` into the `~/.local/share/applications` directory:

```
$ ln -s $PWD/cytoflow.desktop $HOME/.local/share/applications/cytoflow.desktop
```

(g) Update the database of desktop entries:

```
$ update-desktop-database ~/.local/share/applications
```

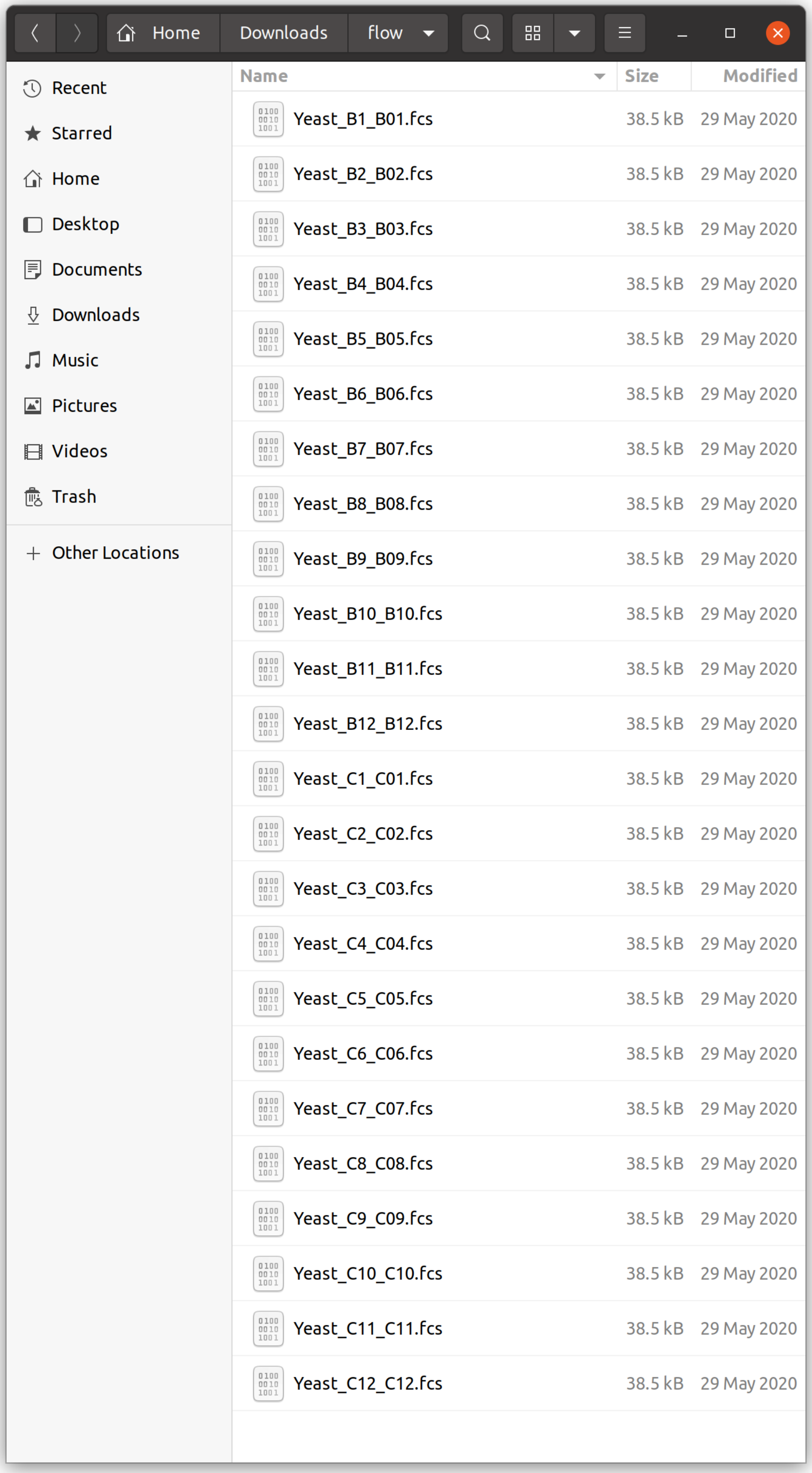
5.1.7 HOWTO: Import an experiment from a table

`Cytoflow` can allow you to analyze complex flow cytometry experiments, with many tubes and many conditions. However, describing those tubes and conditions in the `Experimental Setup` dialog can be a pain – especially if you already have that information in a table somewhere.

Unfortunately, the way I've programmed that dialog box, you can't (yet) copy-and-paste into it. Instead, `Cytoflow` allows you to *import* an experimental design from a CSV (comma-separated values) file, as long as it is formatted in the following way:

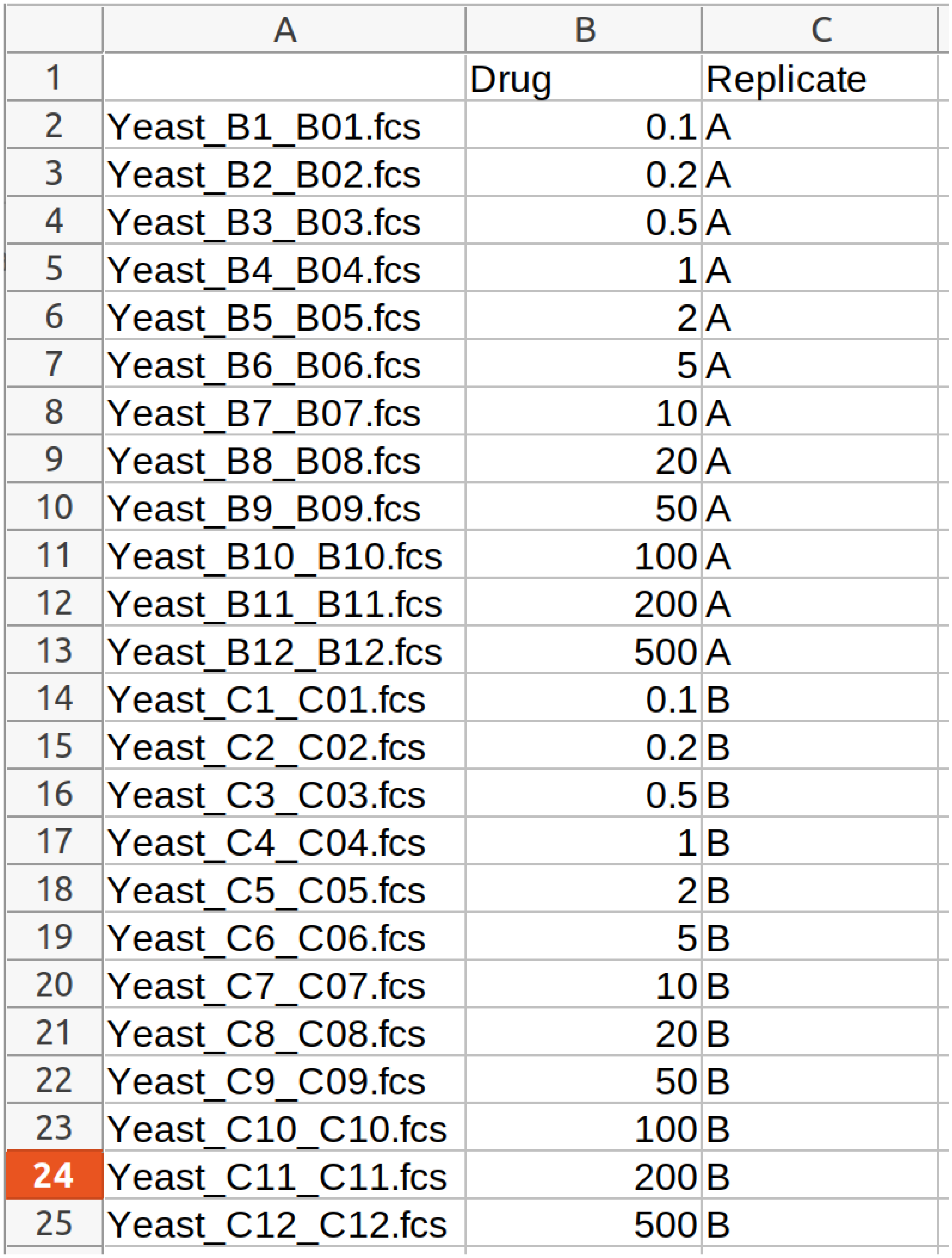
- The first row is a header; subsequent rows are tubes.
- The first column is the filename. (If you include paths, they must be either absolute paths, or relative to the location of the CSV file.)
- Each subsequent column is a variable. The entry in the header is the variable's name, and in subsequent rows the column contains the variable value for that tube.
- There is no way to specify the type of variable. You can change the type once it's been imported.

An example will make this clearer. Let's say I have the following 24 files:



Name	Size	Modified
Yeast_B1_B01.fcs	38.5 kB	29 May 2020
Yeast_B2_B02.fcs	38.5 kB	29 May 2020
Yeast_B3_B03.fcs	38.5 kB	29 May 2020
Yeast_B4_B04.fcs	38.5 kB	29 May 2020
Yeast_B5_B05.fcs	38.5 kB	29 May 2020
Yeast_B6_B06.fcs	38.5 kB	29 May 2020
Yeast_B7_B07.fcs	38.5 kB	29 May 2020
Yeast_B8_B08.fcs	38.5 kB	29 May 2020
Yeast_B9_B09.fcs	38.5 kB	29 May 2020
Yeast_B10_B10.fcs	38.5 kB	29 May 2020
Yeast_B11_B11.fcs	38.5 kB	29 May 2020
Yeast_B12_B12.fcs	38.5 kB	29 May 2020
Yeast_C1_C01.fcs	38.5 kB	29 May 2020
Yeast_C2_C02.fcs	38.5 kB	29 May 2020
Yeast_C3_C03.fcs	38.5 kB	29 May 2020
Yeast_C4_C04.fcs	38.5 kB	29 May 2020
Yeast_C5_C05.fcs	38.5 kB	29 May 2020
Yeast_C6_C06.fcs	38.5 kB	29 May 2020
Yeast_C7_C07.fcs	38.5 kB	29 May 2020
Yeast_C8_C08.fcs	38.5 kB	29 May 2020
Yeast_C9_C09.fcs	38.5 kB	29 May 2020
Yeast_C10_C10.fcs	38.5 kB	29 May 2020
Yeast_C11_C11.fcs	38.5 kB	29 May 2020
Yeast_C12_C12.fcs	38.5 kB	29 May 2020

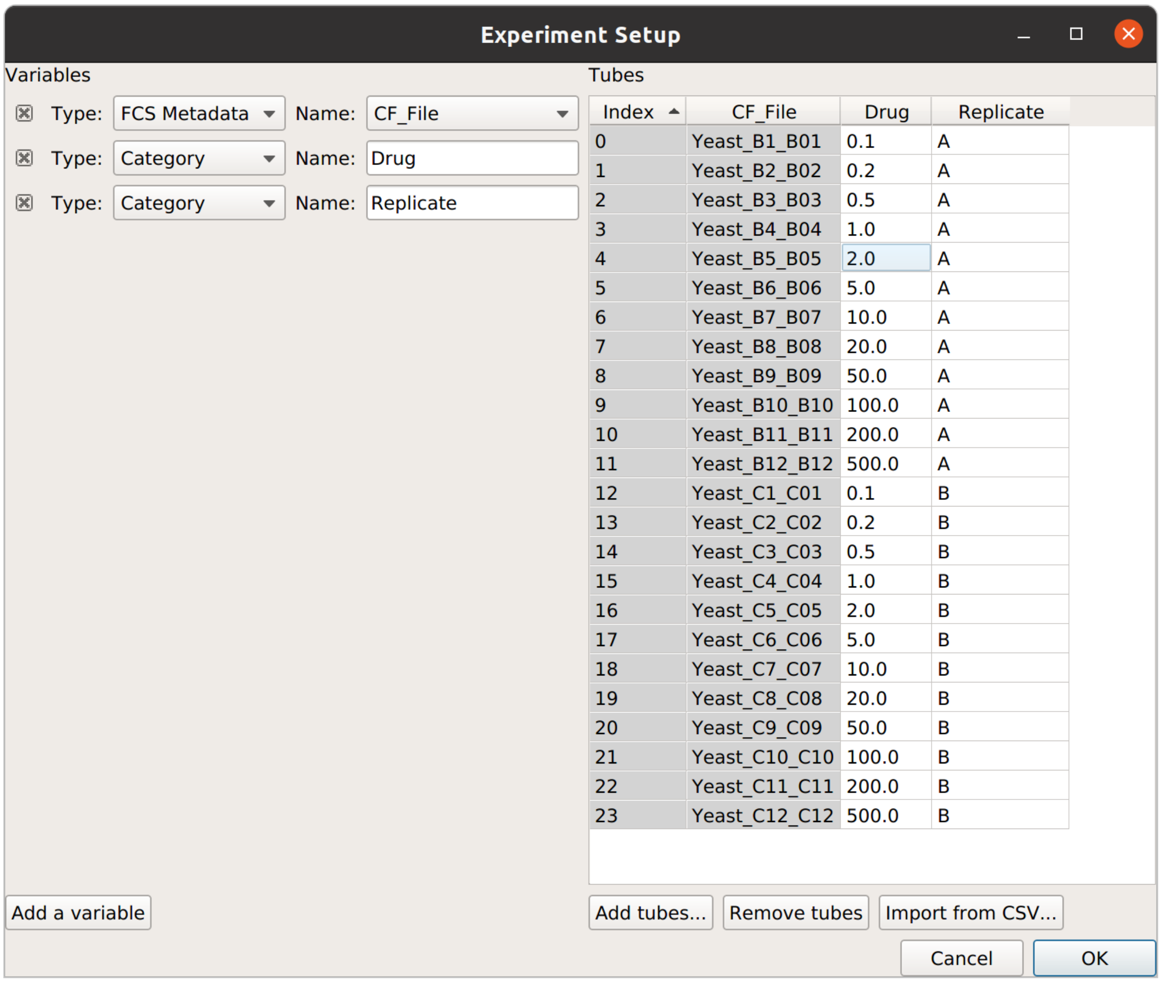
As is clear from the filenames, they came from two rows of a multiwell plate. Let's say that the two rows are replicates, and the amount of some drug I added increases across each row. I might open my favorite spreadsheet editor and create a table like so:



	A	B	C
1		Drug	Replicate
2	Yeast_B1_B01.fcs	0.1	A
3	Yeast_B2_B02.fcs	0.2	A
4	Yeast_B3_B03.fcs	0.5	A
5	Yeast_B4_B04.fcs	1	A
6	Yeast_B5_B05.fcs	2	A
7	Yeast_B6_B06.fcs	5	A
8	Yeast_B7_B07.fcs	10	A
9	Yeast_B8_B08.fcs	20	A
10	Yeast_B9_B09.fcs	50	A
11	Yeast_B10_B10.fcs	100	A
12	Yeast_B11_B11.fcs	200	A
13	Yeast_B12_B12.fcs	500	A
14	Yeast_C1_C01.fcs	0.1	B
15	Yeast_C2_C02.fcs	0.2	B
16	Yeast_C3_C03.fcs	0.5	B
17	Yeast_C4_C04.fcs	1	B
18	Yeast_C5_C05.fcs	2	B
19	Yeast_C6_C06.fcs	5	B
20	Yeast_C7_C07.fcs	10	B
21	Yeast_C8_C08.fcs	20	B
22	Yeast_C9_C09.fcs	50	B
23	Yeast_C10_C10.fcs	100	B
24	Yeast_C11_C11.fcs	200	B
25	Yeast_C12_C12.fcs	500	B

I'll save this as a CSV file *in the same directory as my .FCS files*. When I import this file in the `Experimental`

`Setup` dialog, it looks like this:



	Type	Name
☒	FCS Metadata	CF_File
☒	Category	Drug
☒	Category	Replicate

Index	CF_File	Drug	Replicate
0	Yeast_B1_B01	0.1	A
1	Yeast_B2_B02	0.2	A
2	Yeast_B3_B03	0.5	A
3	Yeast_B4_B04	1.0	A
4	Yeast_B5_B05	2.0	A
5	Yeast_B6_B06	5.0	A
6	Yeast_B7_B07	10.0	A
7	Yeast_B8_B08	20.0	A
8	Yeast_B9_B09	50.0	A
9	Yeast_B10_B10	100.0	A
10	Yeast_B11_B11	200.0	A
11	Yeast_B12_B12	500.0	A
12	Yeast_C1_C01	0.1	B
13	Yeast_C2_C02	0.2	B
14	Yeast_C3_C03	0.5	B
15	Yeast_C4_C04	1.0	B
16	Yeast_C5_C05	2.0	B
17	Yeast_C6_C06	5.0	B
18	Yeast_C7_C07	10.0	B
19	Yeast_C8_C08	20.0	B
20	Yeast_C9_C09	50.0	B
21	Yeast_C10_C10	100.0	B
22	Yeast_C11_C11	200.0	B
23	Yeast_C12_C12	500.0	B

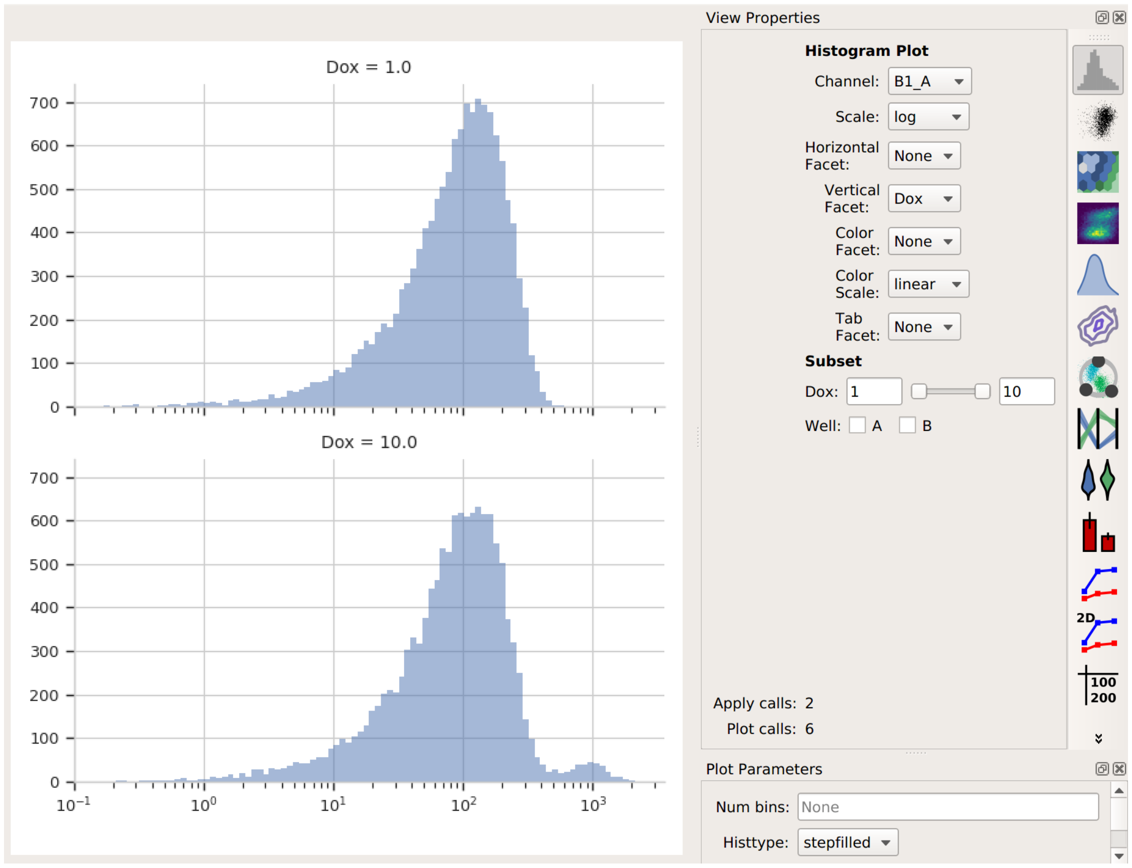
It is important to note that, by default, everything is imported as a **Category** variable. In this case, we obviously want `Drug` to be numeric, so I can pull down the "Type" selector for that variable and change it to "Numeric". Similarly, if your table has TRUE and FALSE for values, "TRUE" and "FALSE" will be imported as categories, but you can change it to a True/False variable by changing the variable type.

5.1.8 HOWTO: Export plots

I'm really proud of the plots you can produce with `Cytoflow`, and I hope that you will find them nice enough to use in your talks, posters and publications! To help you do so, it's easy to export the current plot to a variety of file formats.

Procedure

(a) Create the plot you want to save. Here, I've got a set of histograms.



View Properties

Histogram Plot

Channel: B1_A

Scale: log

Horizontal Facet: None

Vertical Facet: Dox

Color Facet: None

Color Scale: linear

Tab Facet: None

Subset

Dox: 1 10

Well: A B

Apply calls: 2

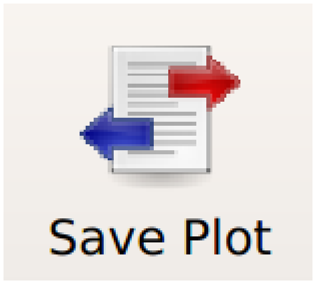
Plot calls: 6

Plot Parameters

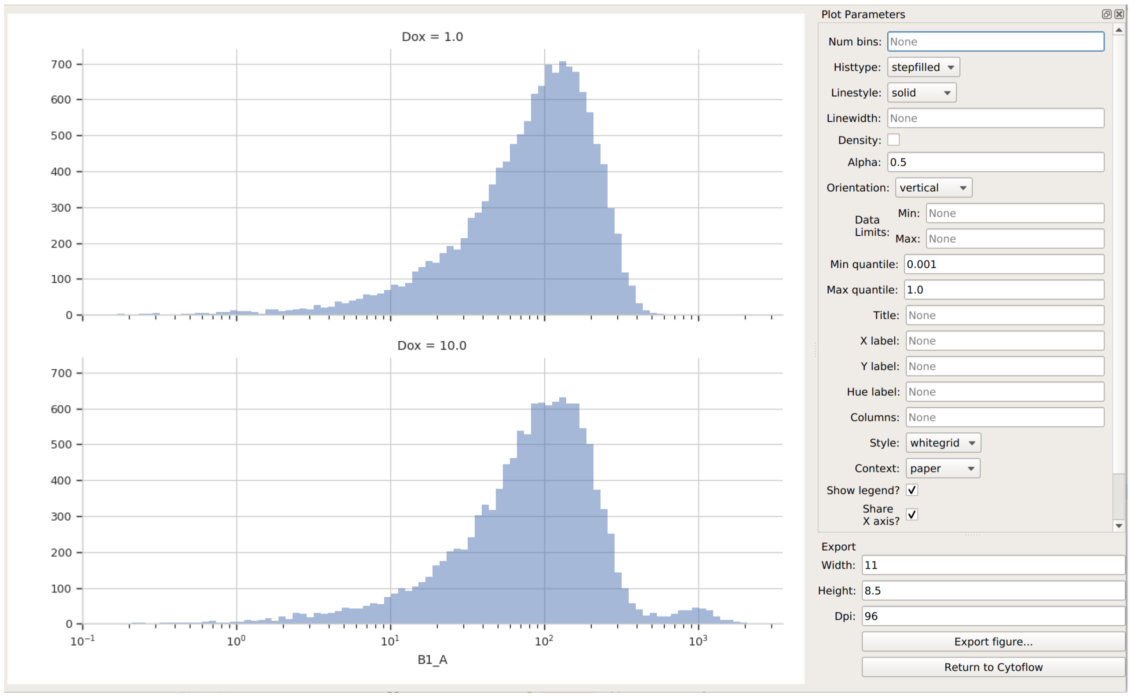
Num bins: None

Histtype: stepfilled

(b) Click the **Save Plot** button in the top toolbar:

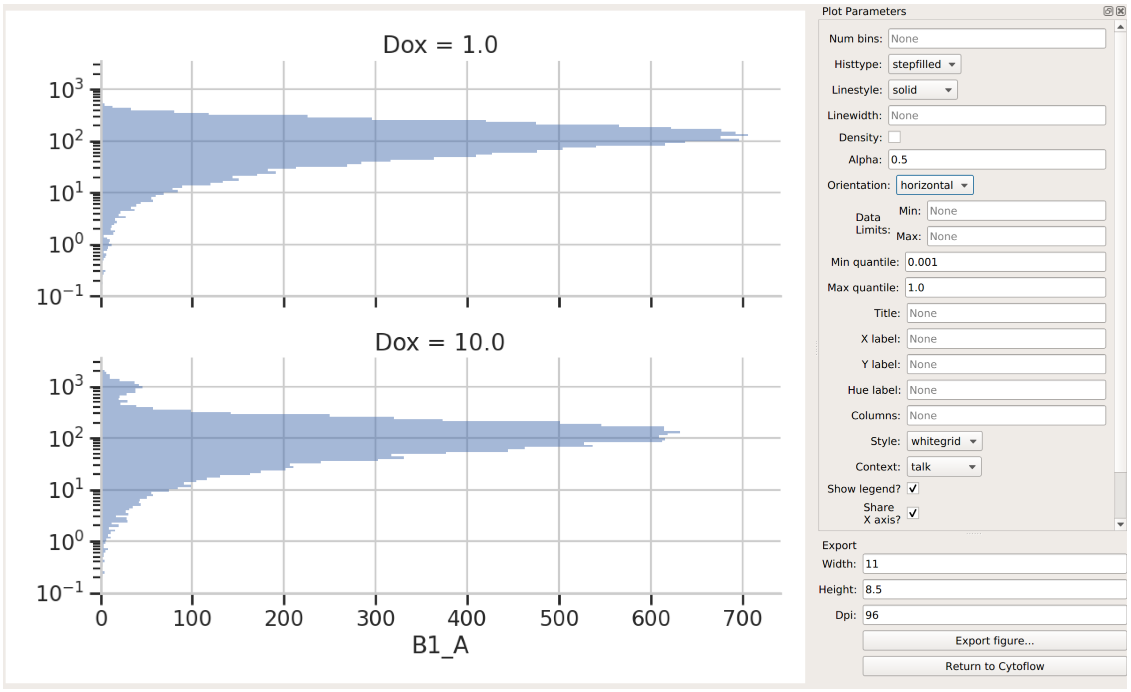


(c) The **Save Plot** view will open. In the center panel, you'll see the same plot – but on the right, you'll see a pane with a bunch of visual style options you can adjust. (The eagle-eyed among you will recognize this as the same panel that shows up below the view parameters pane in the main application.)



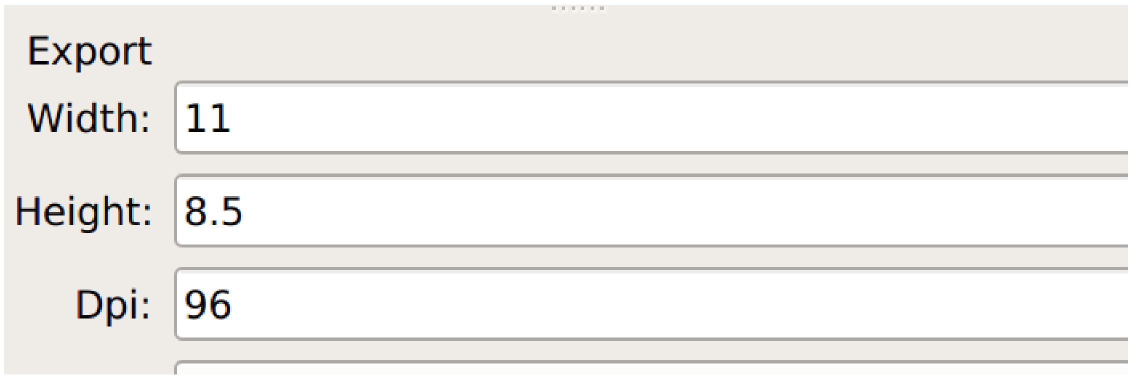
Num bins: None
Histtype: stepfilled
Linestyle: solid
Linewidth: None
Density:
Alpha: 0.5
Orientation: vertical
Data Limits: Min: None Max: None
Min quantile: 0.001
Max quantile: 1.0
Title: None
X label: None
Y label: None
Hue label: None
Columns: None
Style: whitegrid
Context: paper
Show legend? ✓
Share X axis? ✓
Export
Width: 11
Height: 8.5
Dpi: 96

(d) Adjust the display options to your liking. One particular one to pay attention to is the **Context** setting, which changes the relative size of the annotations (axes, labels, legend, etc) relative to the main plot. Here, I've changed it to *talk* and set the plot to a horizontal layout instead of a vertical one.



Num bins: None
Histtype: stepfilled
Linestyle: solid
Linewidth: None
Density:
Alpha: 0.5
Orientation: horizontal
Data Limits: Min: None Max: None
Min quantile: 0.001
Max quantile: 1.0
Title: None
X label: None
Y label: None
Hue label: None
Columns: None
Style: whitegrid
Context: talk
Show legend? ✓
Share X axis? ✓
Export
Width: 11
Height: 8.5
Dpi: 96

(e) Set the *width*, *height*, and *resolution* of the final image. (Width and height are in inches, and resolution is in DPI – dots per inch.)



Export

Width:	11
Height:	8.5
Dpi:	96

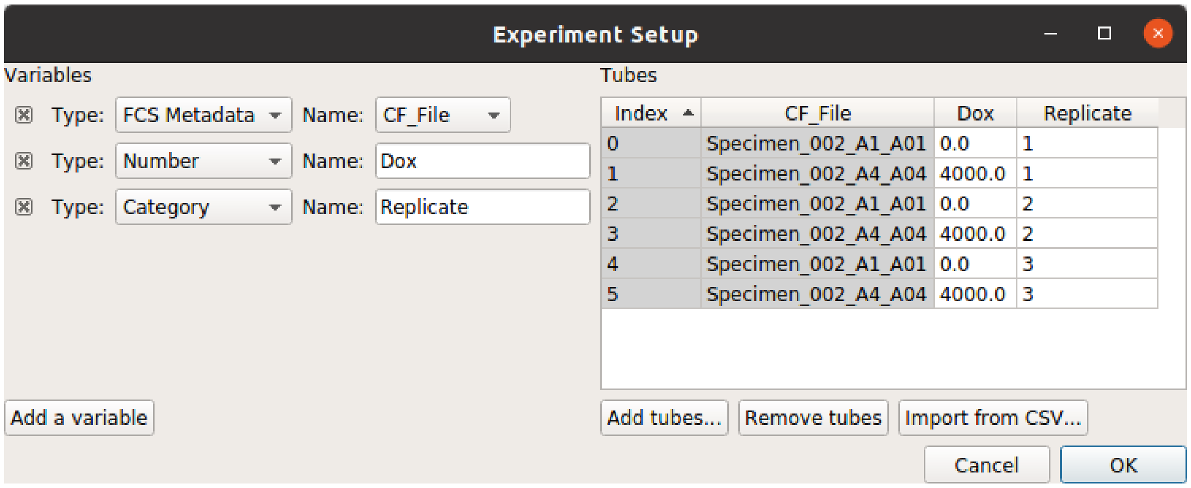
(f) Click **Export figure. . .** to open your system's usual **Save As** dialog box. Pay *particular attention* to the file *type* – it's here where you can choose to save as, say, a PNG or a PDF or a TIFF. Which formats are available to you are system-dependent, but if you need a raster image (to put in a talk, for example), I suggest PNG, while if you need a vector image (to submit with a manuscript, for example), I suggest PDF or EPS.

(g) Once you're done, click **Return to Cytoflow** to return to the main application.

5.1.9 HOWTO: Add error bars to a statistics plot

While statistics plots of things like the *geometric mean* are useful, we often want to add some sort of "error bars" to that plot as well.

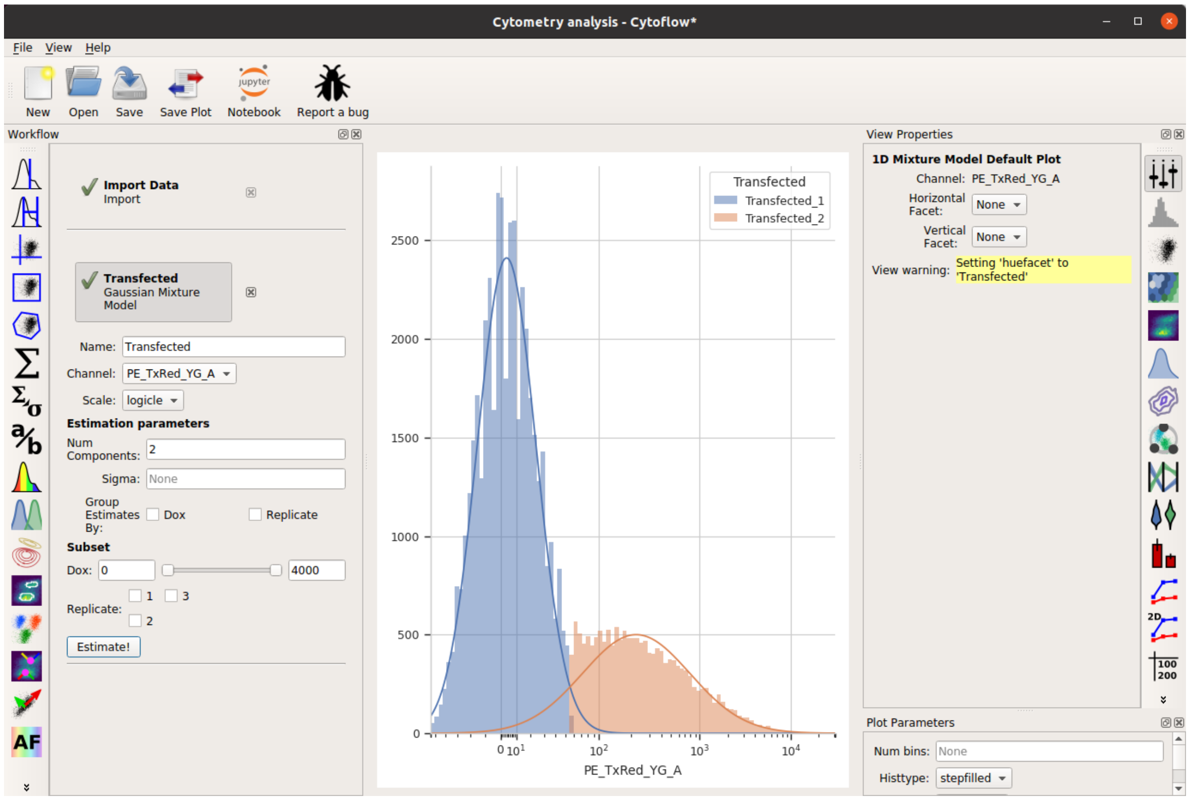
For example, consider the following experiment (taken from the *Kiani et al example*), in which we have two treatment levels that we've measured in triplicate:



Type	Name
FCS Metadata	CF_File
Number	Dox
Category	Replicate

Index	CF_File	Dox	Replicate
0	Specimen_002_A1_A01	0.0	1
1	Specimen_002_A4_A04	4000.0	1
2	Specimen_002_A1_A01	0.0	2
3	Specimen_002_A4_A04	4000.0	2
4	Specimen_002_A1_A01	0.0	3
5	Specimen_002_A4_A04	4000.0	3

Note that, as per the example, I'm only looking at cells that are already fluorescing in the **PE_TxRed_YG** channel, which is my transfection marker.



Import Data
Import

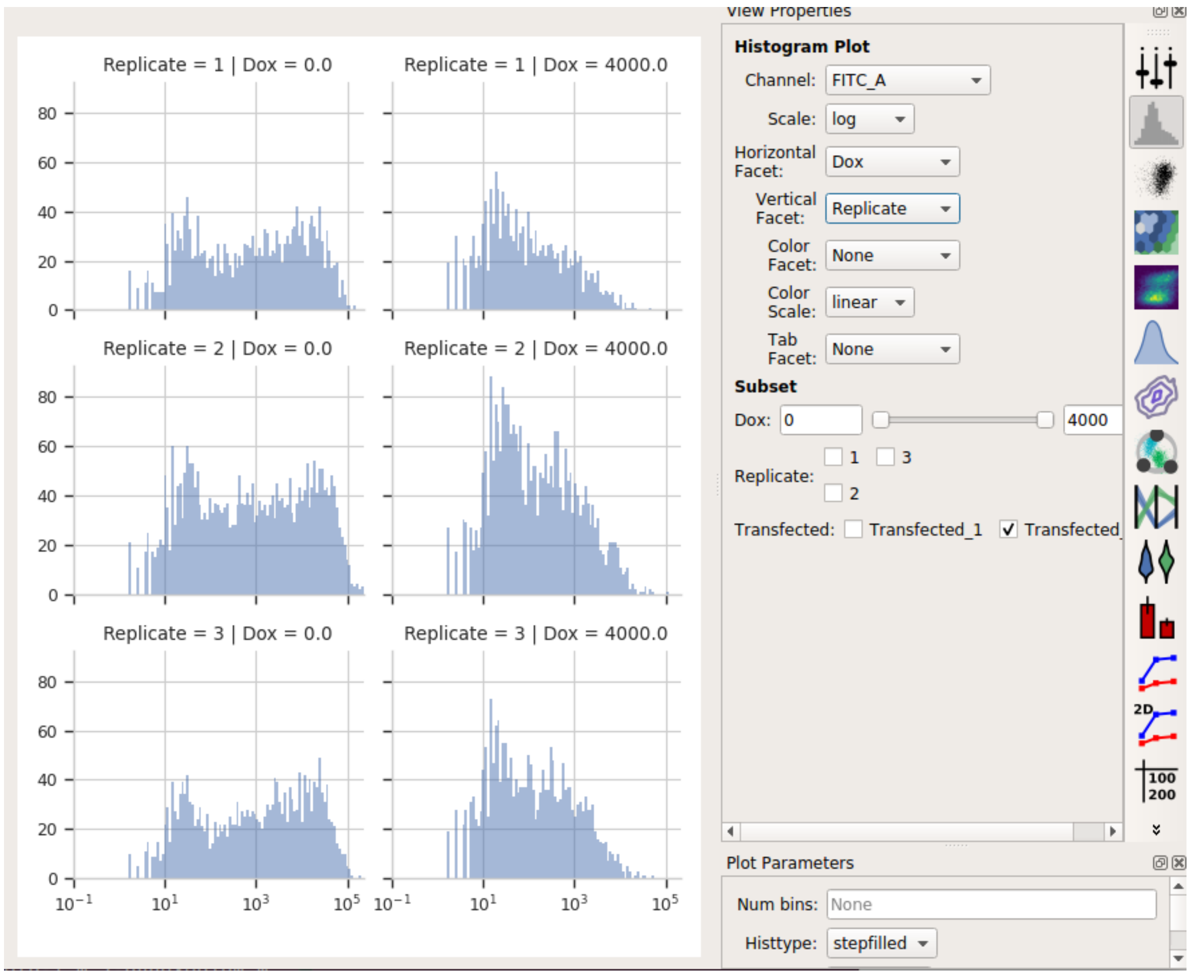
Transfected
Gaussian Mixture Model

Name: Transfected
Channel: PE_TxRed_YG_A
Scale: logicle
Estimation parameters
Num Components: 2
Sigma: None
Group Estimates By: Dox Replicate
Subset
Dox: 0 4000
Replicate: 1 3 2

1D Mixture Model Default Plot
Channel: PE_TxRed_YG_A
Horizontal Facet: None
Vertical Facet: None
View warning: Setting 'huefacet' to 'Transfected'

Num bins: None
Histtype: stepfilled

Let's look at how the **FITC_A** channel changes across my conditions and replicates:

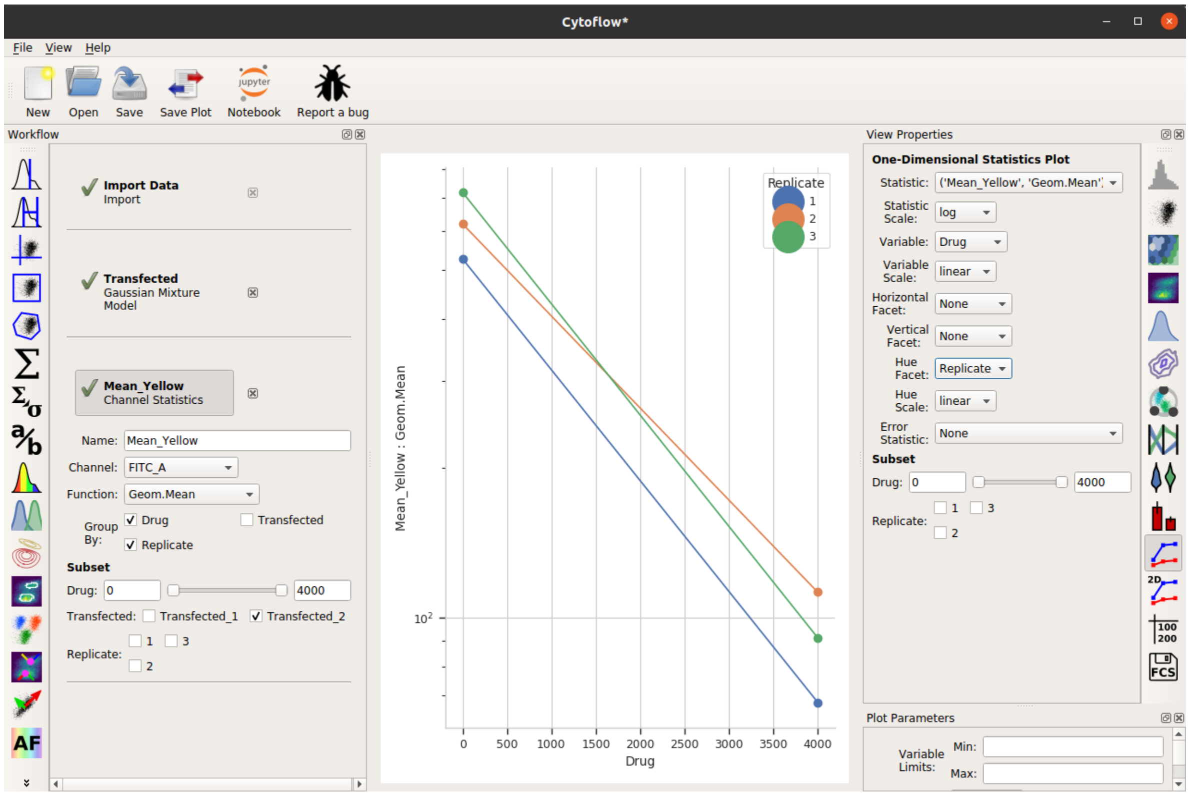

Replicate = 1 | Dox = 0.0
Replicate = 1 | Dox = 4000.0
Replicate = 2 | Dox = 0.0
Replicate = 2 | Dox = 4000.0
Replicate = 3 | Dox = 0.0
Replicate = 3 | Dox = 4000.0
View Properties
Plot Parameters

Histogram Plot
Channel: FITC_A
Scale: log
Horizontal Facet: Dox
Vertical Facet: Replicate
Color Facet: None
Color Scale: linear
Tab Facet: None

Subset
Dox: 0 – 4000
Replicate: ☐ 1 ☐ 3 ☐ 2
Transfected: ☐ Transfected_1 ☑ Transfected_

Num bins: None
Histtype: stepfilled

It's clear from the histograms that the **Drug == 0** populations have a much higher fluorescence in the **FITC_A** channel than the **Drug = 4000.0** condition. We can see that quantitatively by creating a **Geom.Mean** statistic:



Import Data
Import

Transfected
Gaussian Mixture Model

Mean_Yellow
Channel Statistics

Name: Mean_Yellow
Channel: FITC_A
Function: Geom.Mean
Group By: ✓ Drug ☐ Transfected ✓ Replicate
Subset
Drug: 0 4000
Transfected: ☐ Transfected_1 ✓ Transfected_2
Replicate: ☐ 1 ☐ 3 ☐ 2

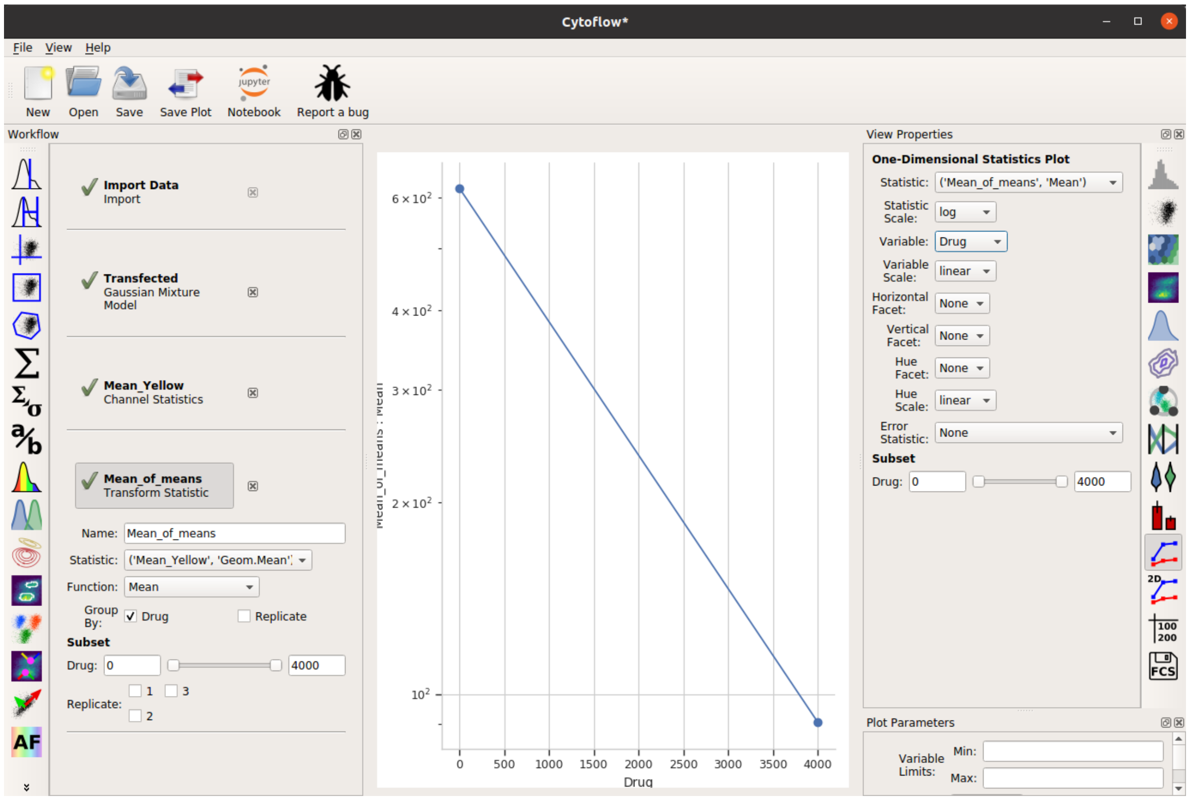
One-Dimensional Statistics Plot
Statistic: ('Mean_Yellow', 'Geom.Mean')
Statistic Scale: log
Variable: Drug
Variable Scale: linear
Horizontal Facet: None
Vertical Facet: None
Hue Facet: Replicate
Hue Scale: linear
Error Statistic: None
Subset
Drug: 0 4000
Replicate: ☐ 1 ☐ 3 ☐ 2

Variable Limits: Min: Max:

When we compute the geometric mean of all six different subsets (two drugs and three replicates), there's a definite decrease. Our goal is to also show a visual representation of the amount of variation between the three replicates.

Before that, though, there's a subtle question to answer – do we want to take a "mean of means" – that is, the geometric mean of these three means? Or do we want a geometric mean of *all* of the underlying data? I'm going to choose the first approach, but I encourage you to think carefully about the which is more appropriate in your own case.

So I'll add a **Transform Statistic** operation, to take the mean-of-geometric-means:



Import Data
Import

Transfected
Gaussian Mixture Model

Mean_Yellow
Channel Statistics

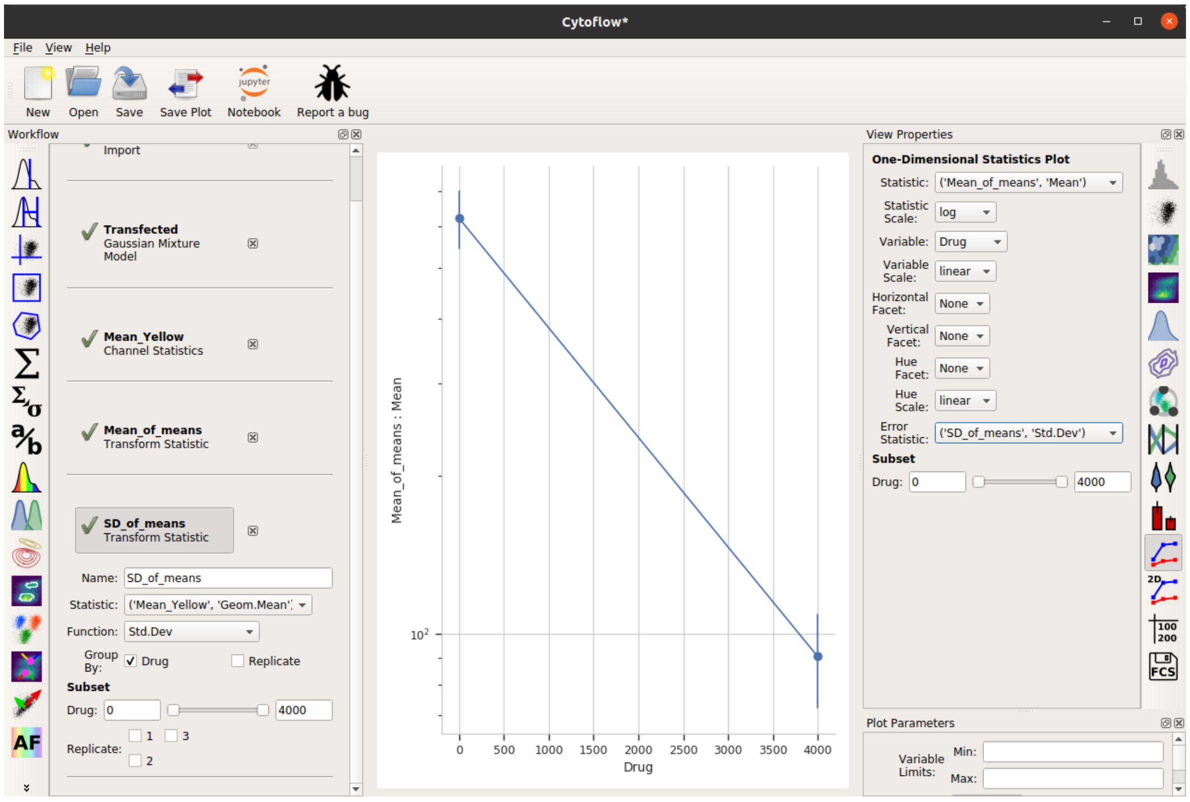
Mean_of_means
Transform Statistic

Name: Mean_of_means
Statistic: ('Mean_Yellow', 'Geom.Mean')
Function: Mean
Group By: ☑ Drug ☐ Replicate
Subset
Drug: 0 4000
Replicate: ☐ 1 ☐ 3 ☐ 2

One-Dimensional Statistics Plot
Statistic: ('Mean_of_means', 'Mean')
Statistic Scale: log
Variable: Drug
Variable Scale: linear
Horizontal Facet: None
Vertical Facet: None
Hue Facet: None
Hue Scale: linear
Error Statistic: None
Subset
Drug: 0 4000

Variable Limits: Min: Max:

And now, to add error bars, I need *another* statistic. This will also be a **Transform Statistic** operation, but this time to compute the standard deviation of the means:



Import

Transfected
Gaussian Mixture Model

Mean_Yellow
Channel Statistics

Mean_of_means
Transform Statistic

SD_of_means
Transform Statistic

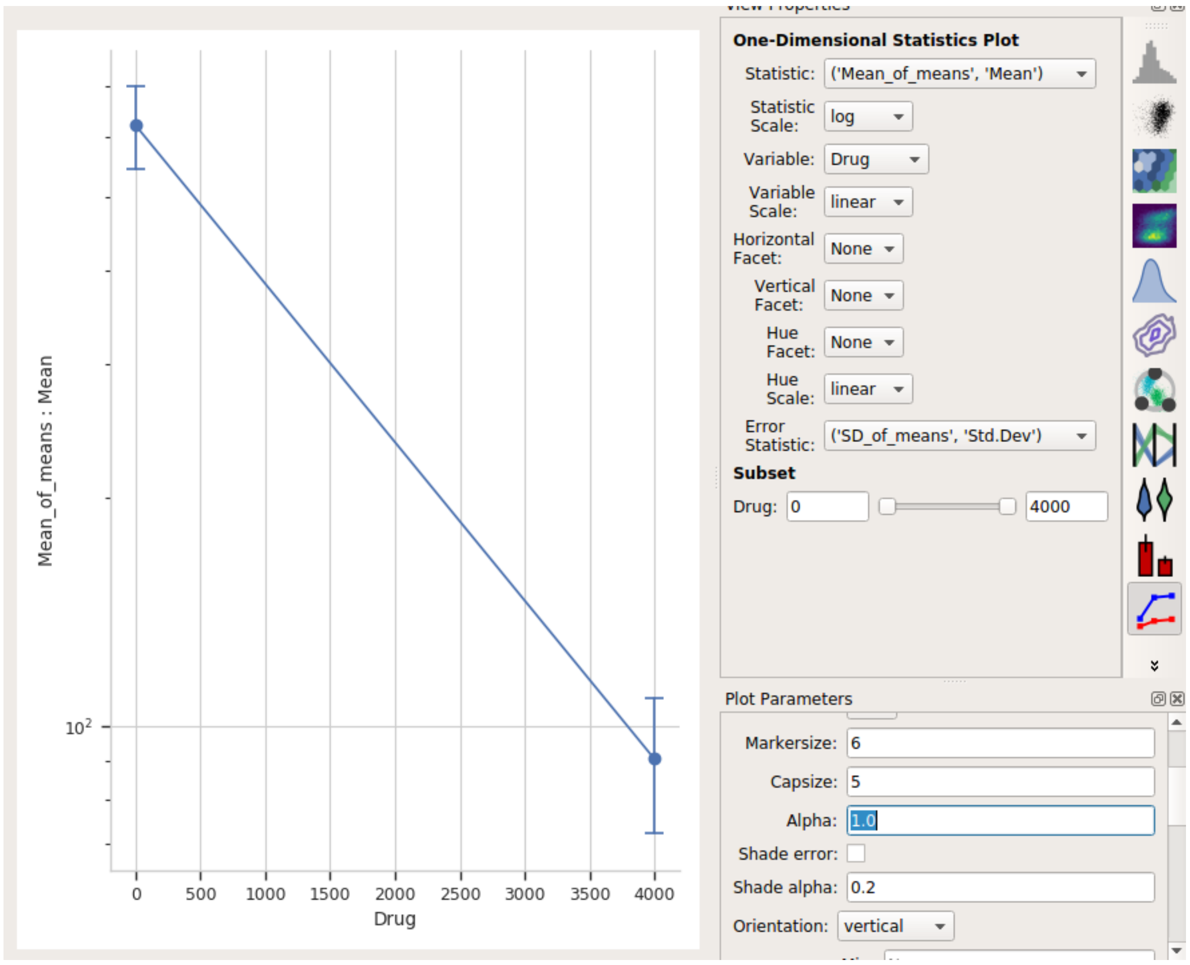
Name: SD_of_means
Statistic: ('Mean_Yellow', 'Geom.Mean')
Function: Std.Dev
Group By: ☑ Drug ☐ Replicate
Subset
Drug: 0 4000
Replicate: ☐ 1 ☐ 3 ☐ 2

One-Dimensional Statistics Plot
Statistic: ('Mean_of_means', 'Mean')
Statistic Scale: log
Variable: Drug
Variable Scale: linear
Horizontal Facet: None
Vertical Facet: None
Hue Facet: None
Hue Scale: linear
Error Statistic: ('SD_of_means', 'Std.Dev')
Subset
Drug: 0 4000

Variable Limits: Min: Max:

Two important things to note about the above image. First, I have used the same **Group by:** settings as the **Mean_of_means** operation. And second, when I set up my **One-dimensional statistics plot**, I chose the new statistic as the **Error statistic**.

One last thing – the "default" visual properties of those error bars are just vertical lines. This is useful if you've got a lot to visualize, but less so if it's just a few points. Many people like "end caps" on their error bars – to get those, change the **Capsize** option in the **Plot Parameters** pane to something greater than 0.



One-Dimensional Statistics Plot

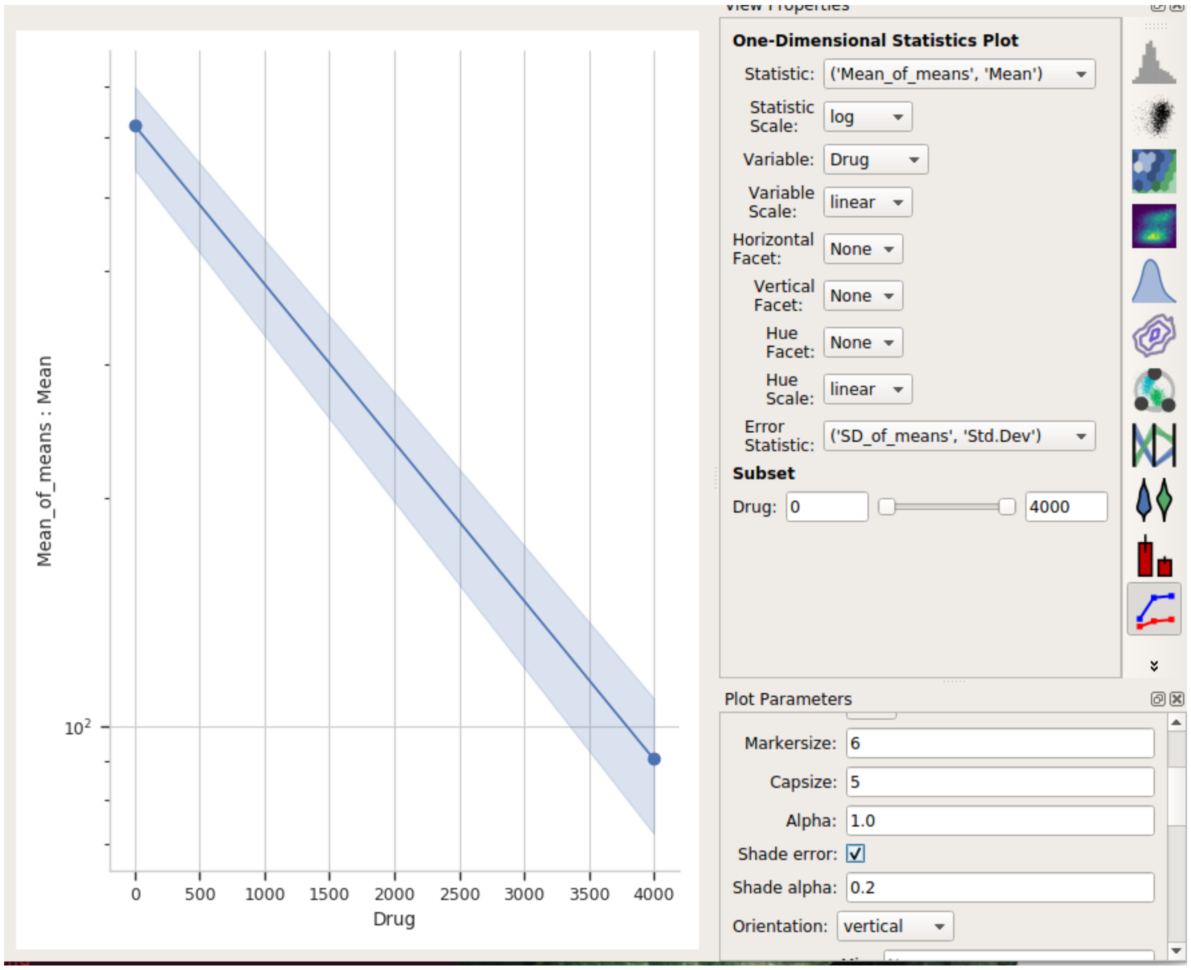
Statistic: ('Mean_of_means', 'Mean')
Statistic Scale: log
Variable: Drug
Variable Scale: linear
Horizontal Facet: None
Vertical Facet: None
Hue Facet: None
Hue Scale: linear
Error Statistic: ('SD_of_means', 'Std.Dev')

Subset

Drug: 0 – 4000

Markersize: 6
Capsize: 5
Alpha: 1.0
Shade error: ☐
Shade alpha: 0.2
Orientation: vertical

Alternately, choose **Shade error** to get a "shaded" error display. This one is particularly nice if you have lots of error bars.



One-Dimensional Statistics Plot

Statistic: ('Mean_of_means', 'Mean')

Statistic Scale: log

Variable: Drug

Variable Scale: linear

Horizontal Facet: None

Vertical Facet: None

Hue Facet: None

Hue Scale: linear

Error Statistic: ('SD_of_means', 'Std.Dev')

Subset

Drug: 0 4000

Markersize: 6

Capsize: 5

Alpha: 1.0

Shade error: ☑

Shade alpha: 0.2

Orientation: vertical

One final thing – the use of error bars is a subtle topic – much more so than most biologists grasp. For a useful overview of the issues at play, please see:

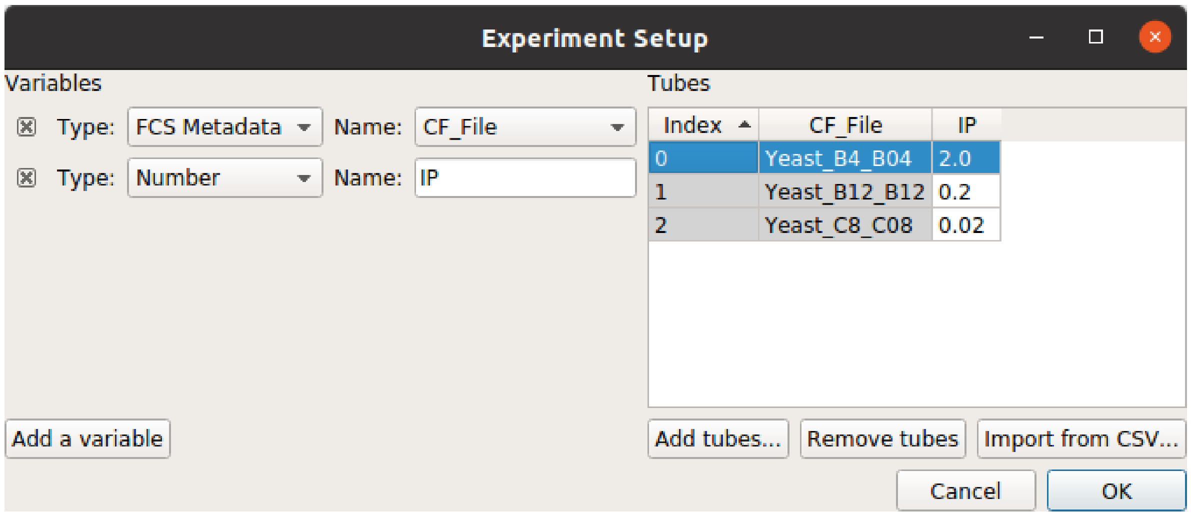
Know when your numbers are significant

and

Error bars in experimental biology

5.1.10 HOWTO: Use different gates for different subsets of data

Sometimes, it makes sense to apply different gates to differen subsets of your data set. It's particularly useful when we're estimating the gate parameters from the data. For example, consider the following data (taken from the **Yeast Dosage-Response Curve** tutorial.)



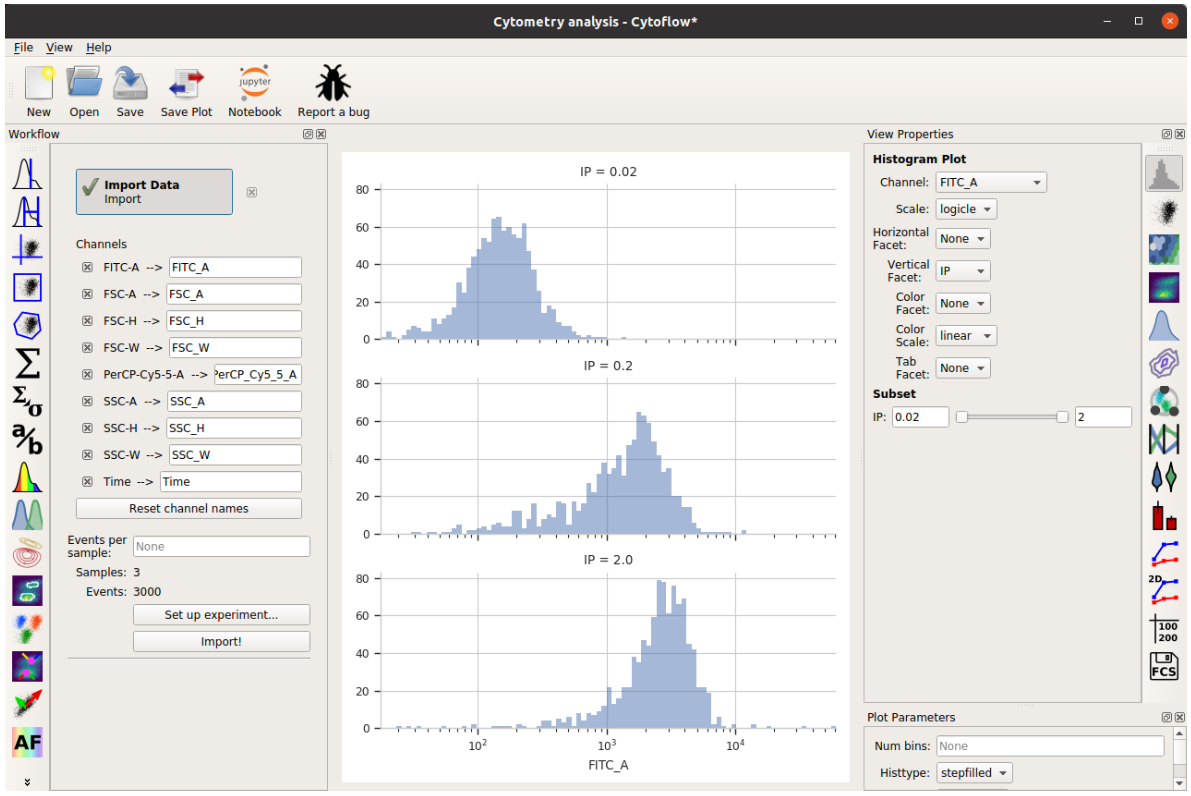
Variables

Type: FCS Metadata Name: CF_File

Type: Number Name: IP

Tubes

Index	CF_File	IP
0	Yeast_B4_B04	2.0
1	Yeast_B12_B12	0.2
2	Yeast_C8_C08	0.02



Channels

FITC-A --> FITC_A
FSC-A --> FSC_A
FSC-H --> FSC_H
FSC-W --> FSC_W
PerCP-Cy5-5-A --> PerCP_Cy5_5_A
SSC-A --> SSC_A
SSC-H --> SSC_H
SSC-W --> SSC_W
Time --> Time

Events per sample: None
Samples: 3
Events: 3000

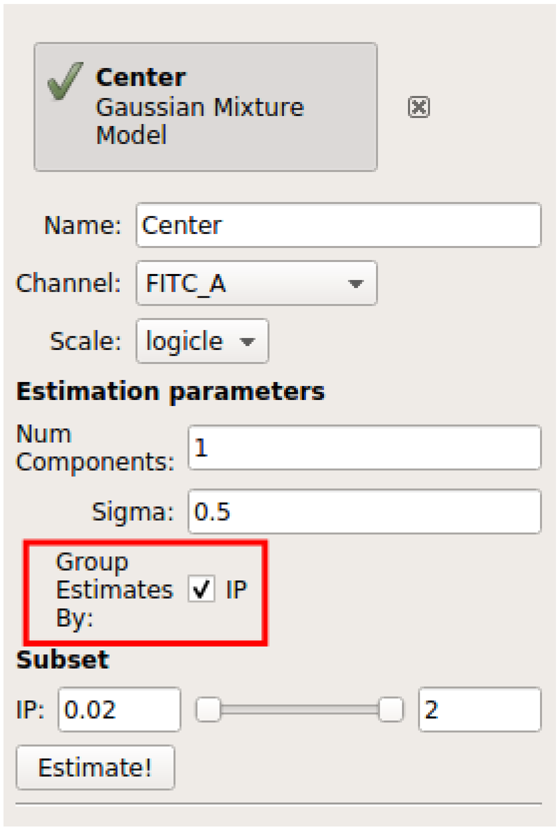
Histogram Plot
Channel: FITC_A
Scale: logicle
Horizontal Facet: None
Vertical Facet: IP
Color Facet: None
Color Scale: linear
Tab Facet: None

Subset
IP: 0.02 – 2

Num bins: None
Histtype: stepfilled

Imagine that we want to use the *1D Gaussian operation* to choose the events from the center +/- 0.5 standard deviation this data. It's clear that a gaussian model that fits the first data set won't fit the second or third!

To address this problem, most data-driven gates have a **Group estimates by** option. Set this to a condition, and the operation will subdivide the data by that condition and estimate *separate model parameters for each subset.* For example, if we choose **IP** as the group-by condition:



Center
Gaussian Mixture Model

Name: Center
Channel: FITC_A
Scale: logicle

Estimation parameters

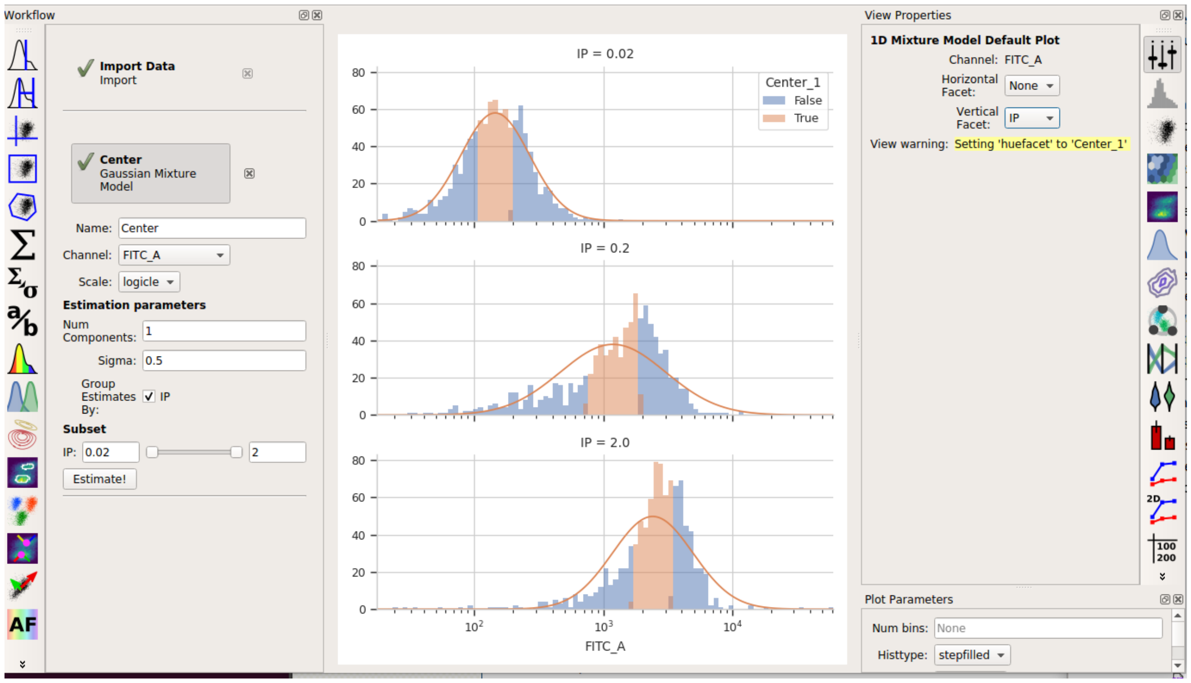
Num Components: 1
Sigma: 0.5
Group Estimates By: ✓ IP

Subset

IP: 0.02 2

Estimate!

Then the operation will give us three different estimates, one for each value of **IP**:



Workflow

Import Data
Import

Center
Gaussian Mixture Model

Name: Center
Channel: FITC_A
Scale: logicle

Estimation parameters

Num Components: 1
Sigma: 0.5
Group Estimates By: ✓ IP

Subset

IP: 0.02 2

Estimate!

View Properties

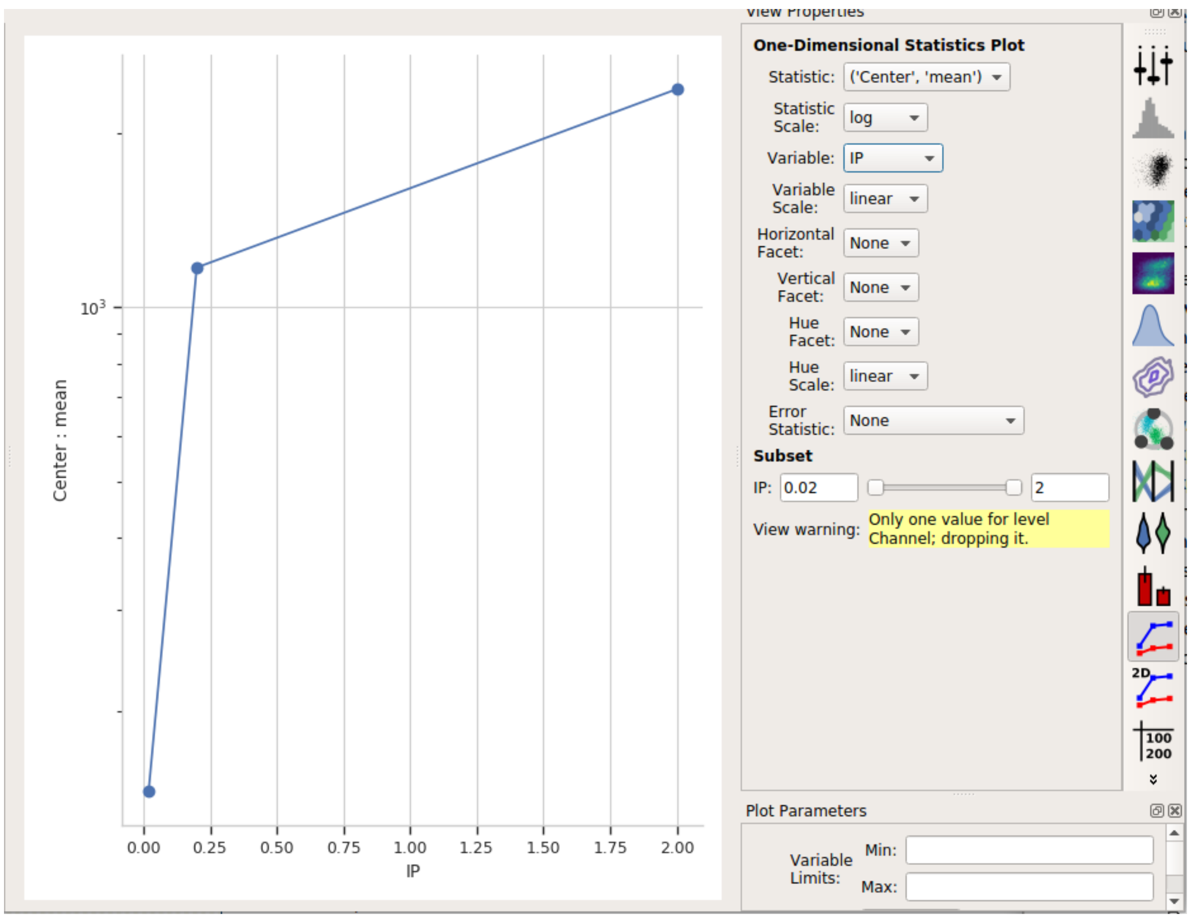
1D Mixture Model Default Plot

Channel: FITC_A
Horizontal Facet: None
Vertical Facet: IP
View warning: Setting 'huefacet' to 'Center_1'

Plot Parameters

Num bins: None
Histtype: stepfilled

And of course, don't forget that the operation also makes a new **statistic** containing the means:


View Properties

One-Dimensional Statistics Plot

Statistic: ('Center', 'mean')
Statistic Scale: log
Variable: IP
Variable Scale: linear
Horizontal Facet: None
Vertical Facet: None
Hue Facet: None
Hue Scale: linear
Error Statistic: None

Subset

IP: 0.02 2

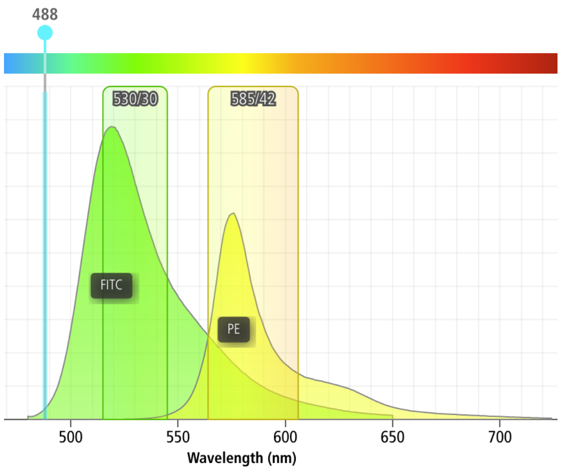
View warning: Only one value for level Channel; dropping it.

Plot Parameters

Variable Limits: Min: Max:

5.1.11 HOWTO: Compensate for bleedthrough

One common issue in flow cytometry is the fact that spectrally adjacent channels often overlap. For example, if I'm trying to measure a green fluorophore like FITC, and a yellow fluorophore like PE, a significant amount of FITC fluorescence will also be picked up by my PE channel, as demonstrated by the screenshot below (from the BD Spectrum Viewer)



As you can see, something like 12% of the FITC fluorescence ends up in the PE channel!

Fortunately, a little linear algebra can fix this problem, and `Cytoflow` makes it easy. However, you'll need to run a few controls:

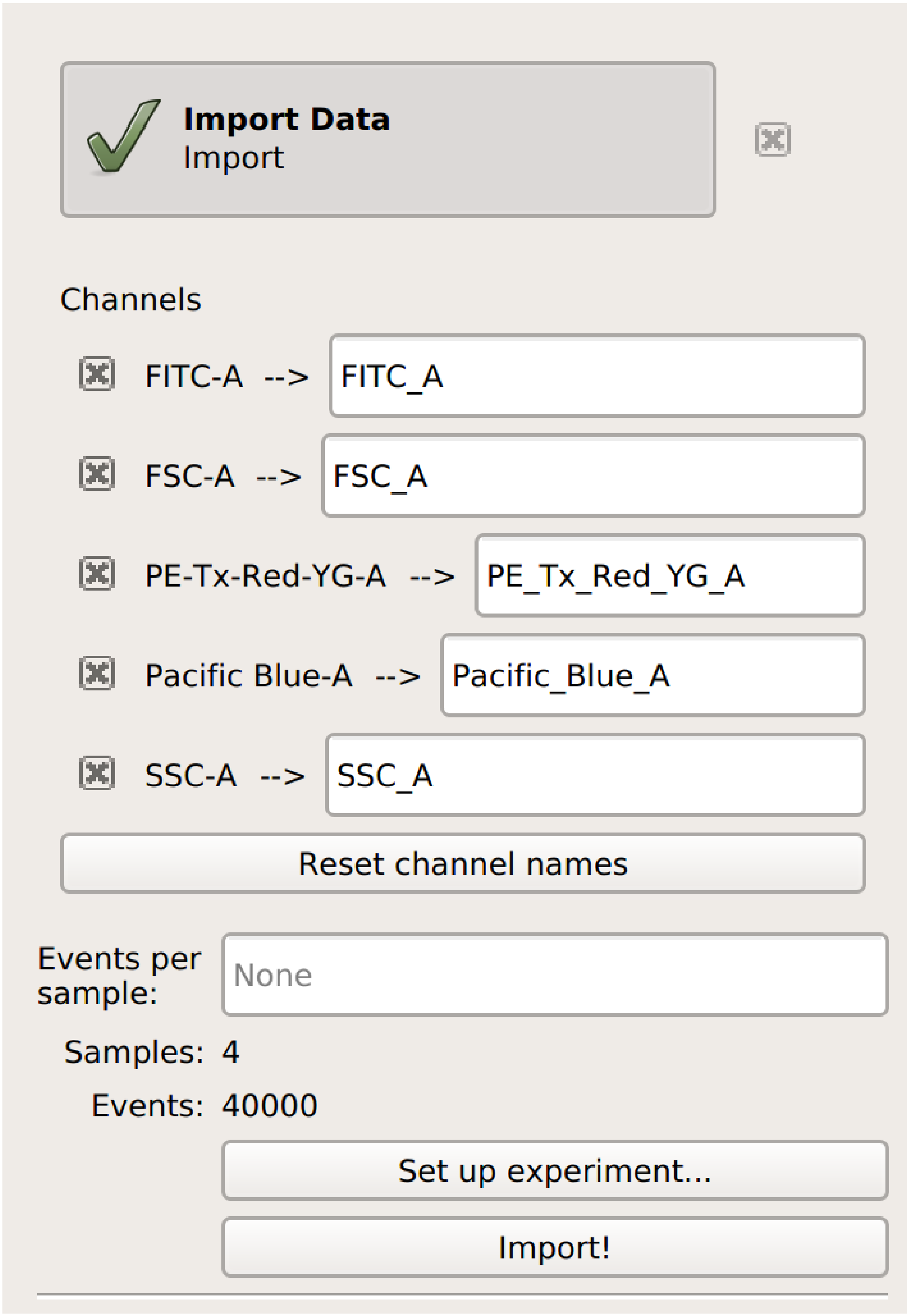
- A *blank* control – one with your cells but without any fluorophores. This will let us measure the "background" fluorescence (or autofluorescence) of the samples.
- A set of *single-color* controls – for each fluorophore, one control that is stained with (or expresses) only that fluorophore and no others. These let us measure how much signal "bleeds through" into the non-target channels. These controls should be as bright as (but no brigher than) your brightest experimental sample.

Warning: These controls must be collected using the SAME instrument settings as your experimental samples. It's really best if they're collected at the SAME TIME as your experimental samples – even properly calibrated instruments are known to drift substantially between days, or even over the course of a single day. And yes, that means you *really should* run these controls for *every* experiment. If you'd like a way to correct for day-to-day variability, see *HOWTO: Use beads to correct for day-to-day variation*.

Procedure

(a) Collect the controls listed above.

(b) Import your data into `Cytoflow`. Do *not* import your control samples (unless they're part of the experiment.) In the example below, we'll have three fluorescence channels – *Pacific Blue-A*, *FITC-A* and *PE-Tx-Red-YG-A* – in addition to the forward and side-scatter channels.



Channels

FITC-A	-->	FITC_A
FSC-A	-->	FSC_A
PE-Tx-Red-YG-A	-->	PE_Tx_Red_YG_A
Pacific Blue-A	-->	Pacific_Blue_A
SSC-A	-->	SSC_A

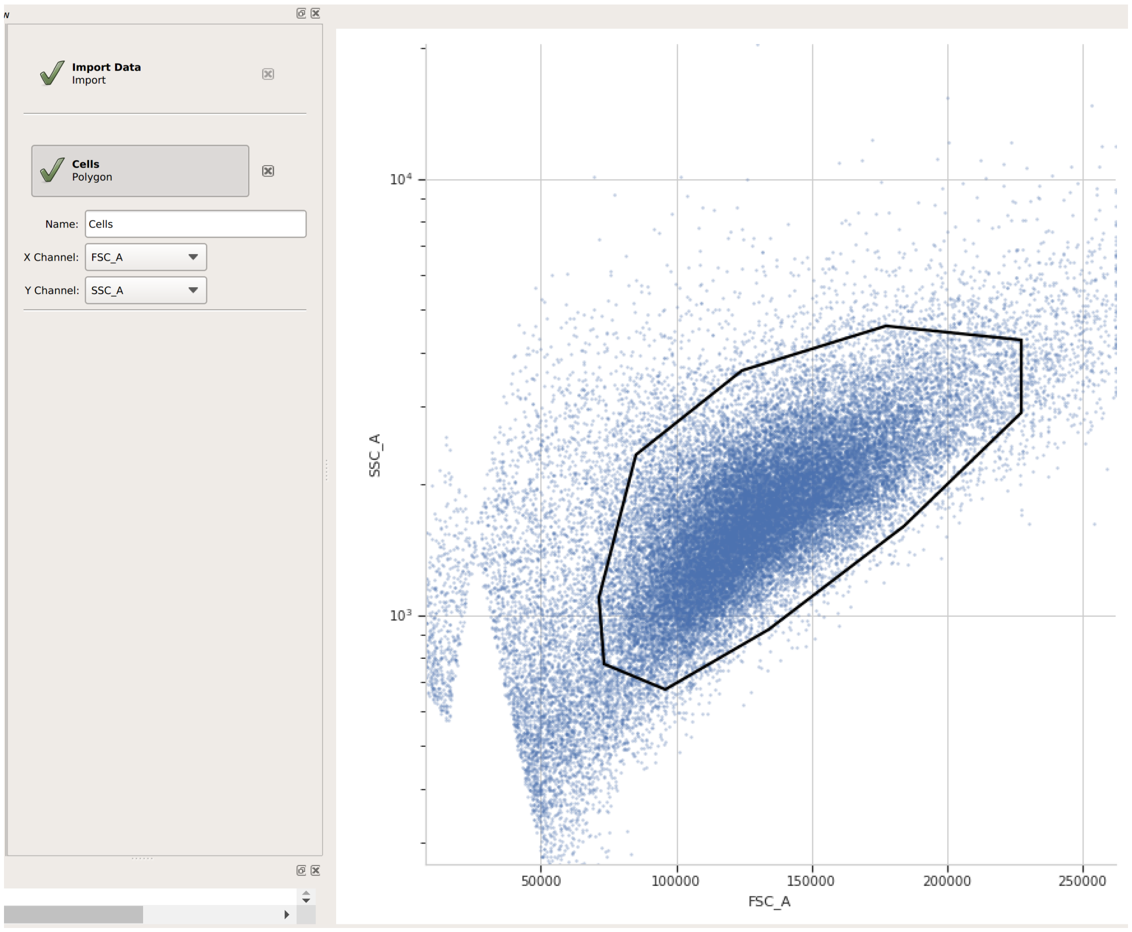
Events per sample: None

Samples: 4

Events: 40000

(c) (Optional but recommended) - use a gate to filter out the “real” cells from debris and clumps. Here, I’m using a polygon gate on the foward-scatter and side-scatter channels to select the population of “real” cells.

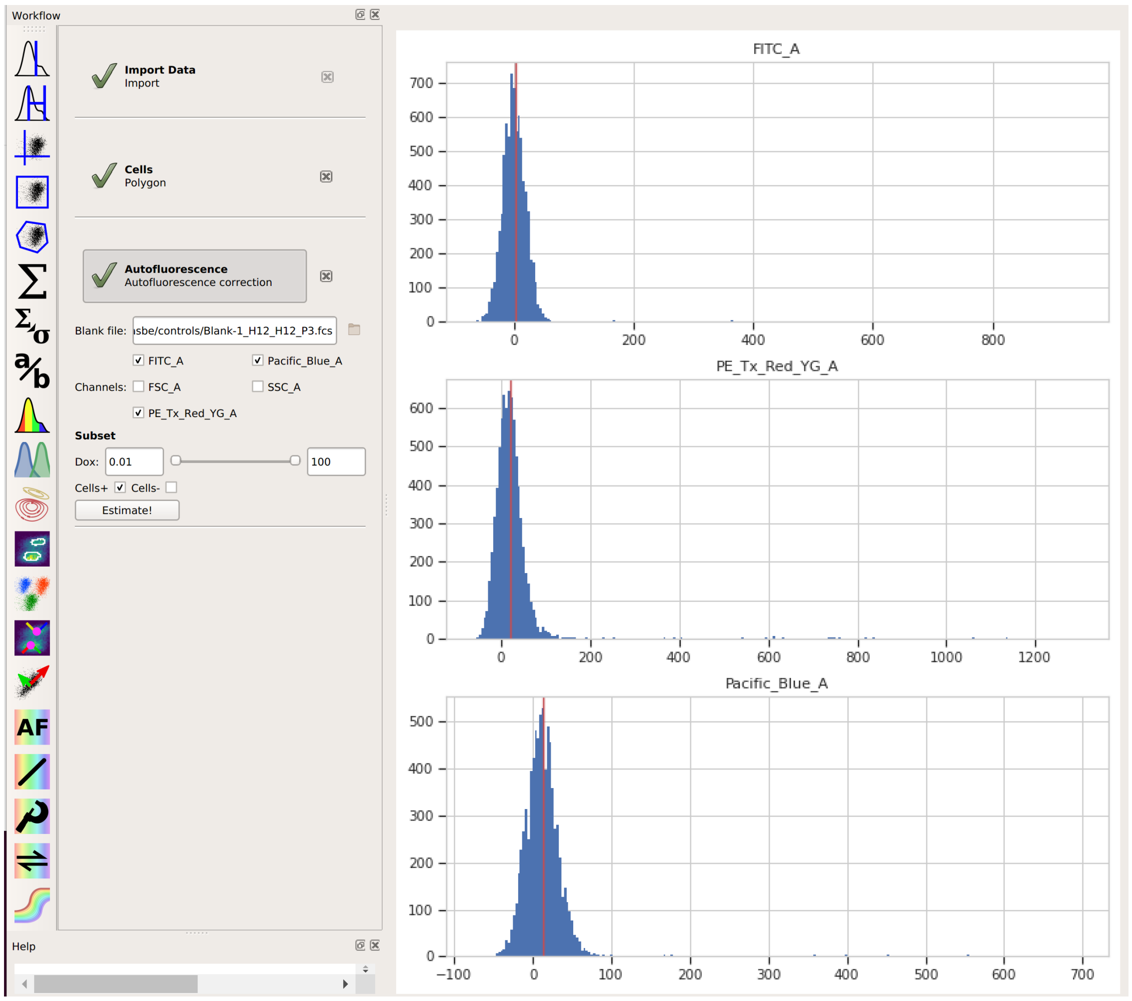
(I've named the population "Cells" – that's how we'll refer to it subsequently.



Name: Cells
X Channel: FSC_A
Y Channel: SSC_A

(d) Add the **Autofluorescence** operation (it's the AF button). Specify the file containing the data from the blank control, choose the channels you want to apply the correction to, and (if you followed the optional step above) choose the subset that you want to use to estimate the correction from. Once you're done, click **Estimate!**

The diagnostic plot shows a histogram of the fluorescence values from the blank file, and the red line indicates their median. This is the amount of "autofluorescence" that will be subtracted from your experimental data.



Blank file: ›sbe/controls/Blank-1_H12_H12_P3.fcs

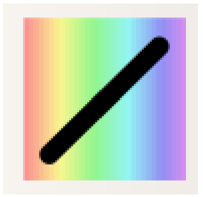
Channels: [✓ FITC_A] [✓ Pacific_Blue_A] FSC_A SSC_A [✓ PE_Tx_Red_YG_A]

Subset

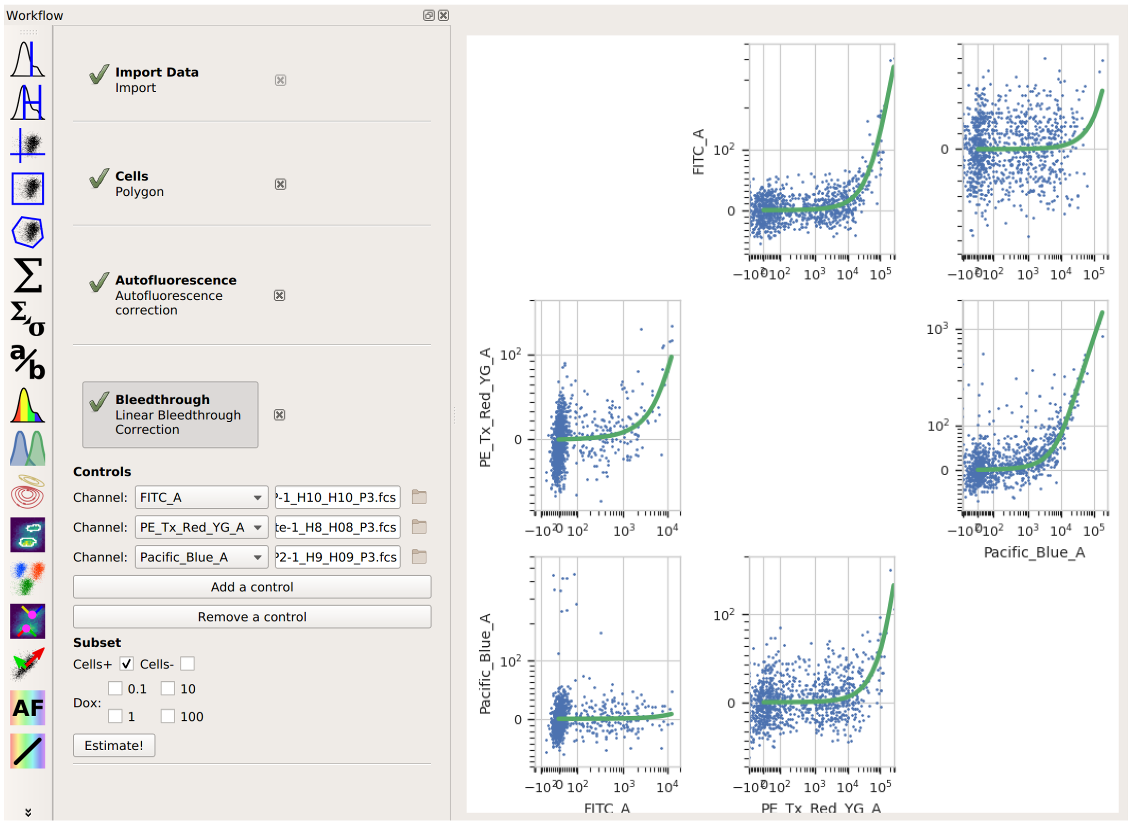
Dox: 0.01 100

[✓ Cells+] Cells-

Note: By choosing the “Cells” subset, I told `Cytoflow` to apply the *same gate* from that operation *to the blank data* before estimating the autofluorescence correction. This way, the clumps and debris in the sample don’t influence that estimate. We’ll do the same thing in the next step, when we apply the bleedthrough compensation.

(e) Add the **Bleedthrough** operation (it’s the  button). For each control you have, click **Add Control**. In the **Controls** list, choose the channel you’re correcting and the file that contains the control data. Again, if you followed the optional step, also choose the subset you want to estimate the correction from. Then, click **Estimate!**.

Here, the diagnostic plots show the data in the control files and the estimate that was fit to them. Because it’s a *linear* estimate, but the data is plotted on a *logarithmic* scale, the estimate lines are shown as curves.



Controls
Channel: FITC_A ›-1_H10_H10_P3.fcs
Channel: PE_Tx_Red_YG_A te-1_H8_H08_P3.fcs
Channel: Pacific_Blue_A ›2-1_H9_H09_P3.fcs

Subset
Cells+ [✓] Cells- [ ]
Dox: [ ] 0.1 [ ] 10 [ ] 1 [ ] 100

And that's it. Now you can continue on with your analysis, secure in the knowledge that you've successfully separated the adjacent fluorescence channels.

If you'd like to learn more, Abcam has a good page about compensation, and Mario Roederer has an even more detailed treatment.

5.1.12 HOWTO: Use beads to correct for day-to-day variation

Flow cytometers are complex instruments, and the precise numerical values they report depend not only on the fluorescence of the sample, but also on the illumination intensity, optical setup, gain of the detectors (ie PMT voltages), and even things such as whether the flow cell has been cleaned recently. We often want to compare data that was collected on different days, and this instrument *drift* can make these comparisons difficult.

Many cytometers, especially those used in a clinical setting, use a daily calibration to counter these effects. However, these calibrations are intended for standardized protocols that are run regularly, not the one-off experiments that many investigators run.

One approach to this problem is to *calibrate* each day's measurements using a stable calibrant. The idea is straightforward: each day, in addition to your experimental samples, you measure a sample of stable fluorescent particles (of known fluorescence), then use these calibrant measurements to convert the arbitrary units (au) for your experimental samples to the known units for your calibrant. For example, if your particles have a fluorescence of 1000 molecules of fluoresceine (MEFL), and you measure their brightness to be 5000 au, then you know that an experimental sample with a brightness of 10,000 au is equivalent to 2,000 MESF. This relationship holds even if tomorrow the laser is a little dimmer or the fluidics are a little dirty.

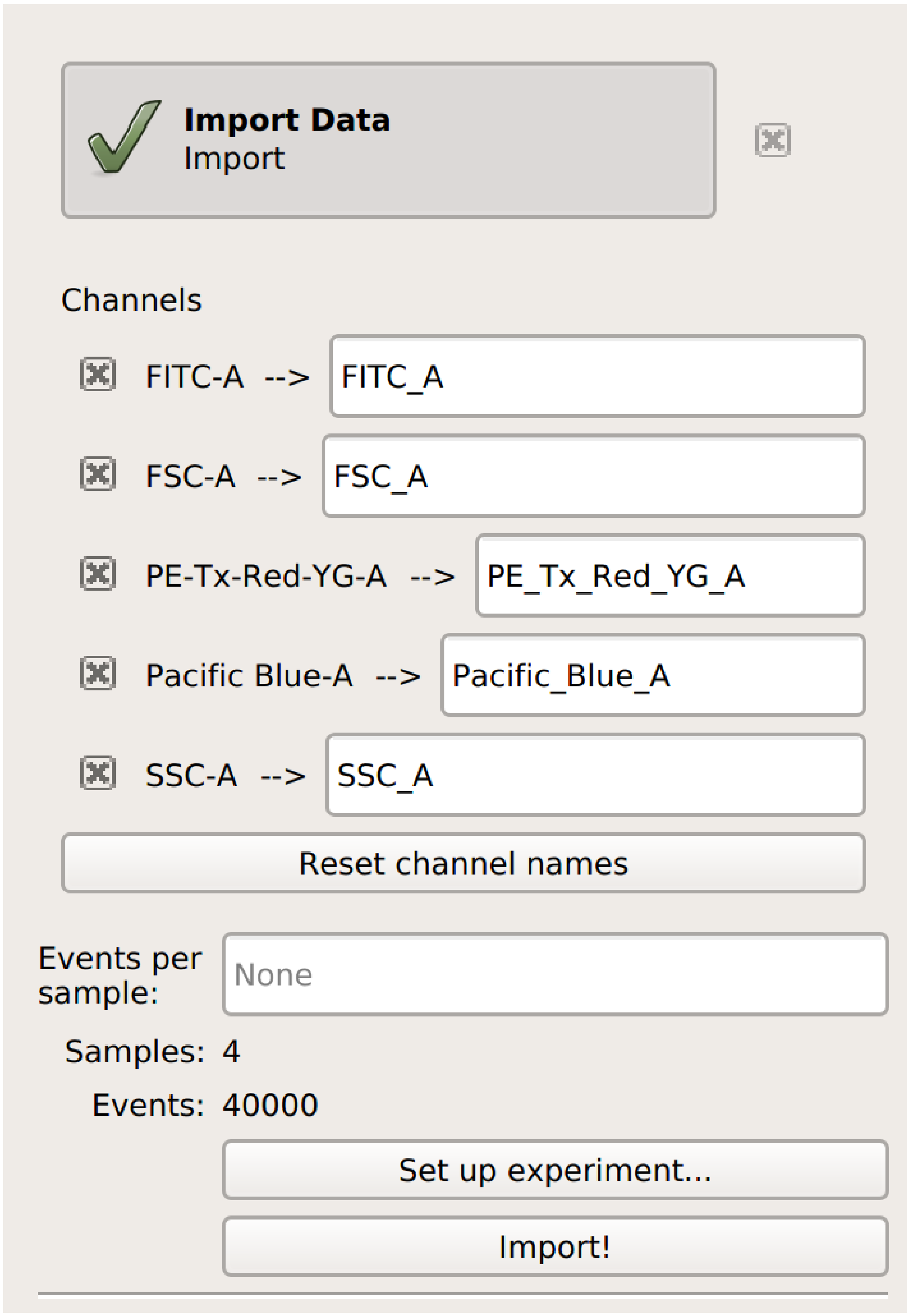
While there are a number of different kinds of stable calibrants that might be used this way, we have had very good success with Spherotech's Rainbow Calibration Particles. (We usually refer to these particles as "beads".) Because they're made of polystyrine with the fluorophores "baked in" (not on the surface), they are ridiculously stable. Spherotech also provides enough technical data about their fluorescence to make it easy to determine a calibration curve and apply it to experimental data. Below, you can find the process for doing so using `Cytoflow`.

Note: It is often considered "best practice" only to compare data that was on the same instrument with the same detector gain settings. Using beads this way can help you get around the "same detector gain settings" requirement – for example, if you have a really bright sample and a really dim sample – but it is *not* a good idea to try to compare between different instruments unless those instruments have *exactly* the same optical configuration (lasers, filters, and from the same vendor.)

Procedure

(a) Collect a sample of beads *on the same day, and with the same settings,* as your experimental samples.

(b) Import your data into `Cytoflow`. Do *not* import your bead control. In the example below, we'll have three fluorescence channels – *Pacific Blue-A*, *FITC-A* and *PE-Tx-Red-YG-A* – in addition to the forward and side-scatter channels.



Import Data
Import

Channels

FITC-A --> FITC_A

FSC-A --> FSC_A

PE-Tx-Red-YG-A --> PE_Tx_Red_YG_A

Pacific Blue-A --> Pacific_Blue_A

SSC-A --> SSC_A

Reset channel names

Events per sample: None

Samples: 4

Events: 40000

Set up experiment...

Import!

(c) Add the **Beads** operation to your workflow. It's the icon.

(d) Choose the beads you used, *including the specific lot*, from the drop-down list.

Note: If the beads you want to use are not in the list, please submit a bug report. Adding a new set of beads is pretty trivial.

(e) Specify the file containing the data from the beads.

(f) Click **Add a channel** for every channel you want to calibrate. In the **Channels** list, specify both the *channel* you want to calibrate and the *units* you want to calibrate it.

Note: It works best to choose units of a fluorophore that is spectrally matched to the channel that you're calibrating. Here are the beads that `Cytoflow` knows about (including the laser and filter sets used to characterize the beads):

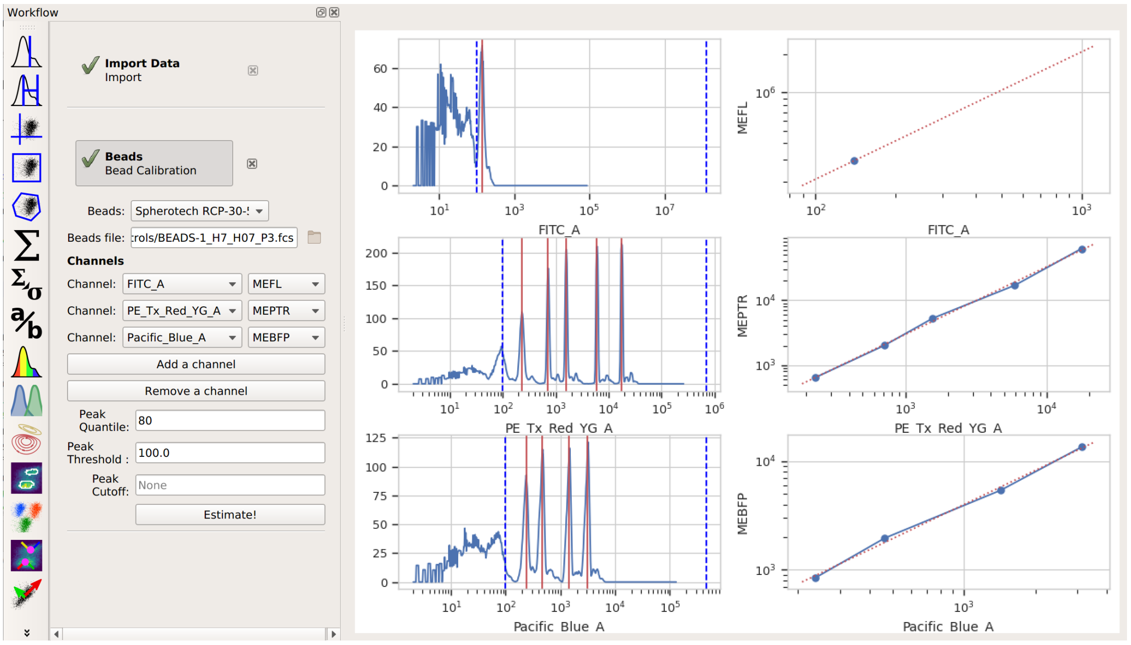
- Spherotech ACP 30-2K
- Spherotech RCP-30-5A Lot AN04, AN03, AN02, AN01, AM02, AM01, AL01, AK04, AK03 & AK02**
- Spherotech RCP-30-5A (Euroflow) Lot EAM02 & EAM01
- Sphreotech RCP-30-5A (Euroflow) Lot EAK01, EAG01, EAE01 & EAF01
- Spherotech RCP-30-5A Lot AK01, AJ01, AH02, AH01, AF02, AF01, AD04 & AE01
- Spherotech RCP-30-5A Lot AG01
- Spherotech RCP-30-5A Lot AA01, AA02, AA03, AA04, AB01, AB02, AC01 & GAA01-R
- Spherotech RCP-30-5A Lot AC02, AC03 & AD01
- Spherotech RCP-30-5A Lot Z02 and Z03
- Spherotech RCP-30-5 Lot AA01, AB01, AB02, AC01 & AD01
- Spherotech RCP-30-5 Lot AM02, AM01, AL01, AH01, AG01, AF01 & AD03
- Spherotech RCP-60-5
- Spherotech URCP 38-2K
- Spherotech URCP 38-2K Lot AN01, AM01, AL02, AL01, AK03, AK02, AK01, AJ02 & AJ03
- Spherotech URCP 50-2K Lot AM01 & AJ01
- Spherotech URCP 50-2K

The Spherotech fluorophore labels and the laser / filter sets used to measure them (that I know about) are:

- **MECSB** (Cascade Blue, 405 –> 450/50)
- **MEBFP** (BFP, 405 –> 530/40)
- **MEFL** (Fluroscein, 488 –> 530/40)
- **MEPE** (Phycoerythrin, 488 –> 575/25)
- **MEPTR** (PE-Texas Red, 488 –> 613/20)

- **MECY** (Cy5, 488 –> 680/30)
- **MEPCY7** (PE-Cy7, 488 –> 750 LP)
- **MEAP** (APC, 633 –> 665/20)
- **MEAPCY7** (APC-Cy7, 635 –> 750 LP)

(g) Click **Estimate!** Check the diagnostic plots to make sure that each peak in your data was found, and that you have a fairly linear relationship between the (measured) peaks and the (known) calibration.


Workflow

Import Data
Import

Beads
Bead Calibration

Beads: Spherotech RCP-30-!

Beads file: :rols/BEADS-1_H7_H07_P3.fcs

Channels

Channel: FITC_A MEFL

Channel: PE_Tx_Red_YG_A MEPTR

Channel: Pacific_Blue_A MEBFP

Peak Quantile: 80

Peak Threshold : 100.0

Peak Cutoff: None

Note: If not all of the peaks were identified, try messing around with the peak-finding parameters.

Note: Bead calibration is particularly powerful when combined with the autofluorescence correction and bleedthrough compensation described in *HOWTO: Compensate for bleedthrough*. They're so useful when done together that this sequence of operations has its own module – see *HOWTO: Use the TASBE workflow for calibrated flow cytometry*.

5.1.13 HOWTO: Use the TASBE workflow for calibrated flow cytometry

As outlined in *HOWTO: Use beads to correct for day-to-day variation*, flow cytometry's quantitative power is somewhat belied by the sensitivity of single-cell measurements to instrument configuration and state. In a word, cytometers drift – sometimes by as much as 20% over the course of a day, even with identical settings. This can complicate experiments or processes that depend on precise, reproducible measurements – predictive modeling of gene expression in particular.

Fortunately, a set of data manipulations can go a long way towards fixing these issues:

- *Subtract* autofluorescence from the experimental measurements. This requires the measurement of a *non-fluorescent* control.

- *Compensate* for spectral bleed-through. This requires the measurement of controls stained with (or expressing) the individual fluorophores alone.
- *Calibrate* the measurements using a stable calibrant. This requires measuring a calibrant such as the Spherotech rainbow calibration particles
- (Optional) *Translate* all the channels so they're in the same units. This requires the measurement of multi-color controls where the staining (or expression) of each fluorophore is equal.

You can read more about this workflow in the following resources:

- A Method for Fast, High-Precision Characterization of Synthetic Biology Devices
- Accurate Predictions of Genetic Circuit Behavior from Part Characterization and Modular Composition
- The TASBE Jupyter notebook

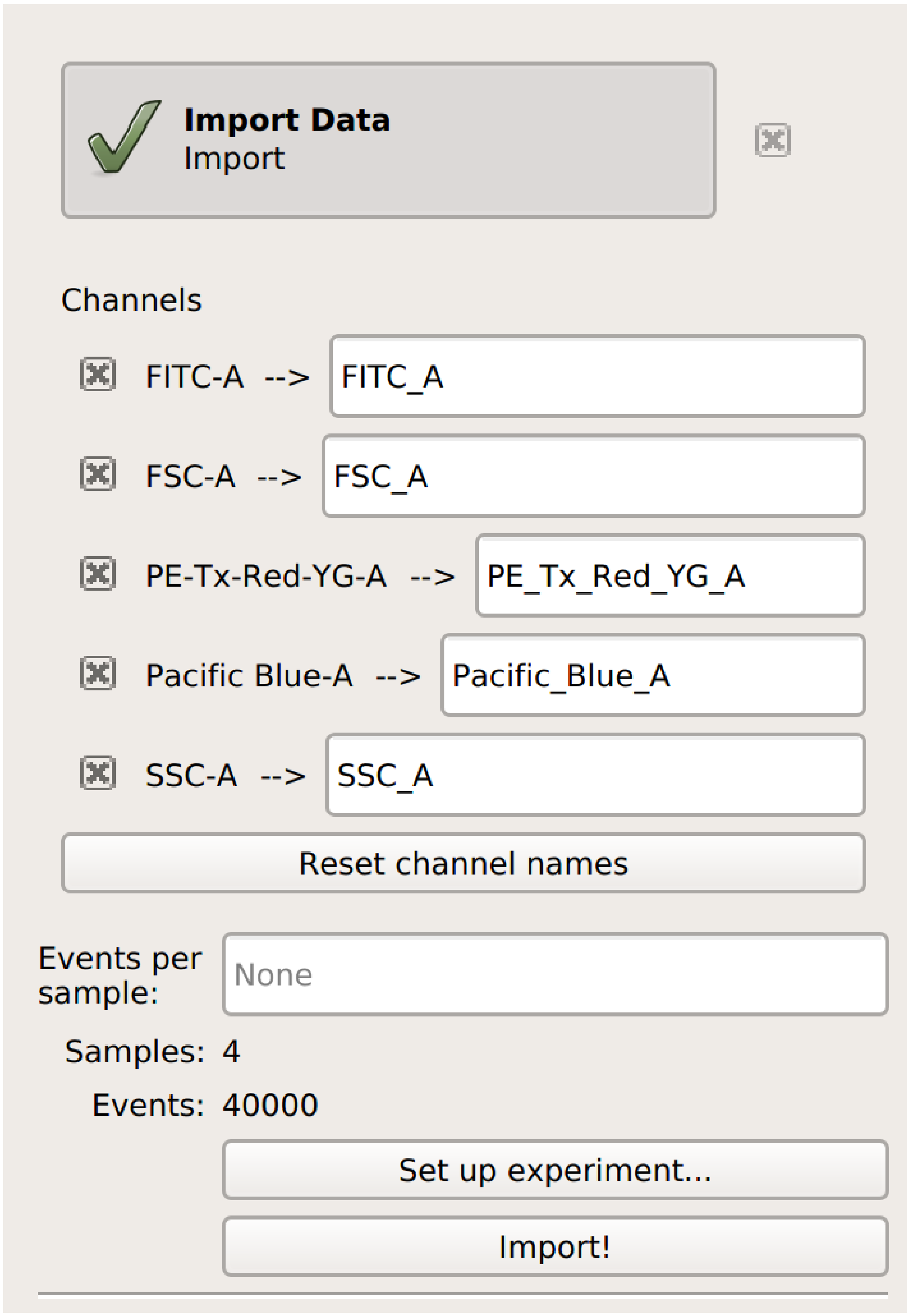
Because this method originated in software called *Tool-Chain to Accelrate Synthetic Biology Engineering*, we abbreviate it *TASBE*. And while it was developed in the context of modeling and simulating synthetic gene networks, there is *nothing* about it that is syn-bio specific. If you want to generate data sets that you can directly compare to eachother (particularly if they're collected on the same instrument), this is the calibration that will let you to so. And while each of the steps above has its own operation, they're also combined in the **TASBE** operation.

Procedure

(a) Collect the following controls:

- Blank, un-stained, non-fluorescent cells (to subtract autofluorescence).
- Cells stained with (or expressing) only one fluorophore (to compensate for spectral bleedthrough).
- Beads (to calibrate the measurements)
- (Optional) multi-color controls, where each *pair* of fluorescent molecules stains (or is expressed) at equal intensity.

(b) #. Import your data into `Cytoflow`. Do *not* import your control samples (unless they're part of the experiment.) In the example below, we'll have three fluorescence channels – *Pacific Blue-A*, *FITC-A* and *PE-Tx-Red-YG-A* – in addition to the forward and side-scatter channels.


Import Data
Import

Channels

- FITC-A --> FITC_A
- FSC-A --> FSC_A
- PE-Tx-Red-YG-A --> PE_Tx_Red_YG_A
- Pacific Blue-A --> Pacific_Blue_A
- SSC-A --> SSC_A

Reset channel names

Events per sample: None

Samples: 4

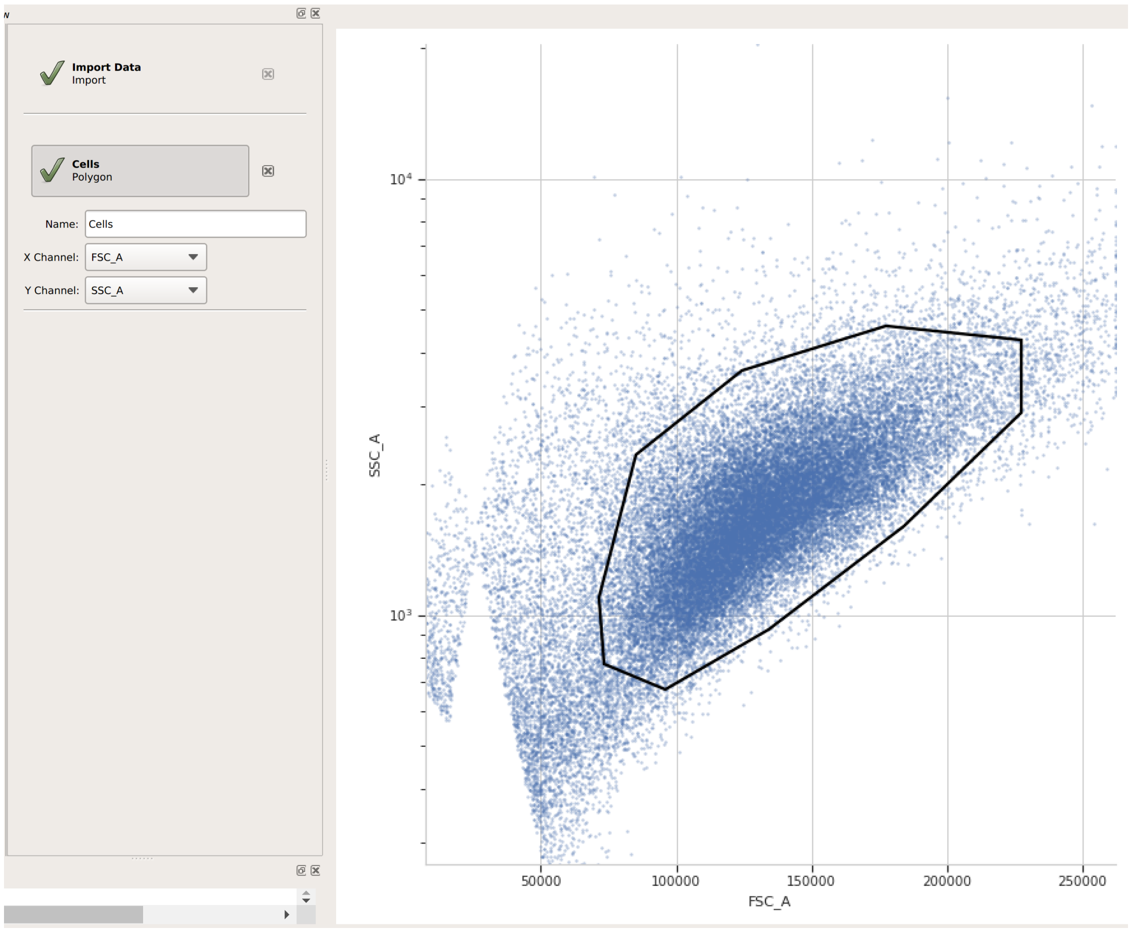
Events: 40000

Set up experiment...

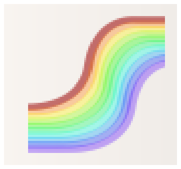
Import!

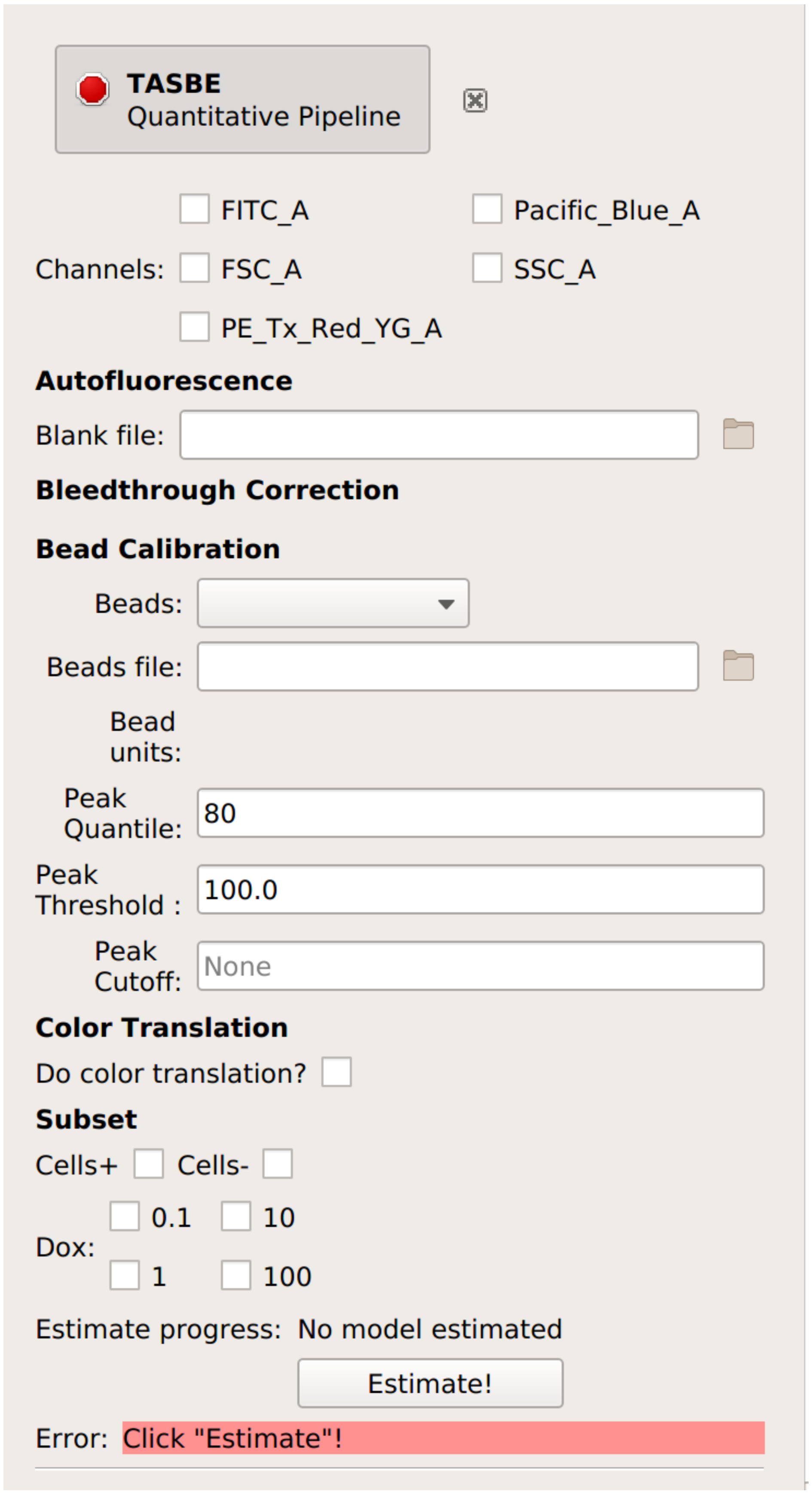
(c) (Optional but recommended) - use a gate to filter out the "real" cells from debris and clumps. Here, I'm using a polygon gate on the foward-scatter and side-scatter channels to select the population of "real" cells.

(I've named the population "Cells" – that's how we'll refer to it subsequently.



Name: Cells
X Channel: FSC_A
Y Channel: SSC_A

(d) Add the **TASBE** operation (it's the  button).



Channels: ☐ FITC_A ☐ Pacific_Blue_A ☐ FSC_A ☐ SSC_A ☐ PE_Tx_Red_YG_A

Autofluorescence

Blank file:

Bleedthrough Correction

Bead Calibration

Beads:

Beads file:

Bead units:

Peak Quantile: 80

Peak Threshold : 100.0

Peak Cutoff: None

Color Translation

Do color translation? ☐

Subset

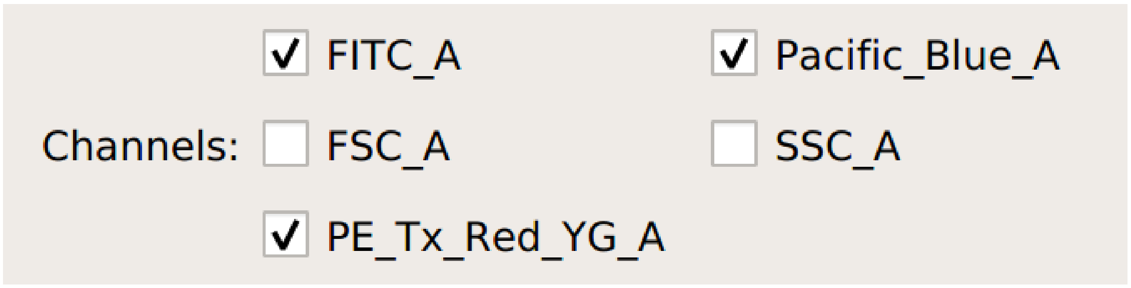
Cells+ ☐ Cells- ☐

Dox: ☐ 0.1 ☐ 10 ☐ 1 ☐ 100

Estimate progress: No model estimated

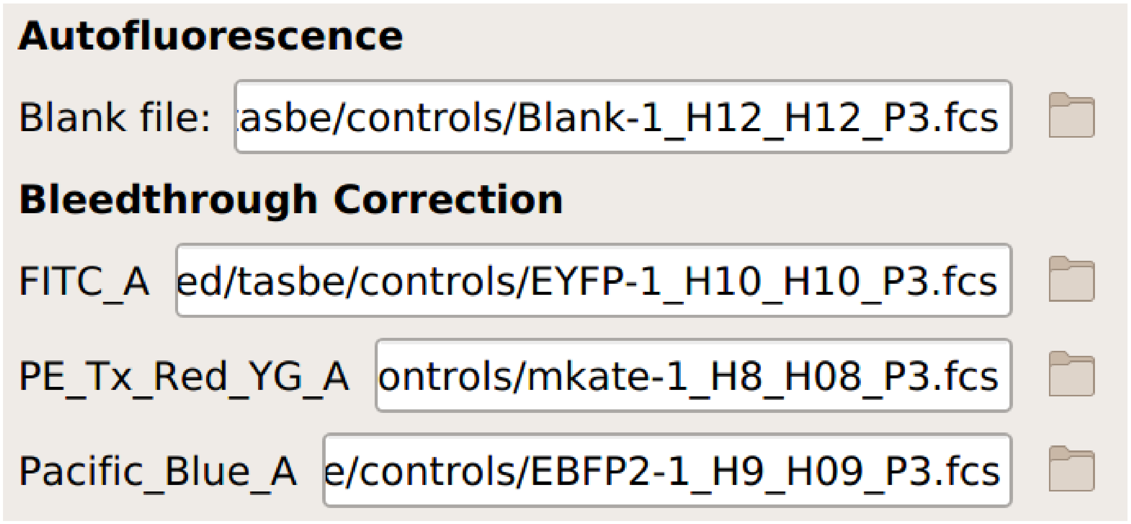
Error: Click "Estimate"!

(e) Select which channels you are calibrating.



Channels: [✓ FITC_A] [✓ Pacific_Blue_A] FSC_A SSC_A [✓ PE_Tx_Red_YG_A]

(f) Specify the files containing the blank and single-fluorophore controls.



Autofluorescence

Blank file: asbe/controls/Blank-1_H12_H12_P3.fcs

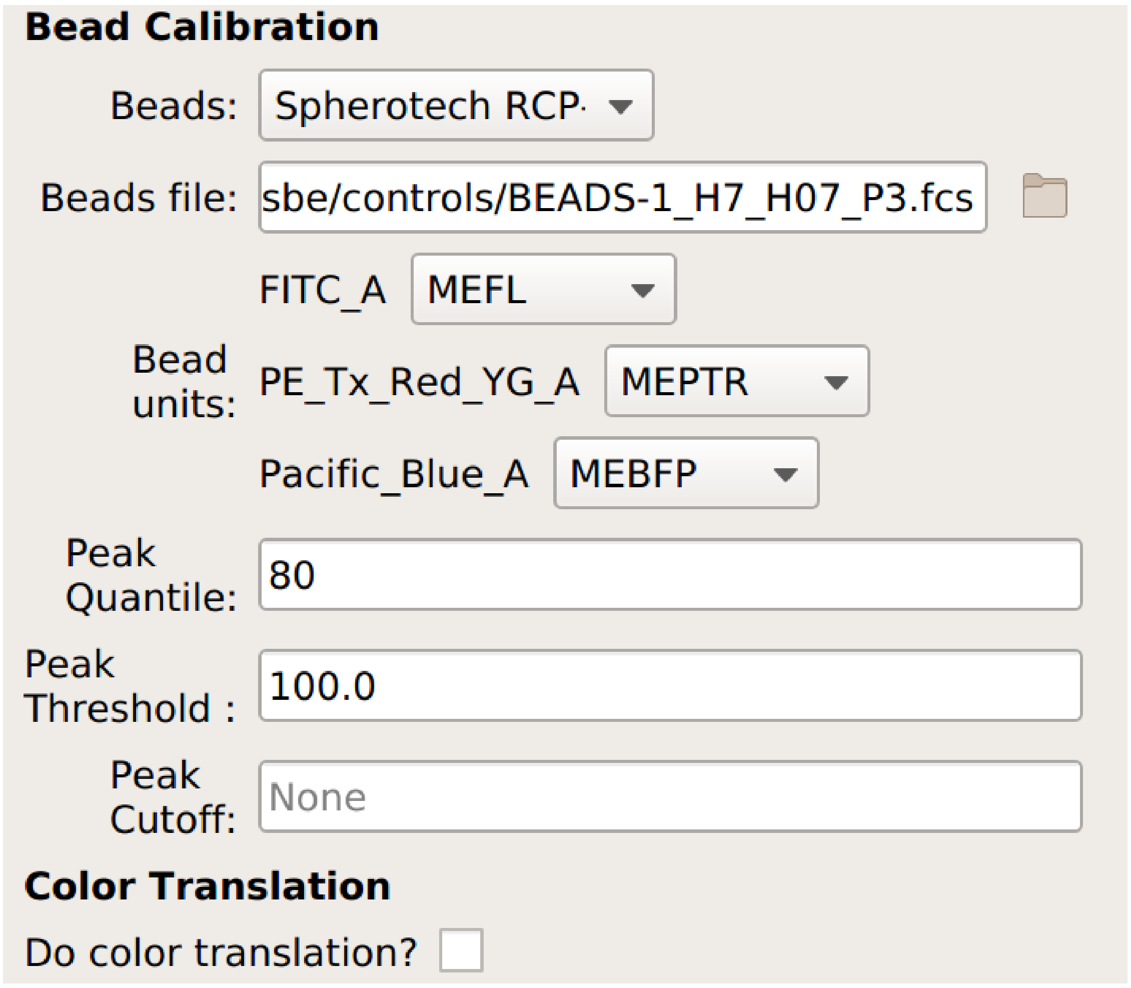
Bleedthrough Correction

FITC_A ed/tasbe/controls/EYFP-1_H10_H10_P3.fcs

PE_Tx_Red_YG_A ontrols/mkate-1_H8_H08_P3.fcs

Pacific_Blue_A e/controls/EBFP2-1_H9_H09_P3.fcs

(g) If you are *not* doing a unit translation: select the beads you're using, the data file, and the appropriate units for each channel:



Bead Calibration

Beads: Spherotech RCP·

Beads file: sbe/controls/BEADS-1_H7_H07_P3.fcs

Bead units:

FITC_A MEFL

PE_Tx_Red_YG_A MEPTR

Pacific_Blue_A MEBFP

Peak Quantile: 80

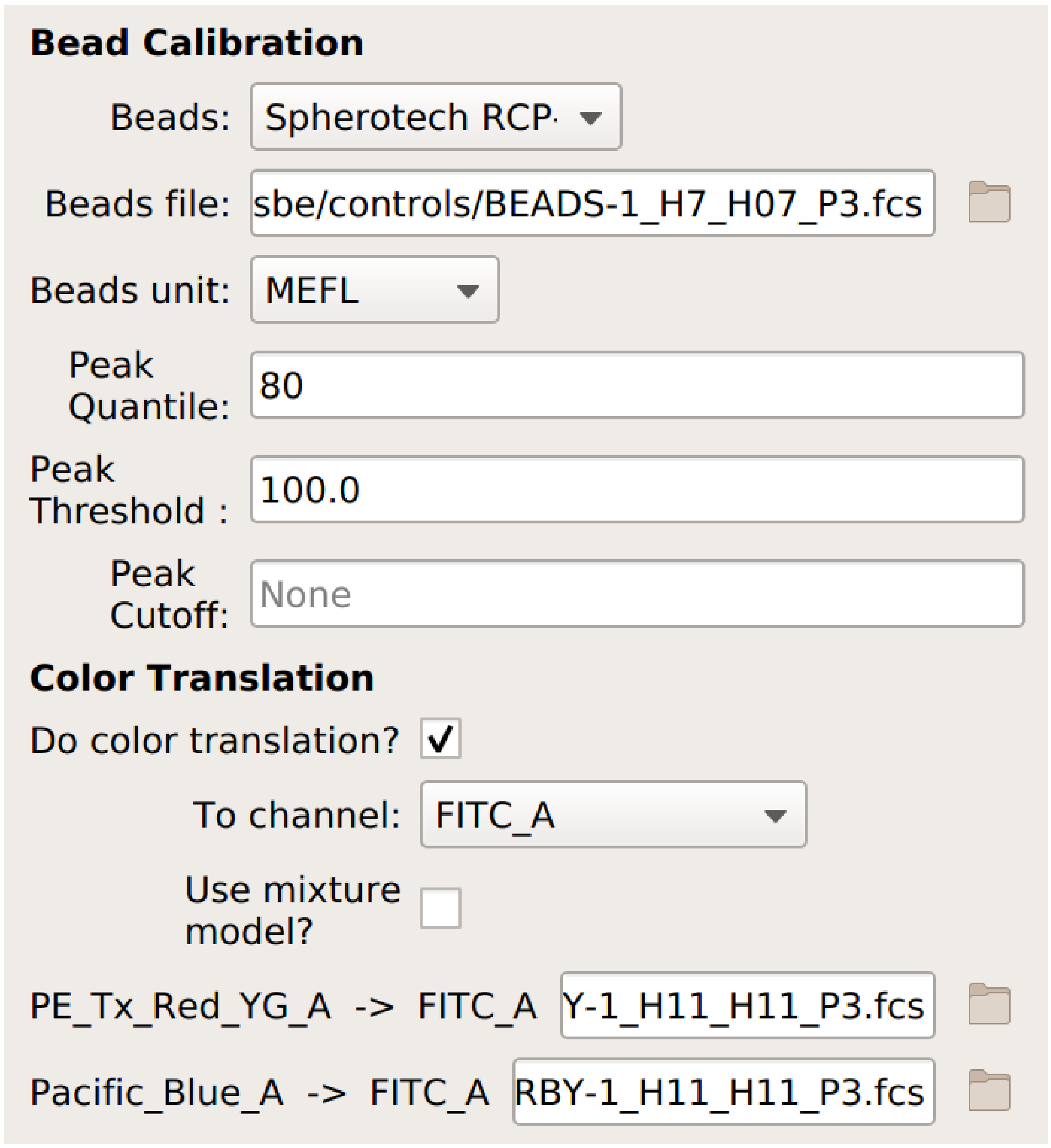
Peak Threshold : 100.0

Peak Cutoff: None

Color Translation

Do color translation? ☐

(h) If you *are* doing a unit translation, specify the beads you're using, the data file, the unit you want *all* the channels in, the channel you want everything translated to, and the multi-color control files.


Bead Calibration

Beads: Spherotech RCP·

Beads file: sbe/controls/BEADS-1_H7_H07_P3.fcs

Beads unit: MEFL

Peak Quantile: 80

Peak Threshold : 100.0

Peak Cutoff: None

Color Translation

Do color translation? ✓

To channel: FITC_A

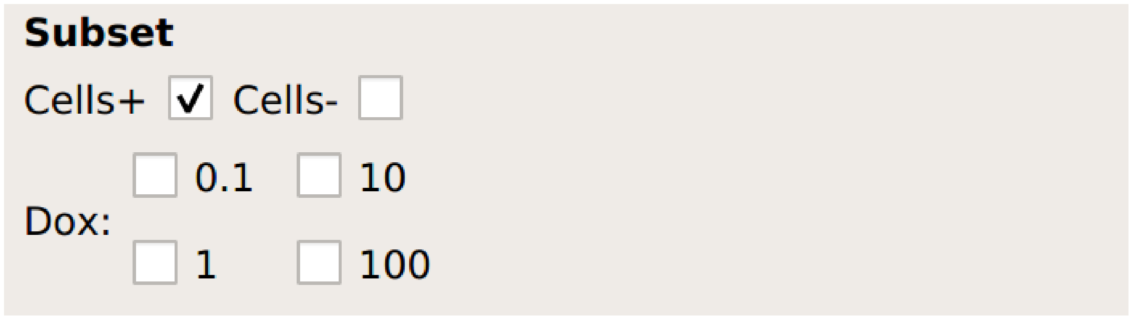
Use mixture model?

PE_Tx_Red_YG_A -> FITC_A Y-1_H11_H11_P3.fcs

Pacific_Blue_A -> FITC_A RBY-1_H11_H11_P3.fcs

Note: If you're measuring cells that have a large non-fluorescent population – such as transfected mammalian cells – choose **Use mixture model** as well.

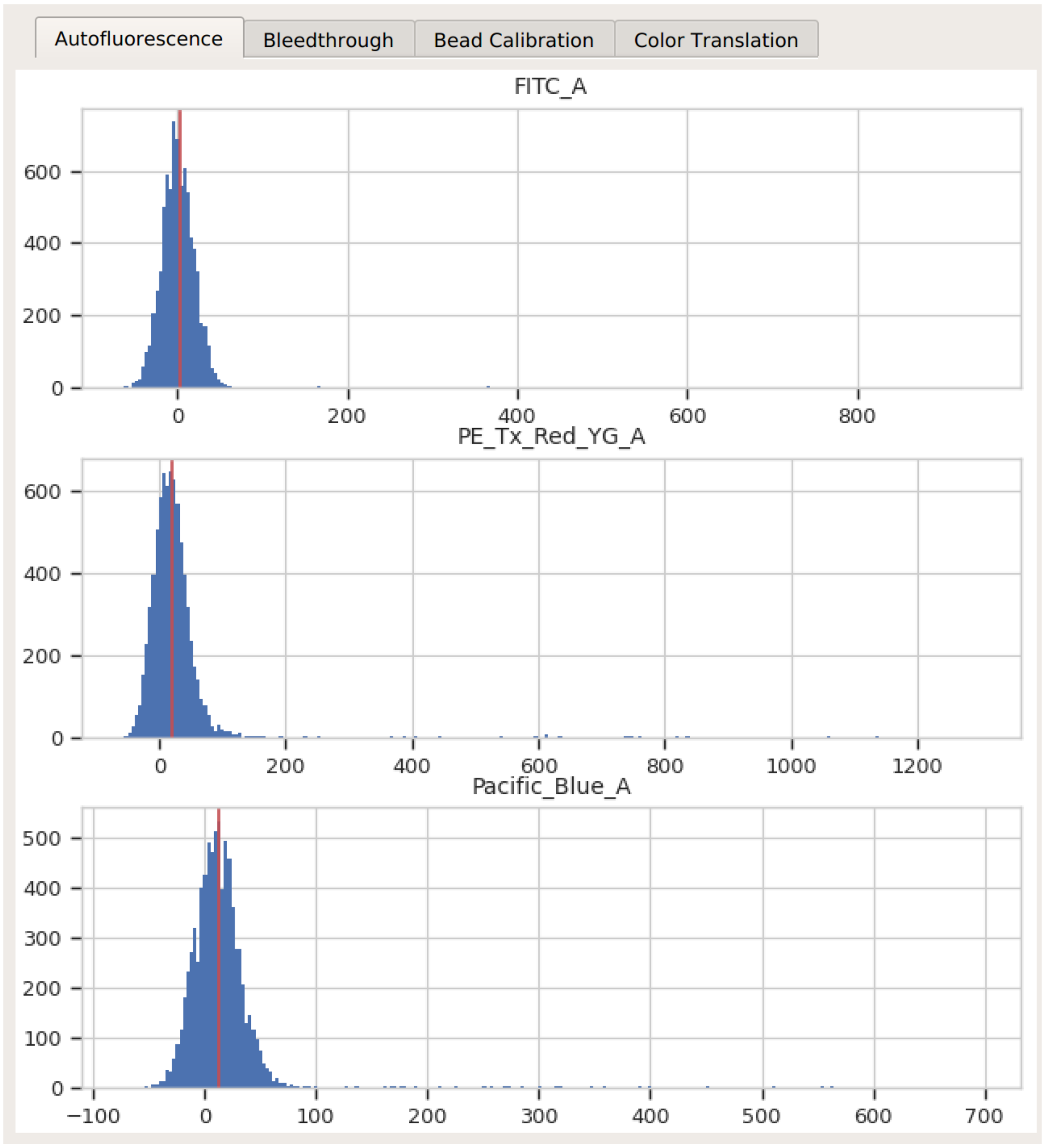
(i) (Optional but recommended) - if you set a gate above to select for cells (and not clumps or debris), select that gate under **Subset**:'



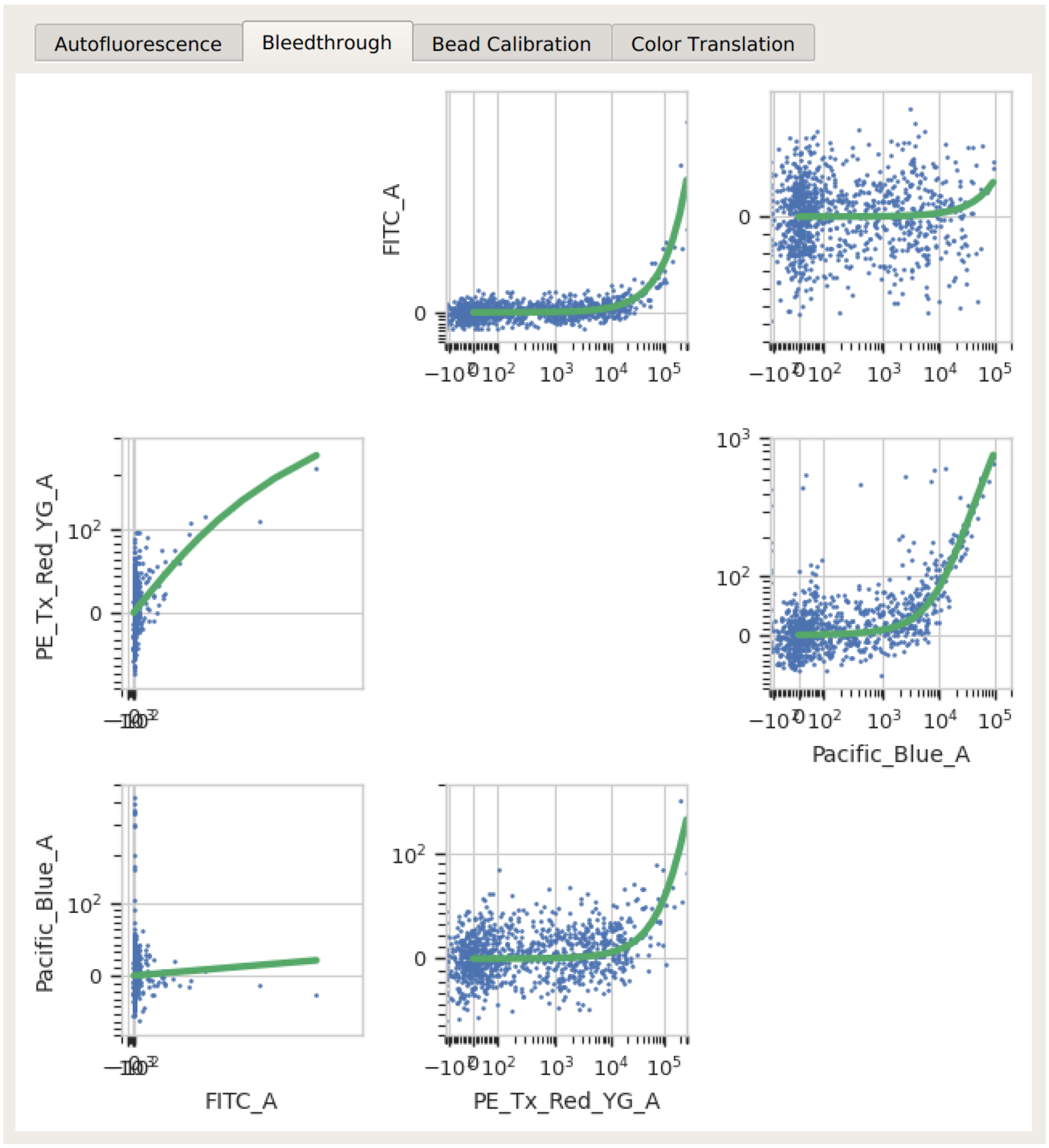
(j) Click **Estimate!**

(k) Check all three (or four) diagnostic plots – make sure that the estimates look reasonable.

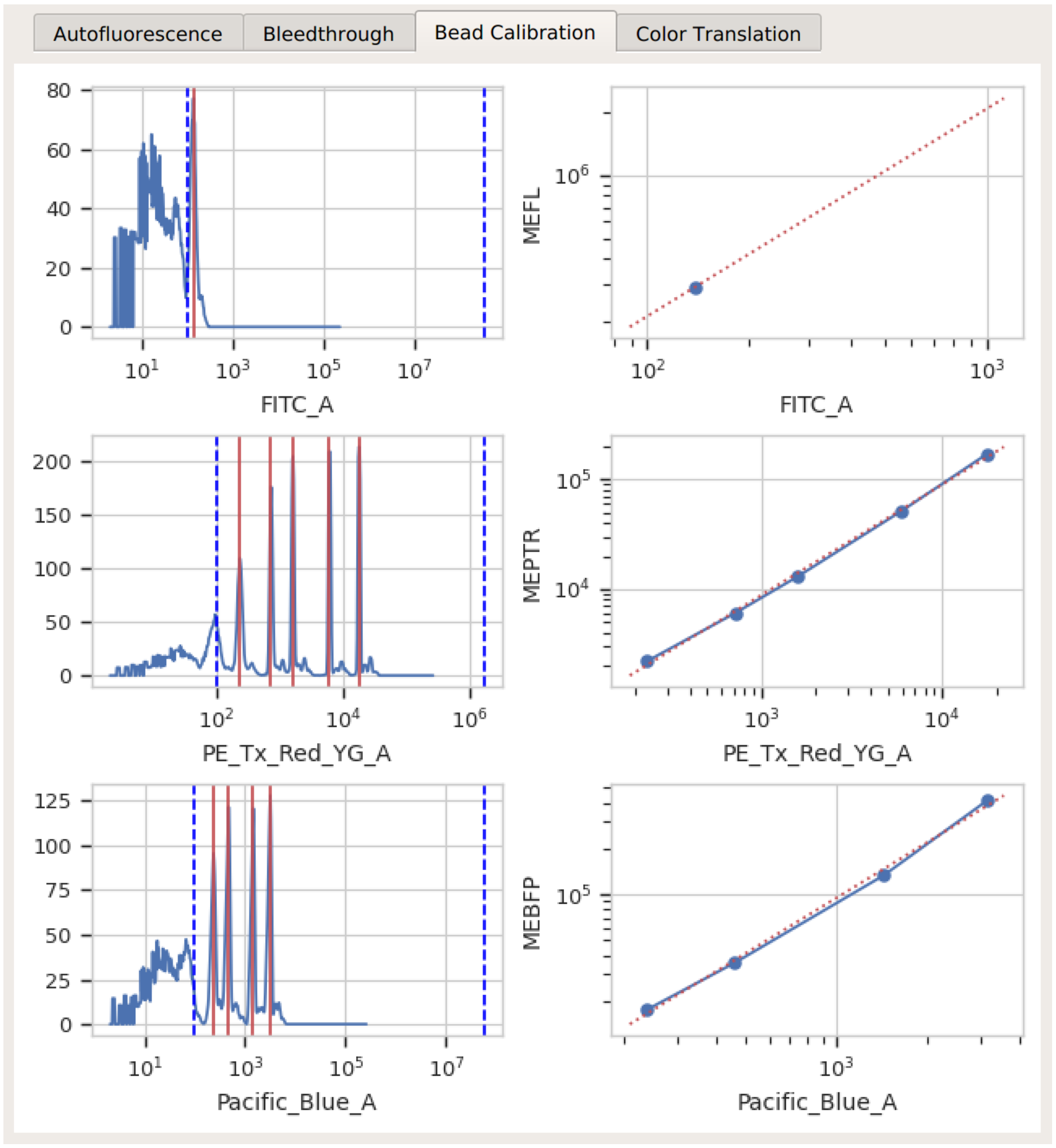
- Autofluorescence – there's a single peak in each channel, and a red line at the center.



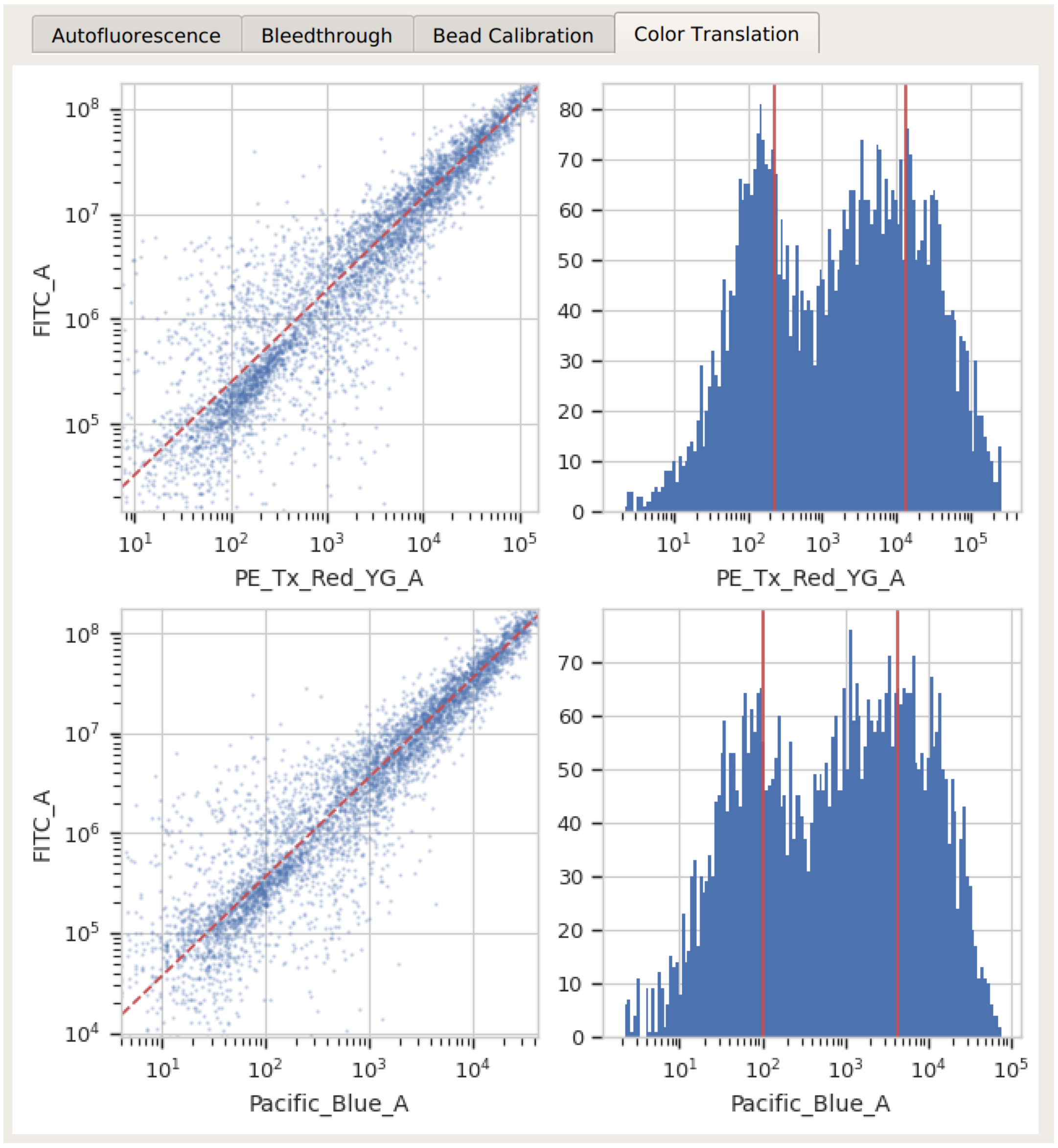
- Bleedthrough – the bleedthrough estimates (green lines) follow the data closely.



- Bead Calibration – all the peaks between the minimum and maximum cutoffs (blue dashed lines) were found, and there's a linear relationship with the vendor-supplied values.



- Color translation – there's a linear relationship between the channels, and if you're using a mixture model, the centers of the two distributions were successfully identified.



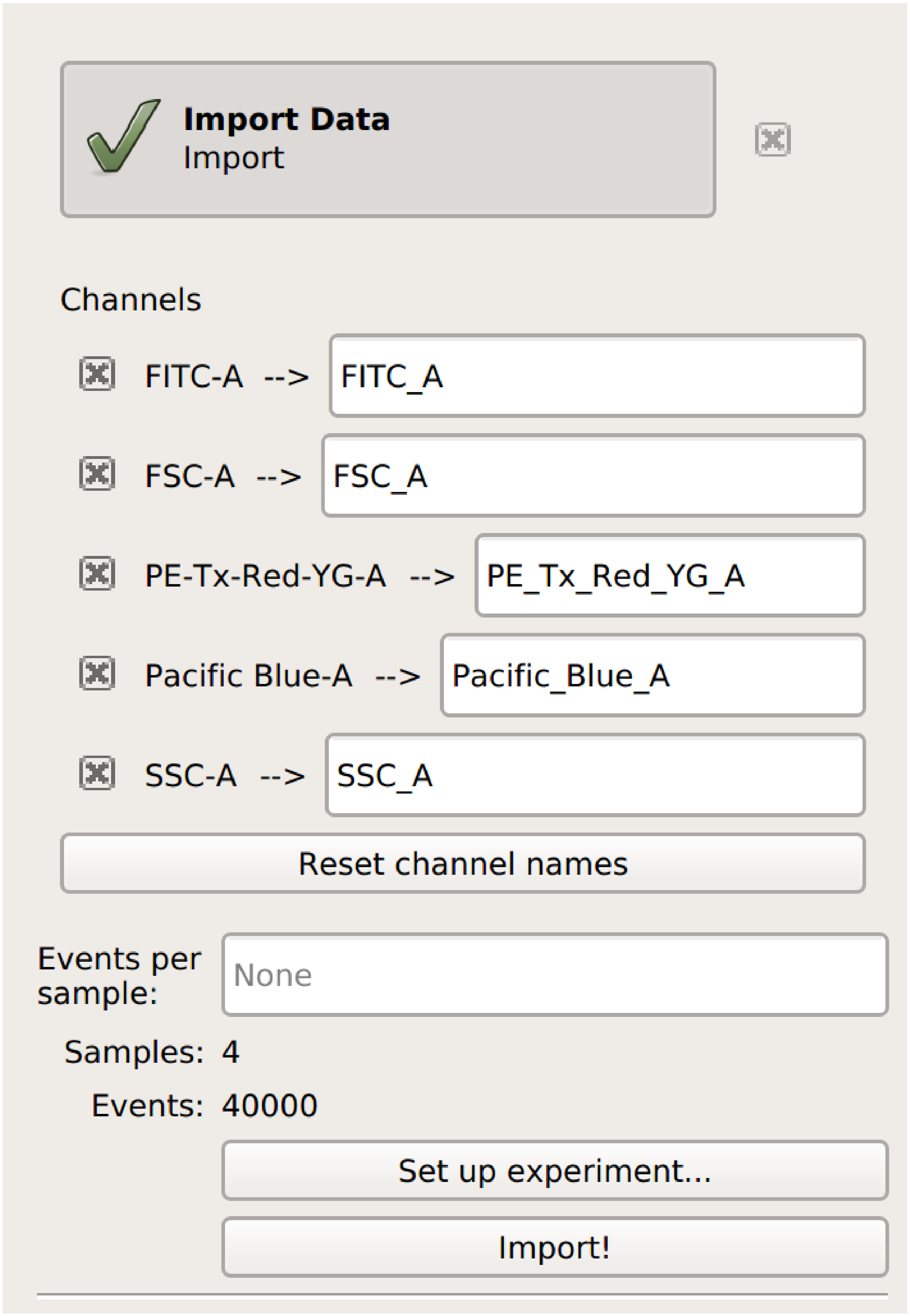
(l) Proceed with your analysis. If this will involve multiple data sets from multiple days, you may wish to export the calibrated data back into FCS files, as described here: *HOWTO: Export FCS files*.

5.1.14 HOWTO: Export FCS files

I've tried to make `Cytoflow` both powerful and straightforward to use, but I understand that some people may wish to use other software for their cytometry analysis. `Cytoflow` has some unique capabilities – in particular, its **Bead Calibration** and **TASBE** modules – and it is possible to import data into `Cytoflow`, calibrate or otherwise manipulate it, then export it back to FCS files that can be opened with other analysis programs.

Procedure

(a) Import your data into `Cytoflow`. Do *not* import your control samples (unless they're part of the experiment.) In the example below, we'll have three fluorescence channels – *Pacific Blue-A*, *FITC-A* and *PE-Tx-Red-YG-A* – in addition to the forward and side-scatter channels.


Import Data
Import

Channels

FITC-A --> FITC_A
FSC-A --> FSC_A
PE-Tx-Red-YG-A --> PE_Tx_Red_YG_A
Pacific Blue-A --> Pacific_Blue_A
SSC-A --> SSC_A

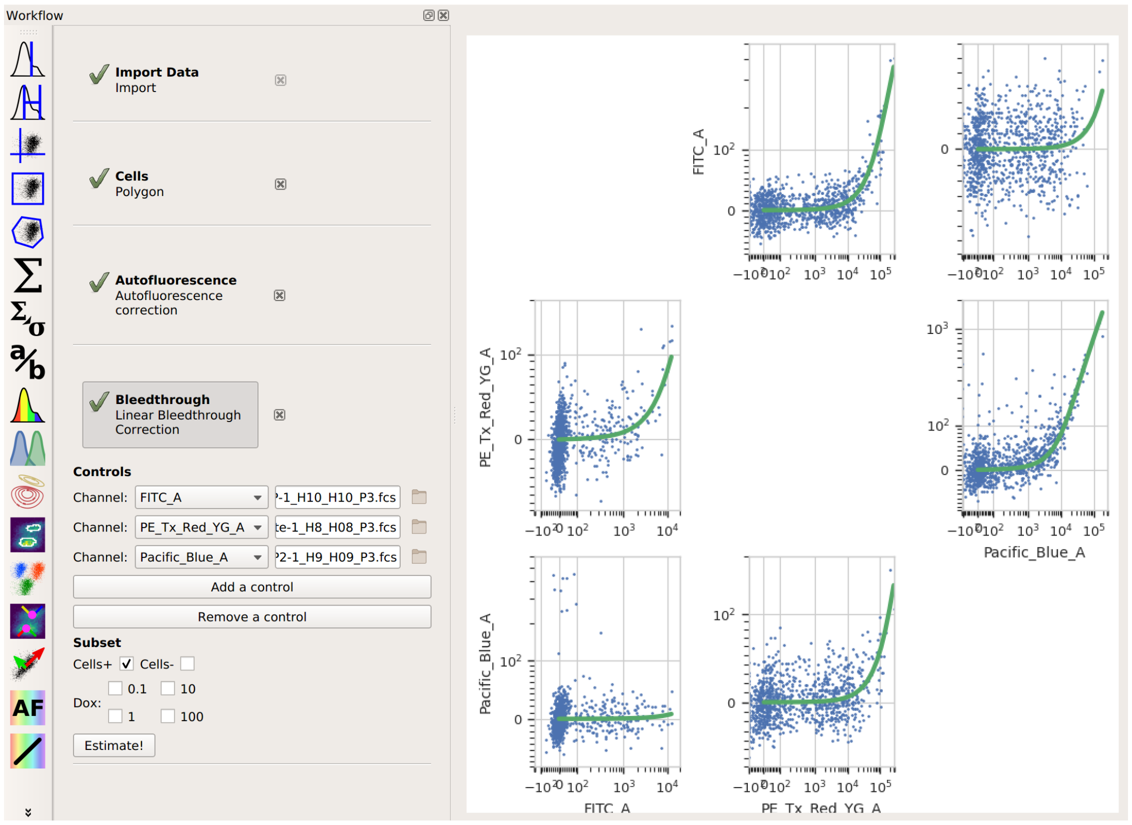
Reset channel names

Events per sample: None

Samples: 4
Events: 40000

Set up experiment...
Import!

(b) Perform whatever calibration or manipulation you would like. In this case, I've applied the **Autofluorescence** and **Bleedthrough** modules.


Workflow

Import Data
Import

Cells
Polygon

Autofluorescence
Autofluorescence correction

Bleedthrough
Linear Bleedthrough Correction

Controls

Channel: FITC_A ›-1_H10_H10_P3.fcs

Channel: PE_Tx_Red_YG_A te-1_H8_H08_P3.fcs

Channel: Pacific_Blue_A ›2-1_H9_H09_P3.fcs

Add a control

Remove a control

Subset

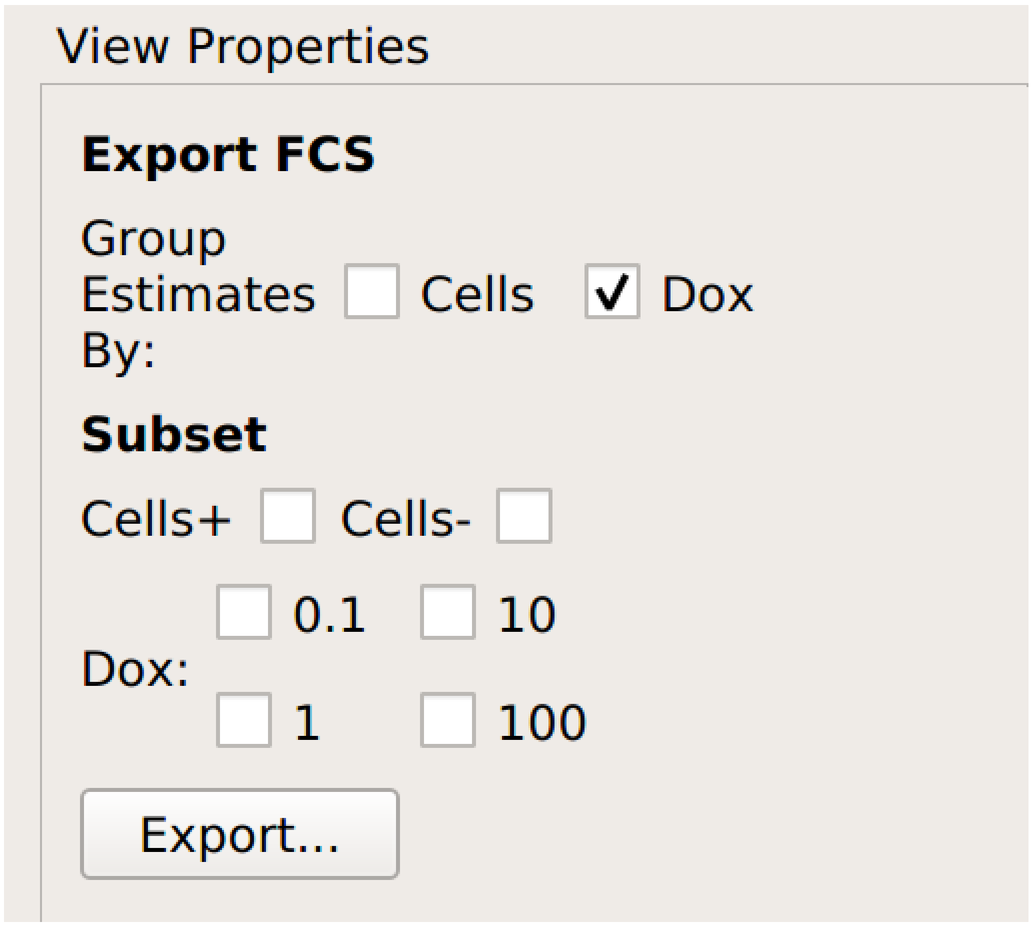
Cells+ [✓] Cells- [ ]

Dox: [ ] 0.1 [ ] 10 [ ] 1 [ ] 100

Estimate!

(c) Choose the **Export FCS** view. It's the button. Note that this is found on the *Views* toolbar, not the *Operations* toolbar.

(d) Select how you'd like the data split up when it is exported. Remember, when `Cytoflow` imports data, it "forgets" everything about the "tubes" that the data came from – it only knows which sets of data were treated with which conditions. Select which conditions should be split into separate FCS files in the view parameters.


View Properties

Export FCS

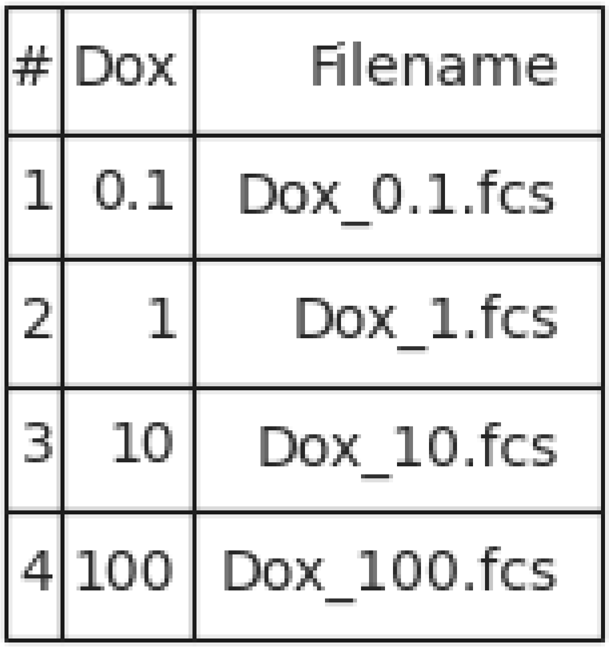
Group Estimates By: ☐ Cells ✓ Dox

Subset

Cells+ ☐ Cells- ☐

Dox: ☐ 0.1 ☐ 10 ☐ 1 ☐ 100

In this case, I've asked for each unique value of **Dox** to go into its own FCS file. The table in the view pane will show you which files will be created.



#	Dox	Filename
1	0.1	Dox_0.1.fcs
2	1	Dox_1.fcs
3	10	Dox_10.fcs
4	100	Dox_100.fcs

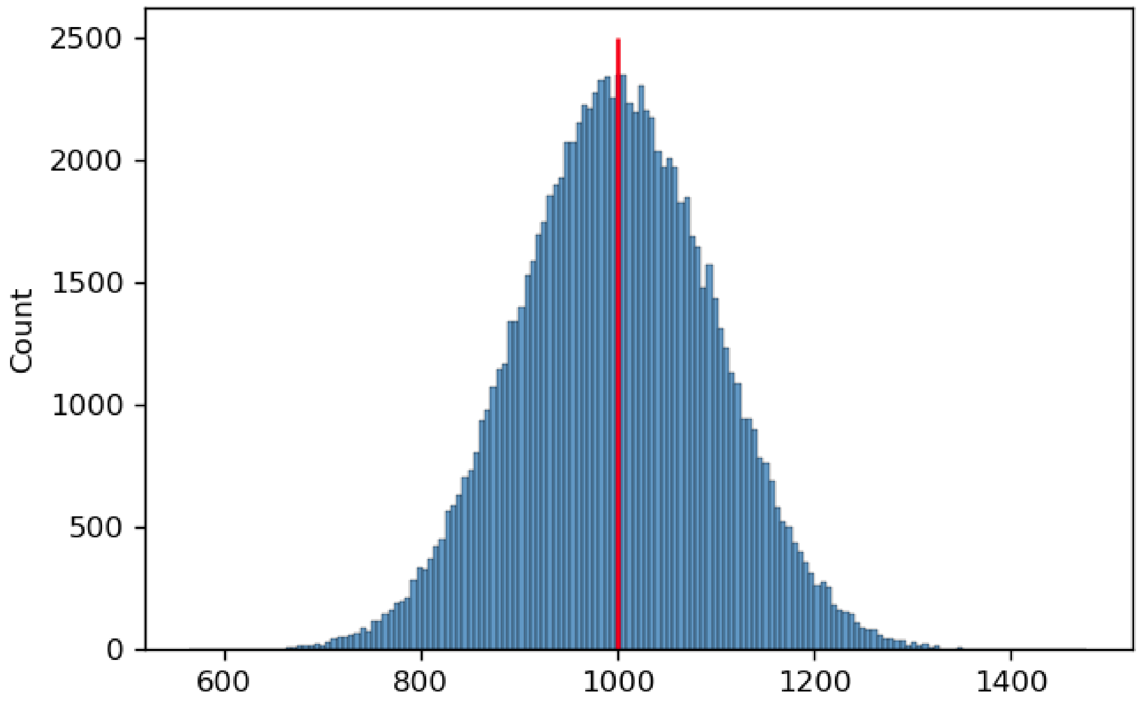
   (e) Click **Export…** and select the directory to save the FCS files to.

3. *Guides* help you understand some of the principles Cytoflow is based on and some subtler points about using it to analyze flow cytometry data. They can help you use Cytoflow more effectively!

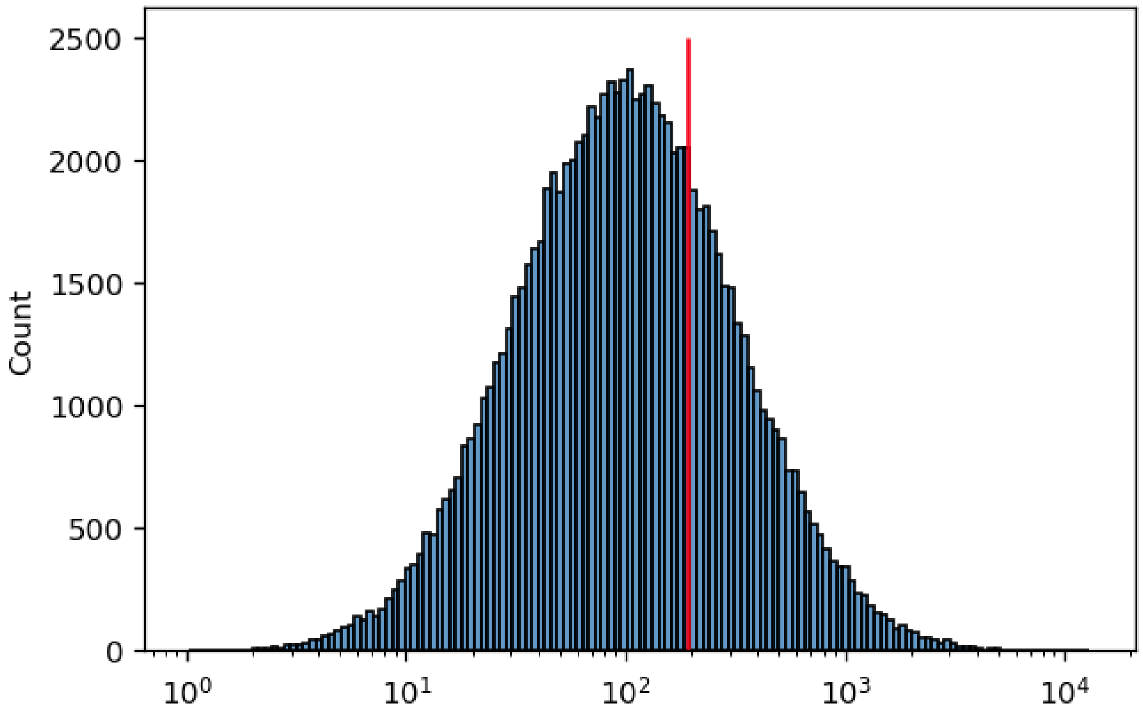
5.1.15 Guide: Which measure of center should I use?

When analyzing flow cytometry data, we often want to know the *center* of a distribution. For example, we may be treating our cells with a drug and want to be able to report that "fluorescence decreased XXX fold."

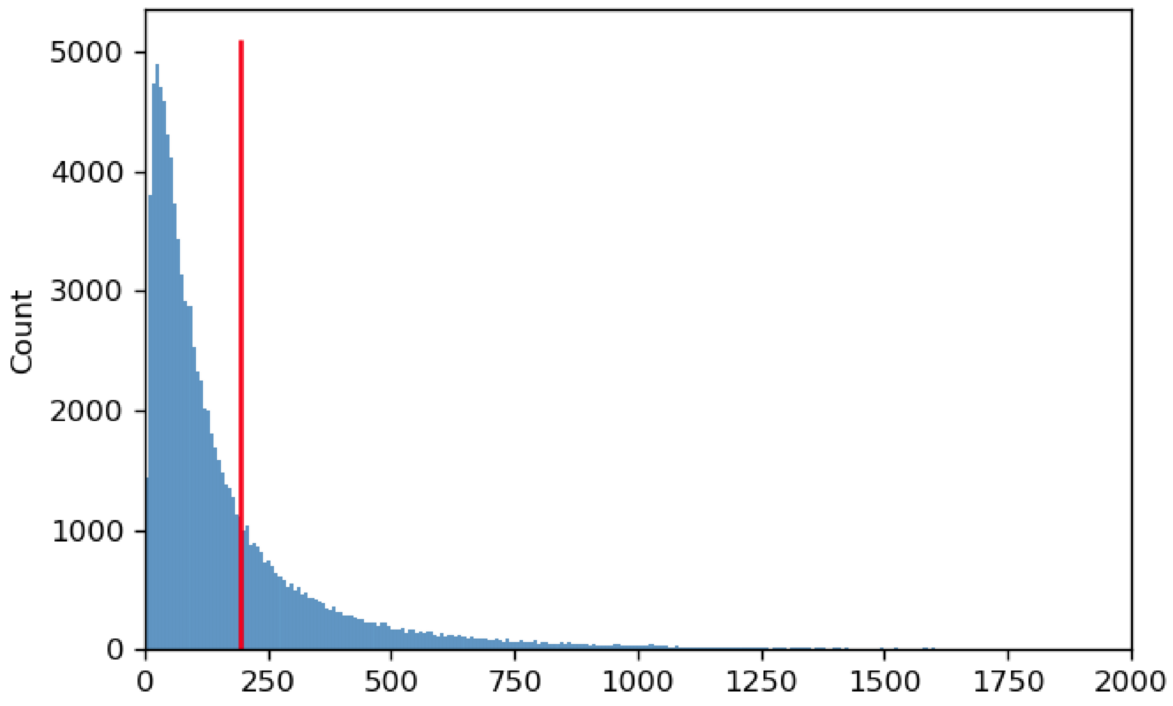
Often, we use the *arithmetic mean* to find the center. For example, in the histogram below, I've drawn 100,000 samples from a normal distribution with a mean of 1000 and a standard deviation of 100. The arithmetic mean is 999.40; I've drawn a red line on the plot at this position.



It's pretty clear that this is a good measure of this distribution's center. However, in cytometry, we often encounter distributions that are symmetrical *on a log scale*. This is the case below – and again, I've drawn a red line at the arithmetic mean of this distribution.

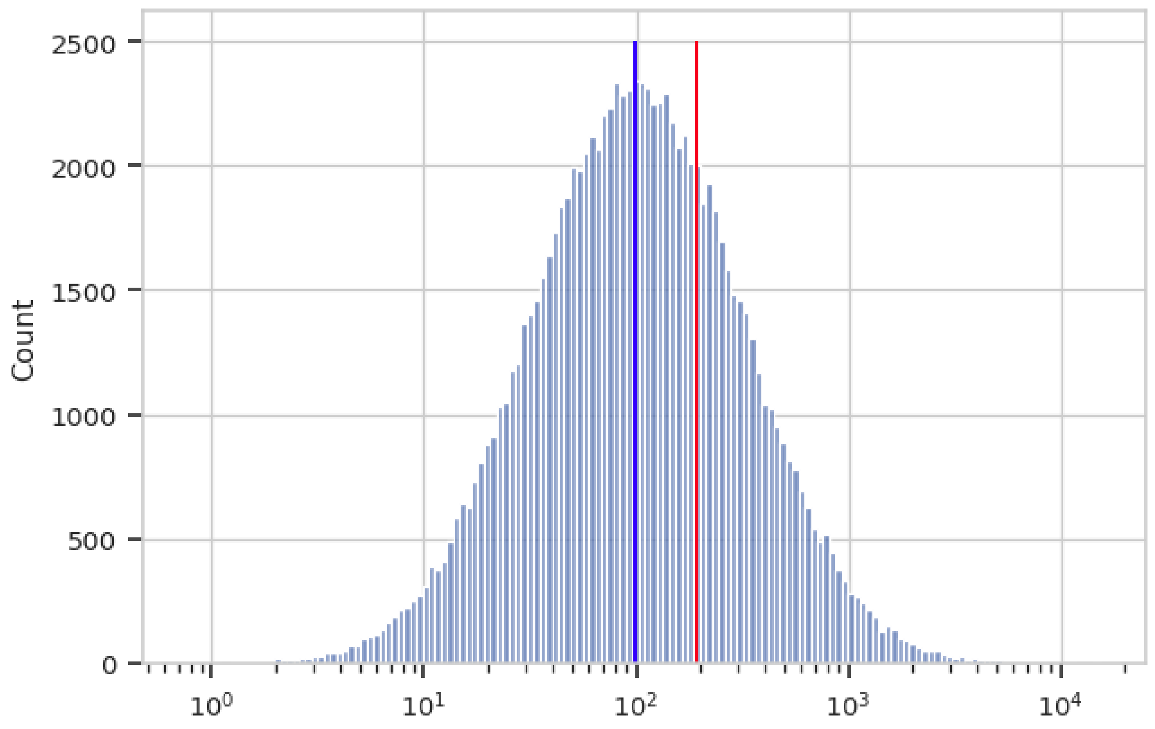


In this case, it's pretty clear that the arithmetic mean is *not* a good measure of this distribution's "center". It's easy to see why this is happening if we plot the *same* data on a *linear* scale instead:



The reason that the *arithmetic* mean isn't a good measure of center is because on a linear scale, *the distribution is not symmetric*. Instead, the mean is being "dragged" up by the larger values in the "tail" of the distribution.

A better measure of center for this kind of distribution is the *geometric mean*. Let's say we have n values. Instead of adding the values together and dividing by n (arithmetic mean), to find the geometric mean we multiply all the numbers together and then take the nth root. Below, you can see the log-scaled data again, with the arithmetic mean in red and the geometric mean in blue. It's clear that for data of this sort, the geometric mean is a better measure of "center".



`Cytoflow` implements both a geometric mean (to measure center) and a geometric "standard deviation" to measure spread. I highly recommend you use these instead of their arithmetic brethren when analyzing data that appears normal (or at least symmetric) on a logarithmic scale.

PS - why is so much biology log-normal? My colleague Jacob Beal has done some theoretical work on the topic. You can read his paper here:

Biochemical complexity drives log-normal variation in genetic expression. Jacob Beal, Engineering Biology, 1.1 (2017), pp. 55-60, July 2017. https://digital-library.theiet.org/content/journals/10.1049/enb.2017.0004

5.1.16 Guide: Using statistics to summarize data

Summarizing data is a key step in flow cytometry. Even simple, "traditional" analyses involve drawing gates and counting the number of events in them.

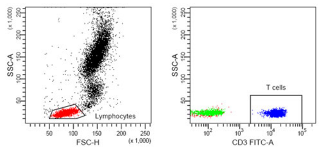
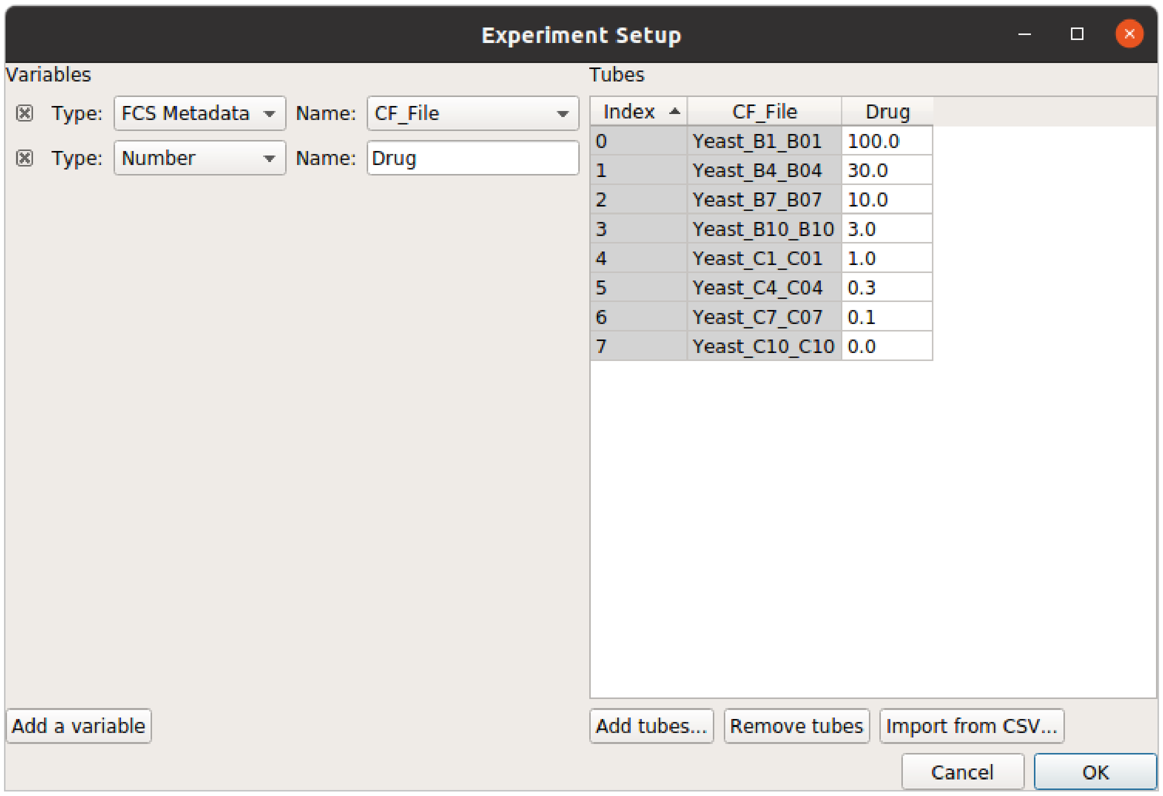


Fig. 1: What proportion of these events were T-cells?

`Cytoflow` calls these summary values – such as the mean or count of a set of events – *statistics*. Here, I'm using the word "statistic" in the technical sense, as in "a quantity that is computed from a sample."

Another key insight is that we are usually interested in *how a statistic changes* across our experiment. For example, let's say that I have some cells that express GFP, and I want to know how the amount of GFP expression changes as I alter the amount of a small molecule that I treat my cells with. I can create a *statistic*, then plot that statistic, to answer my question. Let's see how I might do so. (I'm using data from the *examples-basic* directory – feel free to follow along.)

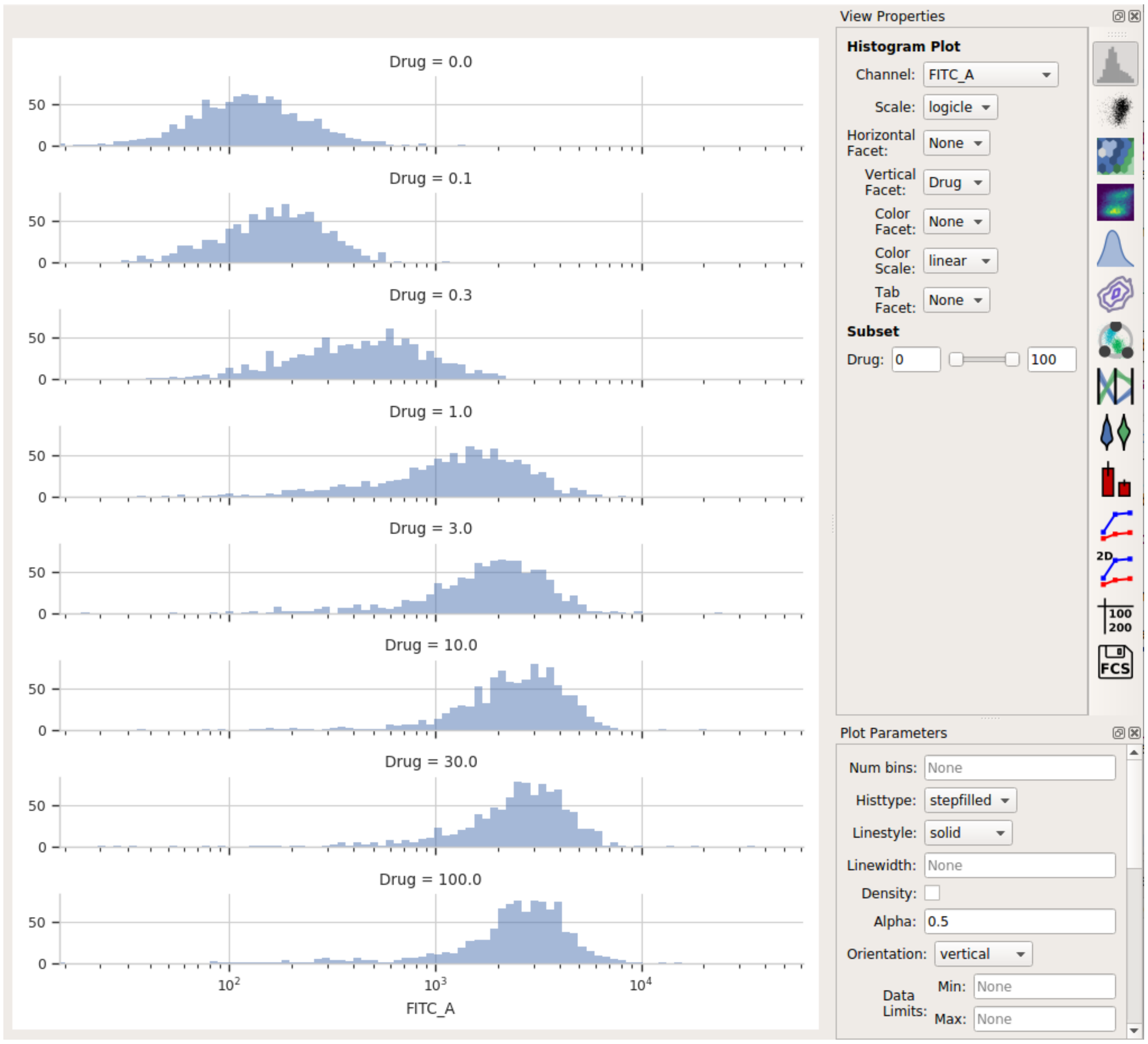
First, I need to import my data. In my experiment setup (in the **Import Data** operation), I must specify the *conditions* for each tube – that is, how the cells in each tube are different. In this case, I treated each tube with a different concentration of my drug, so I use "Drug" as a condition.



Type:	FCS Metadata	Name:	CF_File
Type:	Number	Name:	Drug

Index	CF_File	Drug
0	Yeast_B1_B01	100.0
1	Yeast_B4_B04	30.0
2	Yeast_B7_B07	10.0
3	Yeast_B10_B10	3.0
4	Yeast_C1_C01	1.0
5	Yeast_C4_C04	0.3
6	Yeast_C7_C07	0.1
7	Yeast_C10_C10	0.0

Using a histogram (and a vertical *facet*), I can see that the GFP intensity (as displayed in the FITC-A channel) is in fact changing when I vary the amount of drug I treat the cells with.



Histogram Plot

Channel: FITC_A

Scale: logicle

Horizontal Facet: None

Vertical Facet: Drug

Color Facet: None

Color Scale: linear

Tab Facet: None

Subset

Drug: 0 100

Num bins: None

Histtype: stepfilled

Linestyle: solid

Linewidth: None

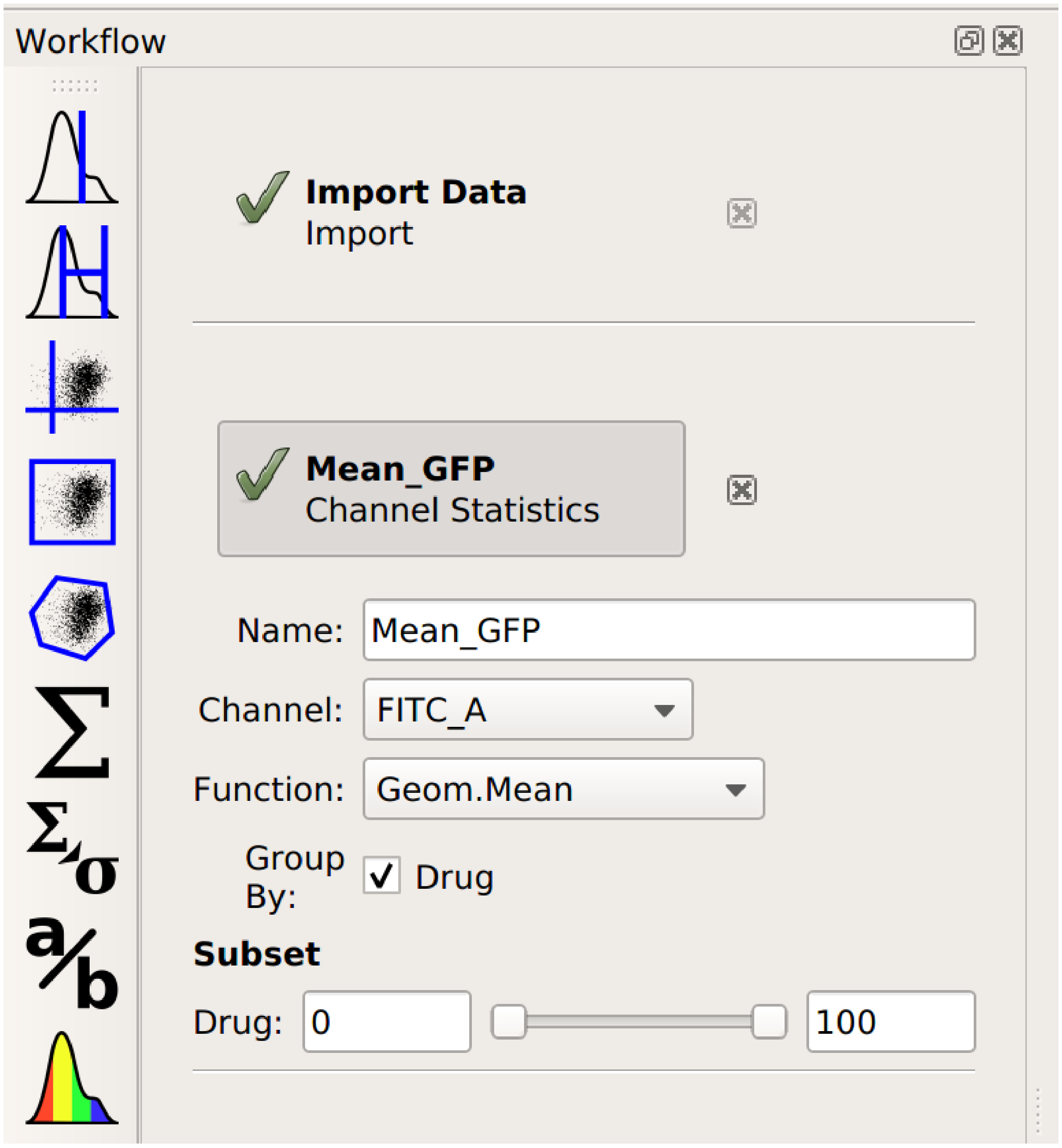
Density: ☐

Alpha: 0.5

Orientation: vertical

Data Limits: Min: None Max: None

Each distribution seems pretty symmetrical when plotted on a logarithmic scale, so let's use a *geometric mean* to summarize them. I can do that with a **Channel Statistics** operation (the one whose button is a big sigma.)



Name: Mean_GFP
Channel: FITC_A
Function: Geom.Mean
Group By: ✓ Drug
Subset
Drug: 0 100

Note that I've set a name for the statistic, the channel I want to summarize, and the function I want to apply. Probably the most important – and most confusing! – parameter is **By**. This specifies how I want to `Cytoflow` to group the data before applying **Function** to channel – in this case, I've set it to *Drug*. Here's the order that things happen in:

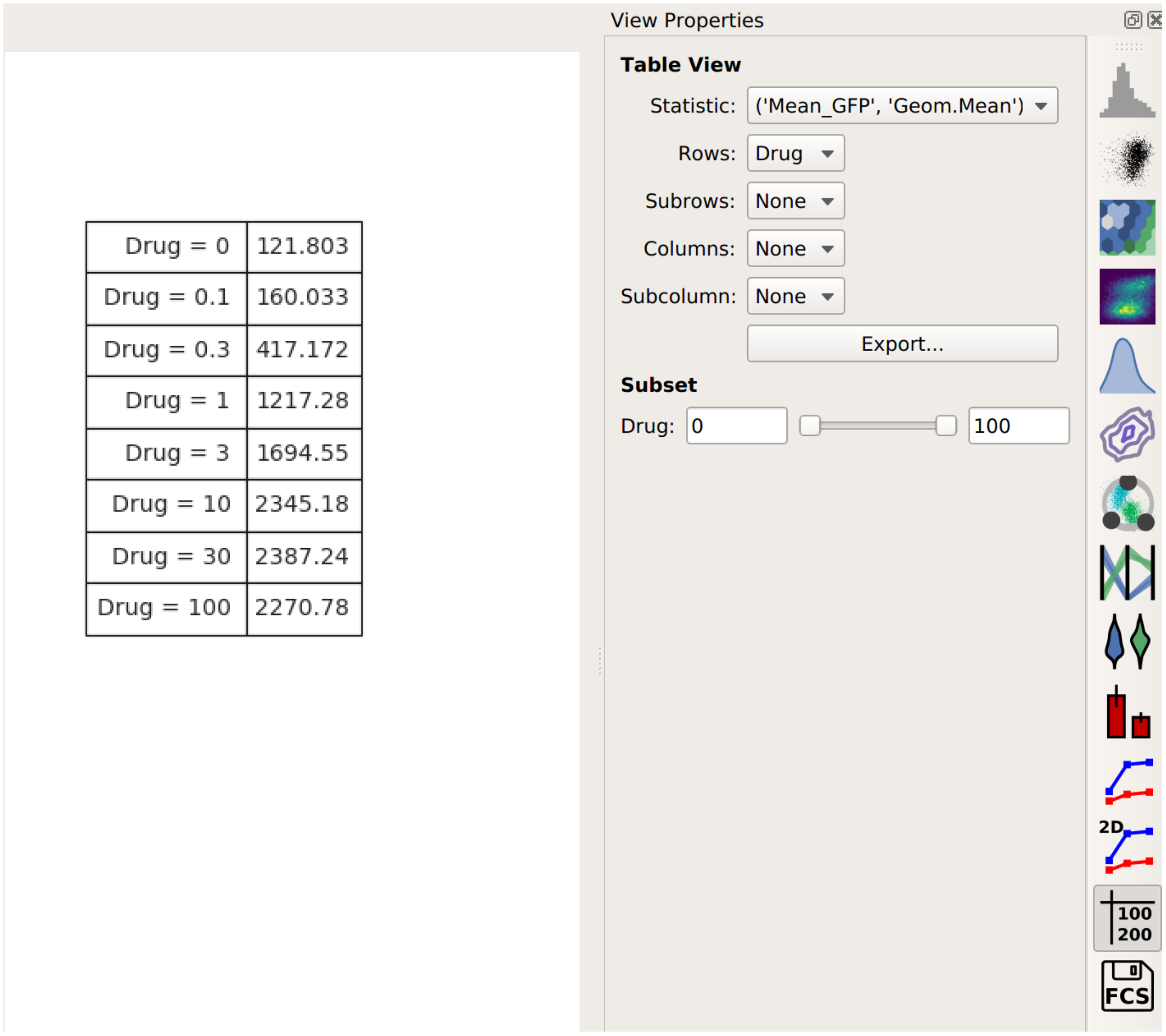
# Cytoflow sees how many different values of the *Drug* parameter are in the data set.

# It separates the data into groups – subsets – by those different values of *Drug*. If each tube has a different *Drug* value, then each of those subsets is the events from a single tube. However, *if I had multiple tubes with the same* **Drug* value, those tubes would be combined.*

# For each subset of the data – each unique value of *Drug* – `Cytoflow` applies the function I asked for (in this case, *Geom.Mean* – the geometric mean – to the channel I said – in this case, *FITC_A*.

This results in a table of those summary numbers for each subset. You can view this table directly (with the

Table View, natch) – here's what we see.



Drug = 0	121.803
Drug = 0.1	160.033
Drug = 0.3	417.172
Drug = 1	1217.28
Drug = 3	1694.55
Drug = 10	2345.18
Drug = 30	2387.24
Drug = 100	2270.78

View Properties

Table View

Statistic: ('Mean_GFP', 'Geom.Mean')

Rows: Drug

Subrows: None

Columns: None

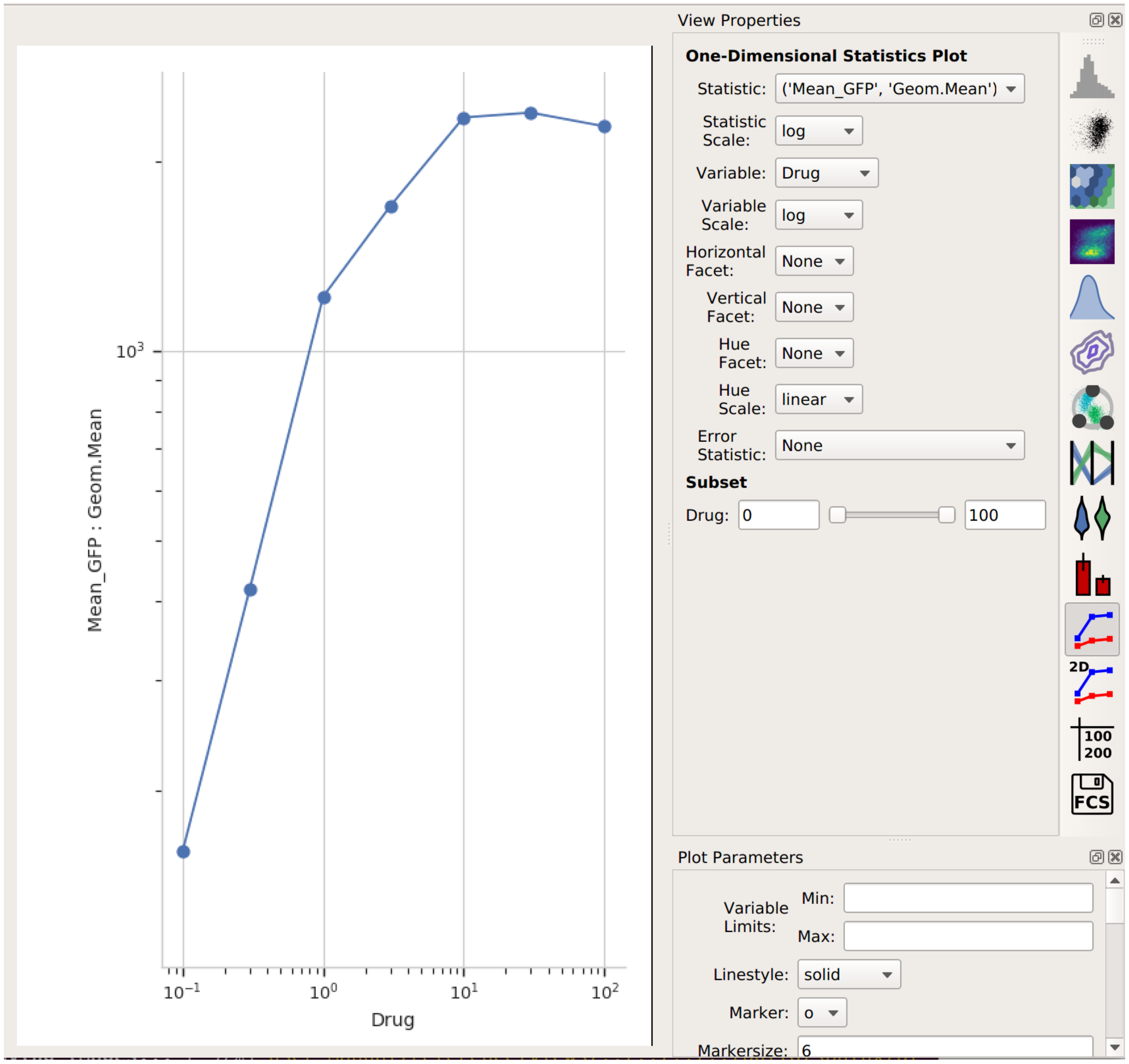
Subcolumn: None

Export...

Subset

Drug: 0 100

If you would like to export this data, you can of course do so with the **Export** button. However, `Cytoflow` can plot it directly as well, with the **1D Statistics View**.



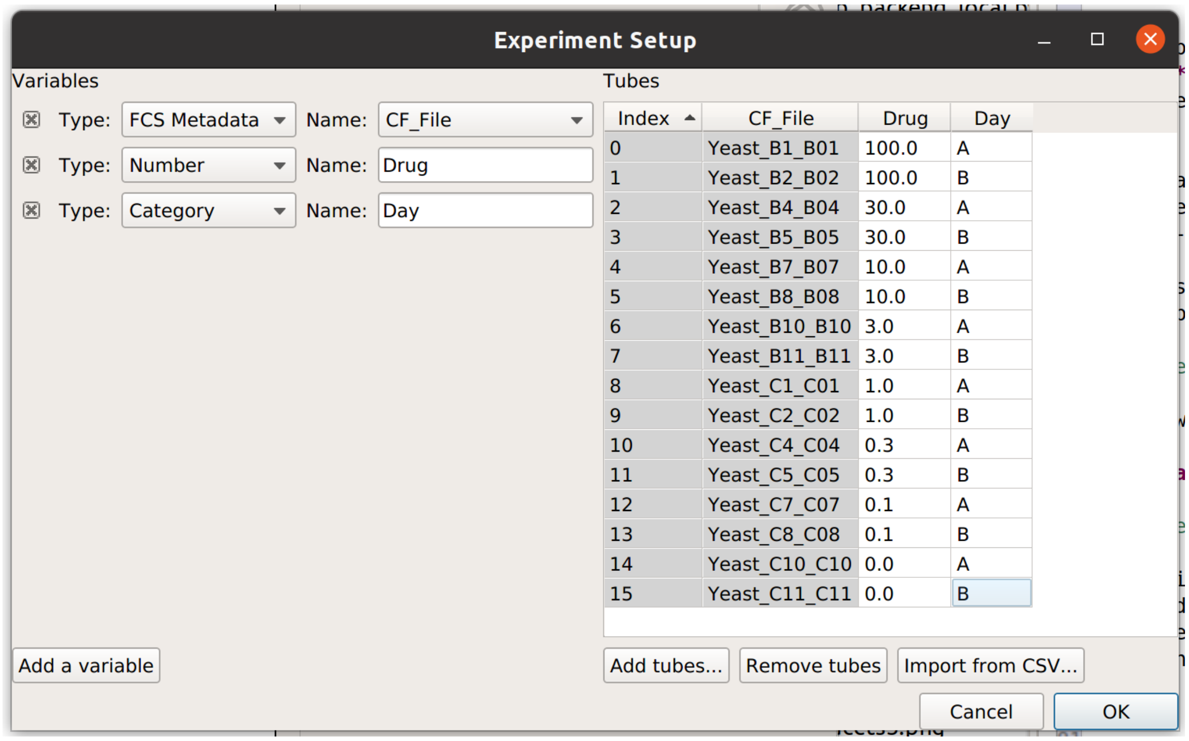
Statistic: ('Mean_GFP', 'Geom.Mean')
Statistic Scale: log
Variable: Drug
Variable Scale: log
Horizontal Facet: None
Vertical Facet: None
Hue Facet: None
Hue Scale: linear
Error Statistic: None

Drug: 0 100

Variable Limits: Min: Max:
Linestyle: solid
Marker: o
Markersize: 6

In setting up this plot, I selected the statistic to plot and the variable I wanted on the X axis. I also changed the statistic scale (to *log*) and the variable scale (to *logicle*), which makes things easier to interpret. (Note that if I had left the variable scale on *log*, it would not have plotted the "Drug = 0" condition, because log(0) is undefined!

It's clear from this analysis that the mean GFP value increases as the amount of drug increases. However, statistics are more powerful than this because they can capture multiple variables at the same time. For example, let's imagine that I did two different replicates on two different days:



Variables

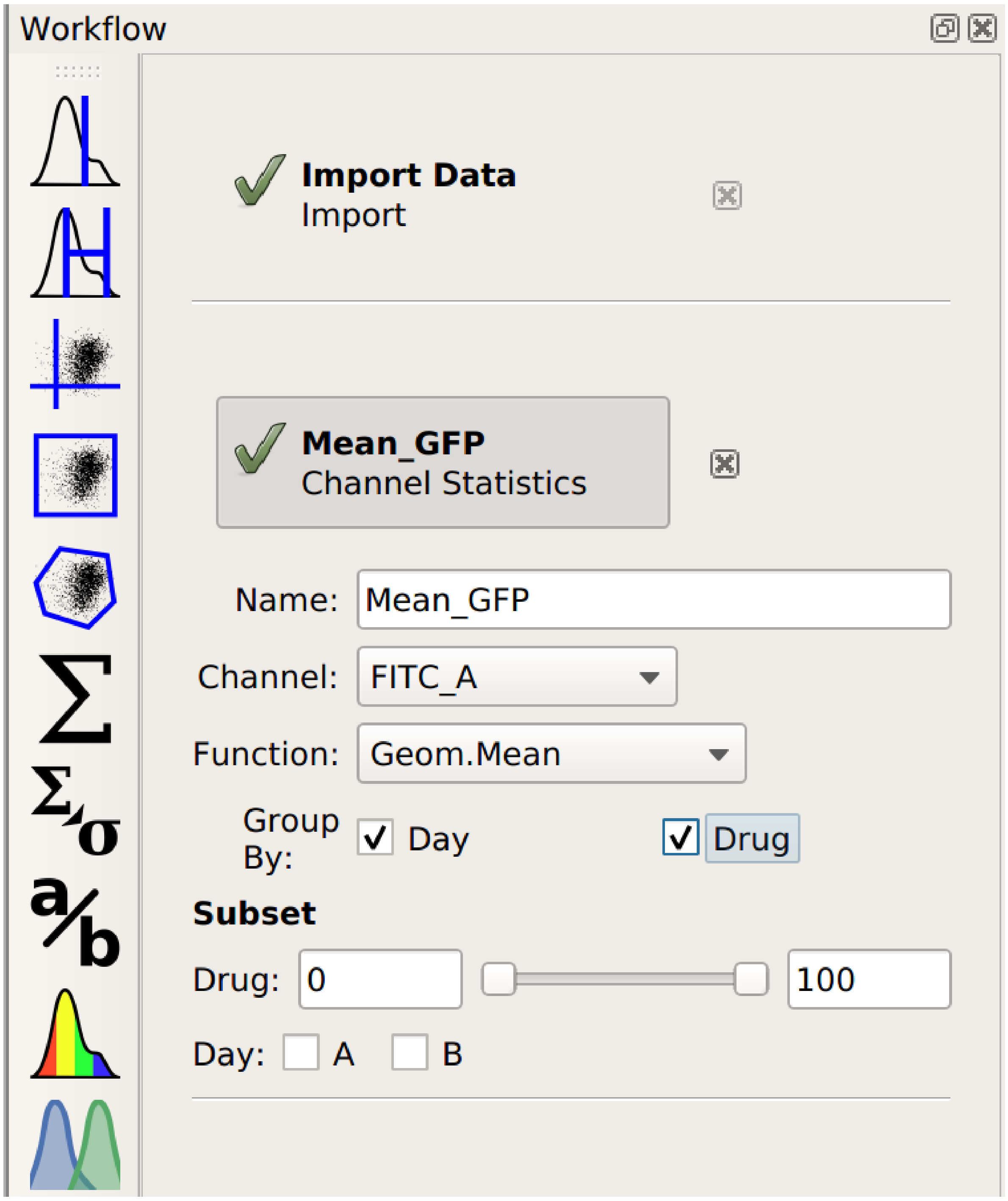
Type: FCS Metadata	Name: CF_File	
Type: Number	Name: Drug	
Type: Category	Name: Day	

Tubes

Index	CF_File	Drug	Day
0	Yeast_B1_B01	100.0	A
1	Yeast_B2_B02	100.0	B
2	Yeast_B4_B04	30.0	A
3	Yeast_B5_B05	30.0	B
4	Yeast_B7_B07	10.0	A
5	Yeast_B8_B08	10.0	B
6	Yeast_B10_B10	3.0	A
7	Yeast_B11_B11	3.0	B
8	Yeast_C1_C01	1.0	A
9	Yeast_C2_C02	1.0	B
10	Yeast_C4_C04	0.3	A
11	Yeast_C5_C05	0.3	B
12	Yeast_C7_C07	0.1	A
13	Yeast_C8_C08	0.1	B
14	Yeast_C10_C10	0.0	A
15	Yeast_C11_C11	0.0	B

In this case, I have imported twice as many tubes, and labeled them with *both* the amount of drug I used *and* which day I did the experiment on.

Now, when I set up my **Channel Statistic**, I'll set **Group By** to *both Day and Drug*.


Workflow

Import Data
Import

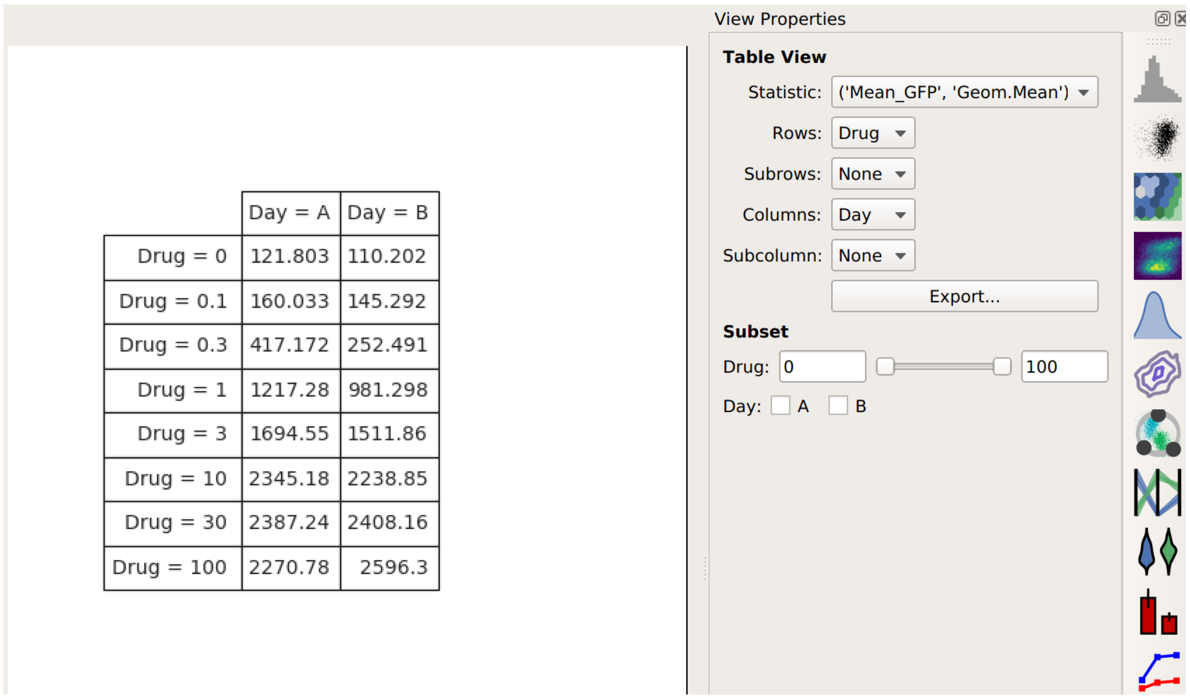
Mean_GFP
Channel Statistics

Name: Mean_GFP
Channel: FITC_A
Function: Geom.Mean
Group By: ✓ Day ✓ Drug

Subset

Drug: 0 100
Day: ☐ A ☐ B

Let's see what this does to my table:



	Day = A	Day = B
Drug = 0	121.803	110.202
Drug = 0.1	160.033	145.292
Drug = 0.3	417.172	252.491
Drug = 1	1217.28	981.298
Drug = 3	1694.55	1511.86
Drug = 10	2345.18	2238.85
Drug = 30	2387.24	2408.16
Drug = 100	2270.78	2596.3

View Properties

Table View

Statistic: ('Mean_GFP', 'Geom.Mean')

Rows: Drug

Subrows: None

Columns: Day

Subcolumn: None

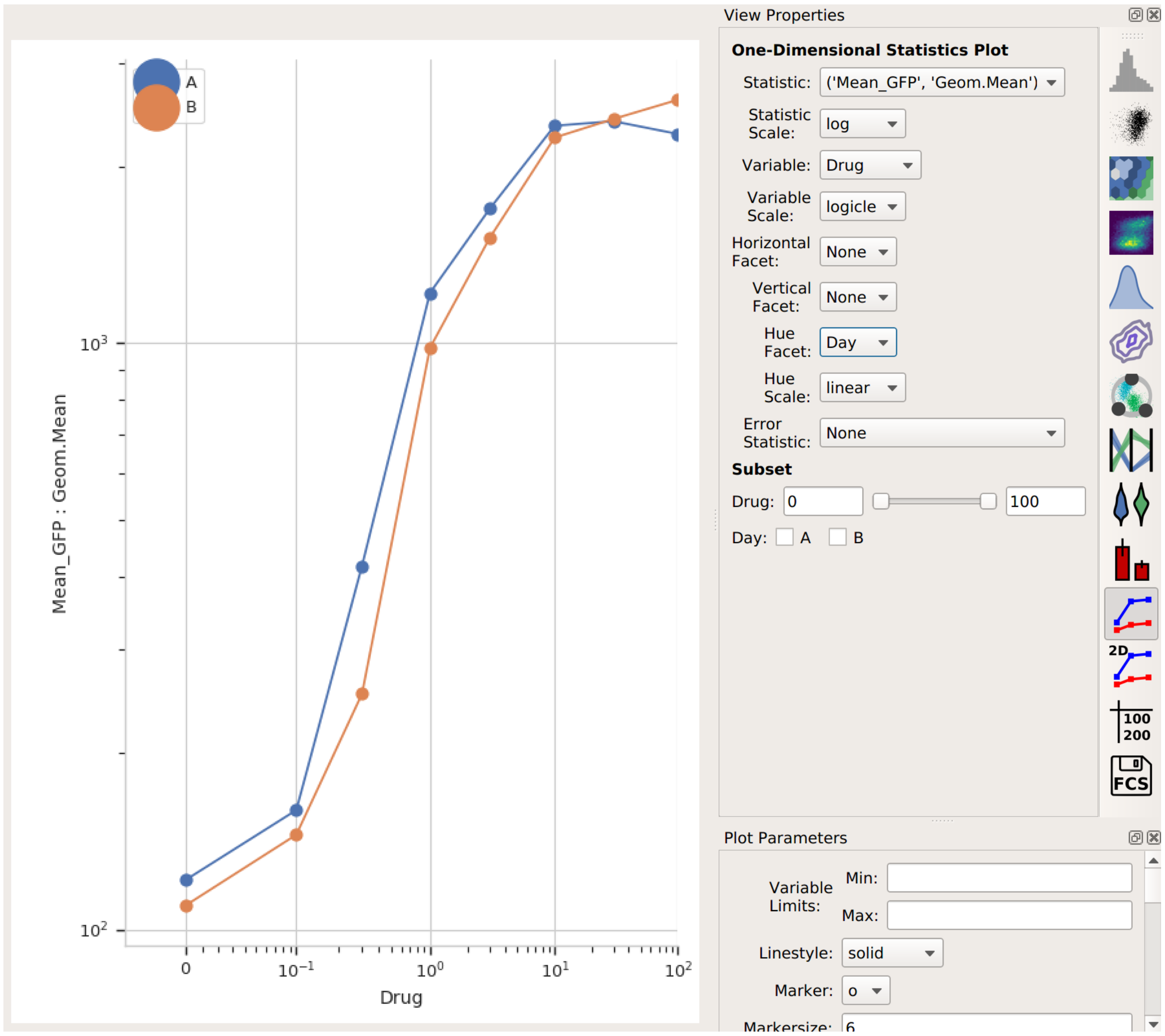
Subset

Drug: 0 100

Day: A B

Note that I now have *two* conditions that I can use when making my table: **Drug** and **Day**. This is because the channel statistic operation computed a geometric mean for *each unique combination of Drug and Day values.* So instead of 8 means, now I have 16. Here, I've configured the table view to show different amounts of drug on different rows and different days in different columns.

Again, I could export this if I wanted – or I could plot it. Let's make another **1D Statistics View** plot, putting the two different days in different colors:



One-Dimensional Statistics Plot

Statistic: ('Mean_GFP', 'Geom.Mean')
Statistic Scale: log
Variable: Drug
Variable Scale: logicle
Horizontal Facet: None
Vertical Facet: None
Hue Facet: Day
Hue Scale: linear
Error Statistic: None

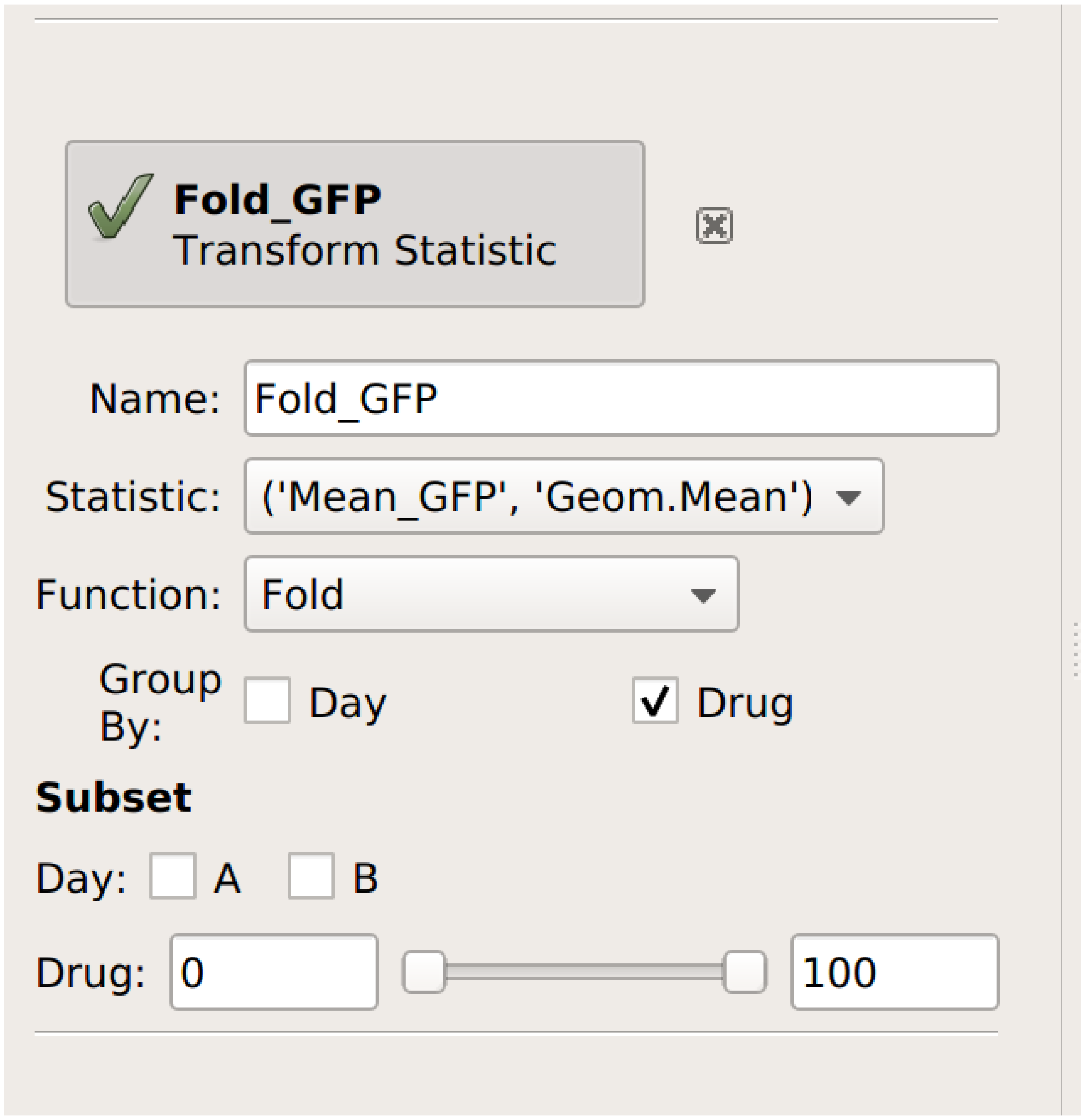
Subset

Drug: 0 100
Day: A B

Variable Limits: Min: Max:
Linestyle: solid
Marker: o
Markersize: 6

Note that the "Day A" values are pretty consistently higher than "Day B". I wonder how much higher, and how consistently so? We can answer this question with another operation, called **Transform Statistic**. This works similarly to **Channel Statistic**, in that it groups things together and applies a function. However, instead of grouping together events from the flow cytometer, it groups together *values in another statistic* before applying the function. (This way, it *transforms* that statistic – see?)

This time, I'll group by **Drug** (and *not* **Day**). Remember, this will take the starting statistic, split it into groups for each unique value of **Drug**, and then apply the **Fold** function to each group. (Fold simply divides every value in the group by the minimum value.)



Fold_GFP
Transform Statistic

Name: Fold_GFP

Statistic: ('Mean_GFP', 'Geom.Mean')

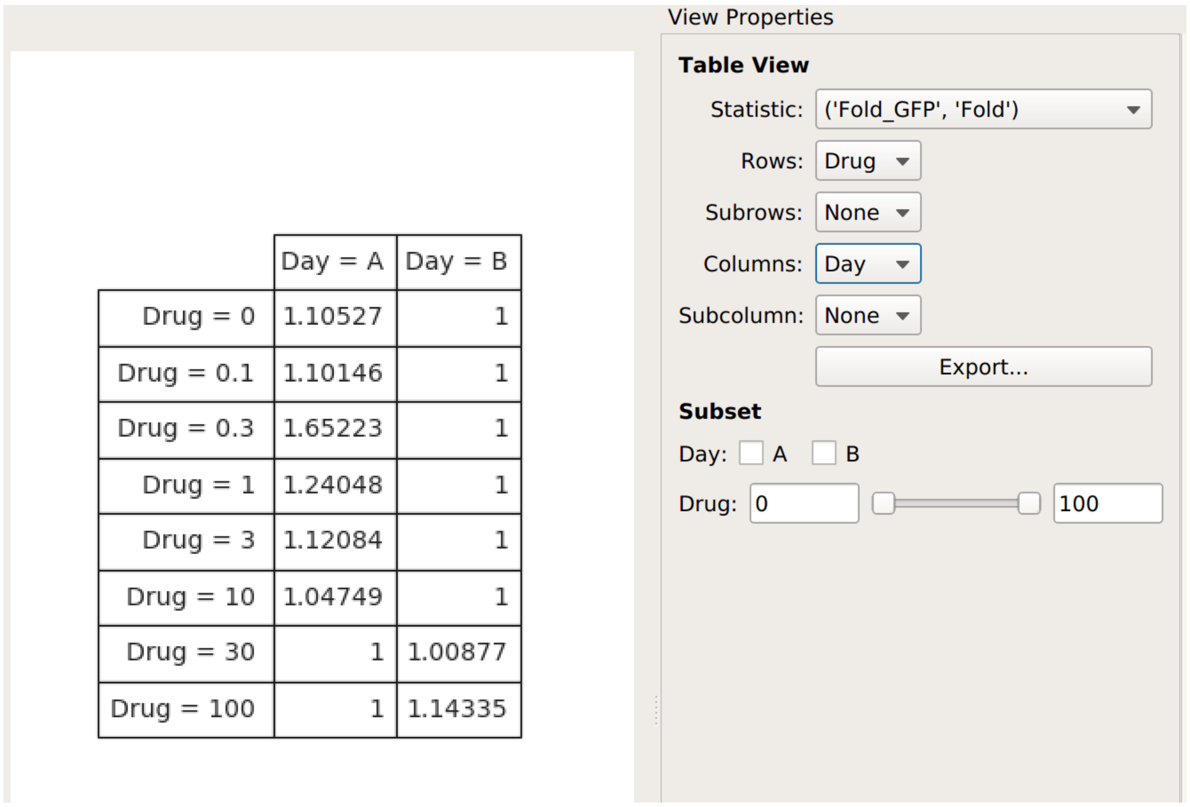
Function: Fold

Group By: [ ] Day [✓ Drug]

Subset

Day: [ ] A [ ] B

Drug: 0 100



	Day = A	Day = B
Drug = 0	1.10527	1
Drug = 0.1	1.10146	1
Drug = 0.3	1.65223	1
Drug = 1	1.24048	1
Drug = 3	1.12084	1
Drug = 10	1.04749	1
Drug = 30	1	1.00877
Drug = 100	1	1.14335

View Properties

Table View

Statistic: ('Fold_GFP', 'Fold')

Rows: Drug

Subrows: None

Columns: Day

Subcolumn: None

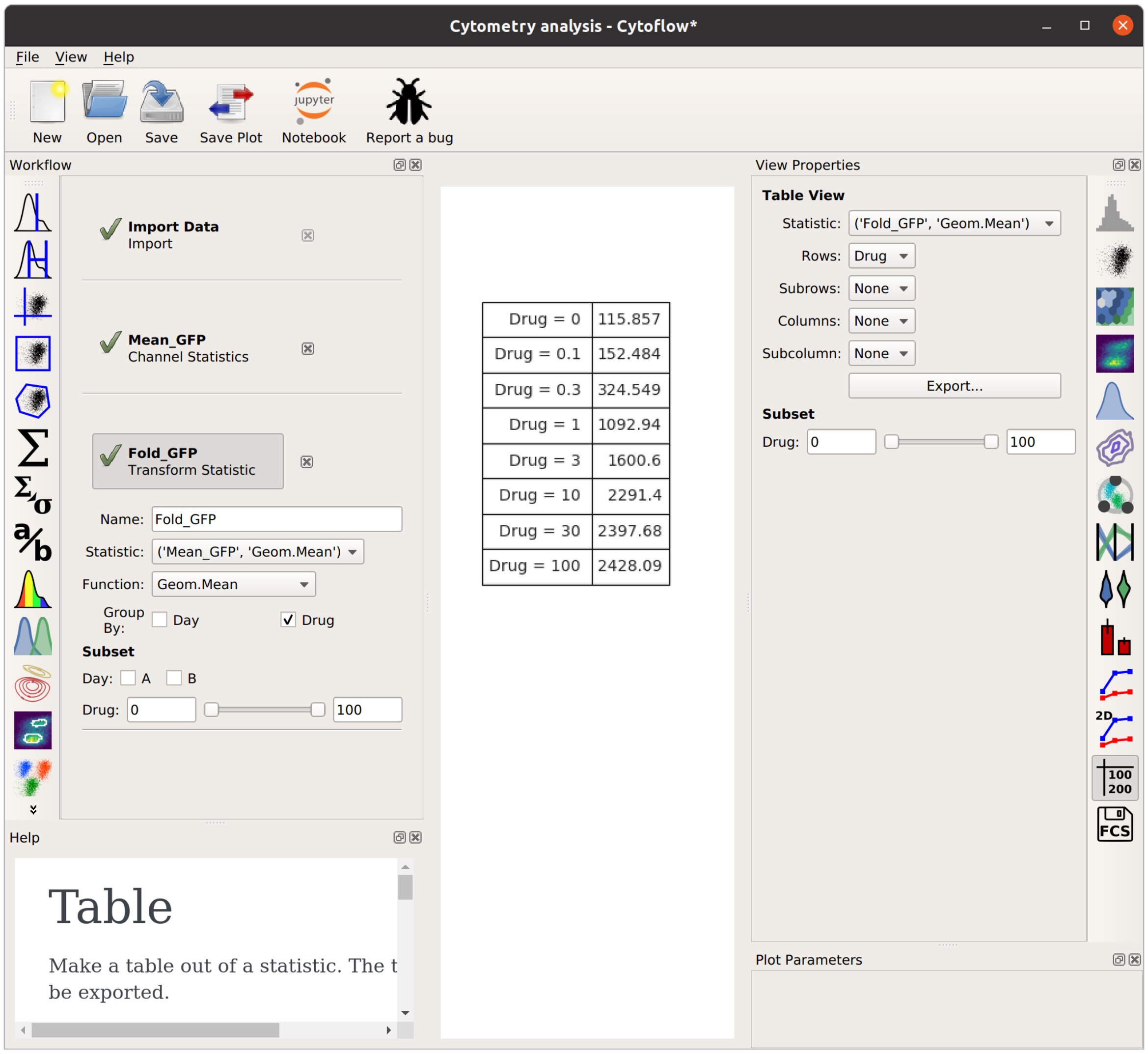
Export...

Subset

Day: ☐ A ☐ B

Drug: 0 100

The **Fold** function produces a statistic the same size and shape as the one it's operating on. However, some functions *reduce* the size of the statistic – for example, if we apply **Geom.Mean** again, we only get a table with one column (because it's taking the geometric mean of all the values in each group):



Import Data
Import

Mean_GFP
Channel Statistics

Fold_GFP
Transform Statistic

Name: Fold_GFP
Statistic: ('Mean_GFP', 'Geom.Mean')
Function: Geom.Mean
Group By: ☐ Day ☑ Drug
Subset
Day: ☐ A ☐ B
Drug: 0 100

Drug = 0	115.857
Drug = 0.1	152.484
Drug = 0.3	324.549
Drug = 1	1092.94
Drug = 3	1600.6
Drug = 10	2291.4
Drug = 30	2397.68
Drug = 100	2428.09

Table View
Statistic: ('Fold_GFP', 'Geom.Mean')
Rows: Drug
Subrows: None
Columns: None
Subcolumn: None
Subset
Drug: 0 100

Table

Make a table out of a statistic. The t
be exported.

Statistics are a pretty key part of the way `Cytoflow` is meant to be used. I hope this explanation made sense – if you feel it can be improved, please feel free to submit a bug (or, even better, a patch or pull request) to improve it.

5.1.17 Guide: Using *facets* to compare data sets

`Cytoflow` uses an idea called *facets* to help you analyze data. It’s taken from Tufte’s trellis plots, which is a fancy term for a simple idea: that to *compare* two data sets, you should plot them next to eachother using the same scale and axes:

Let’s say I have two different FCS files. As I imported them, I created a `Sample` variable and assigned the tubes `Sample_1` and `Sample_2`, respectively. Now I want to compare them using a histogram plot. Remember that by default, `Cytoflow` shows *all* of the data, *from both of the tubes*, at the same time on the plot:

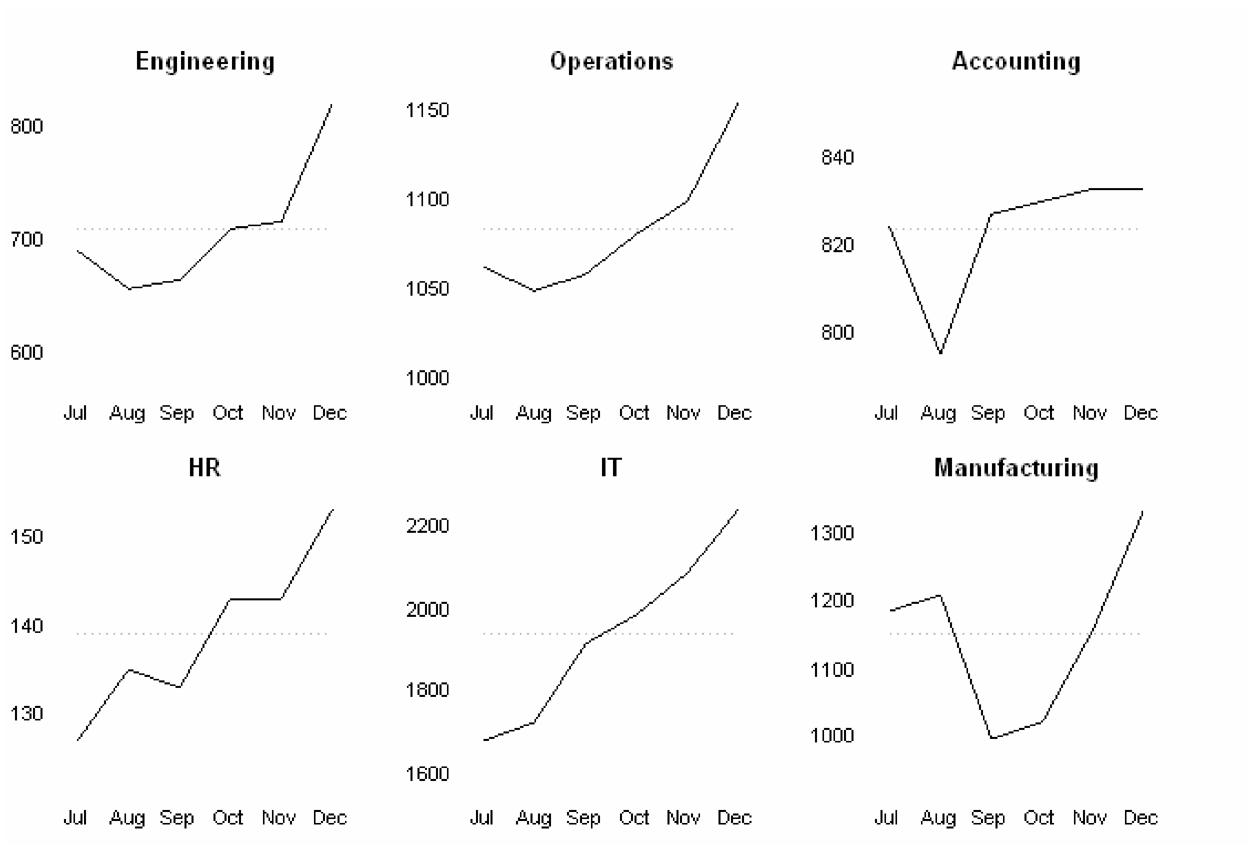
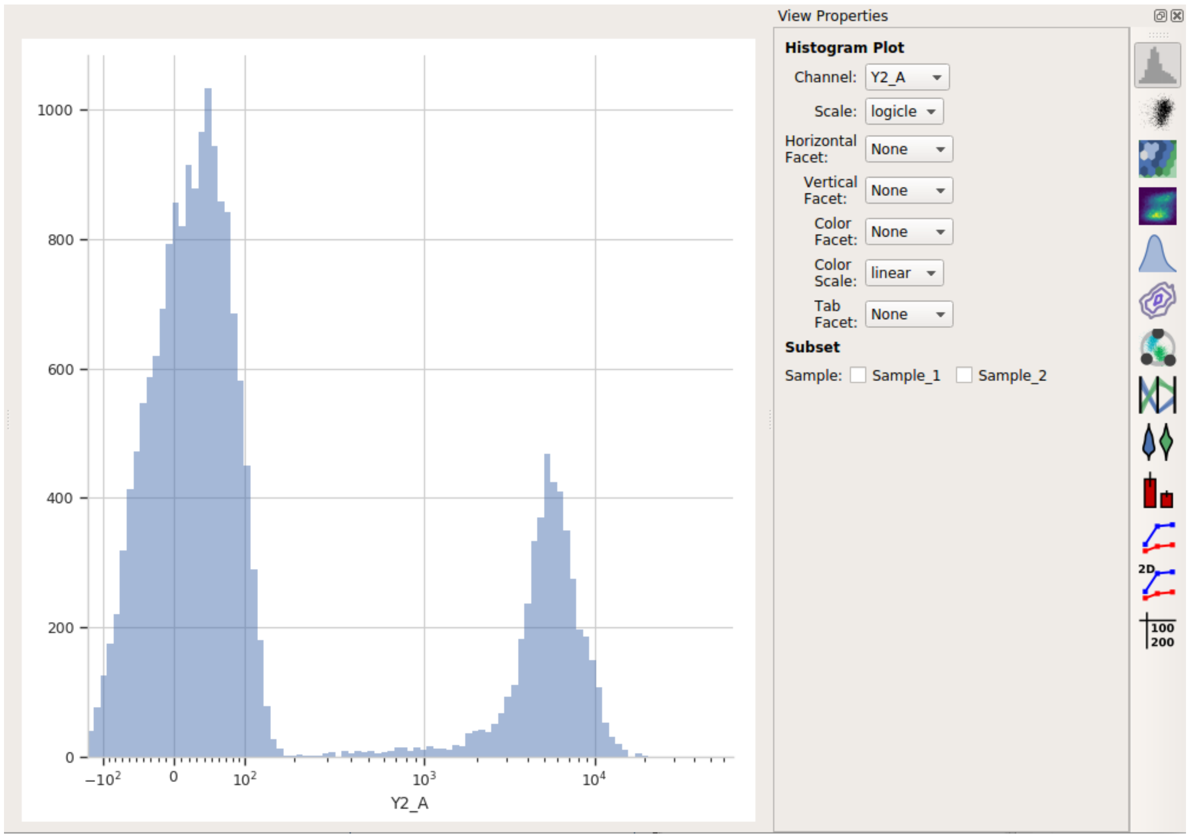


Fig. 2: This figure shows multiple line plots on the same X axes, making it easy to compare between them. (Image by Wikipedia user `Chrispounds`)



Histogram Plot

Channel: Y2_A

Scale: logicle

Horizontal Facet: None

Vertical Facet: None

Color Facet: None

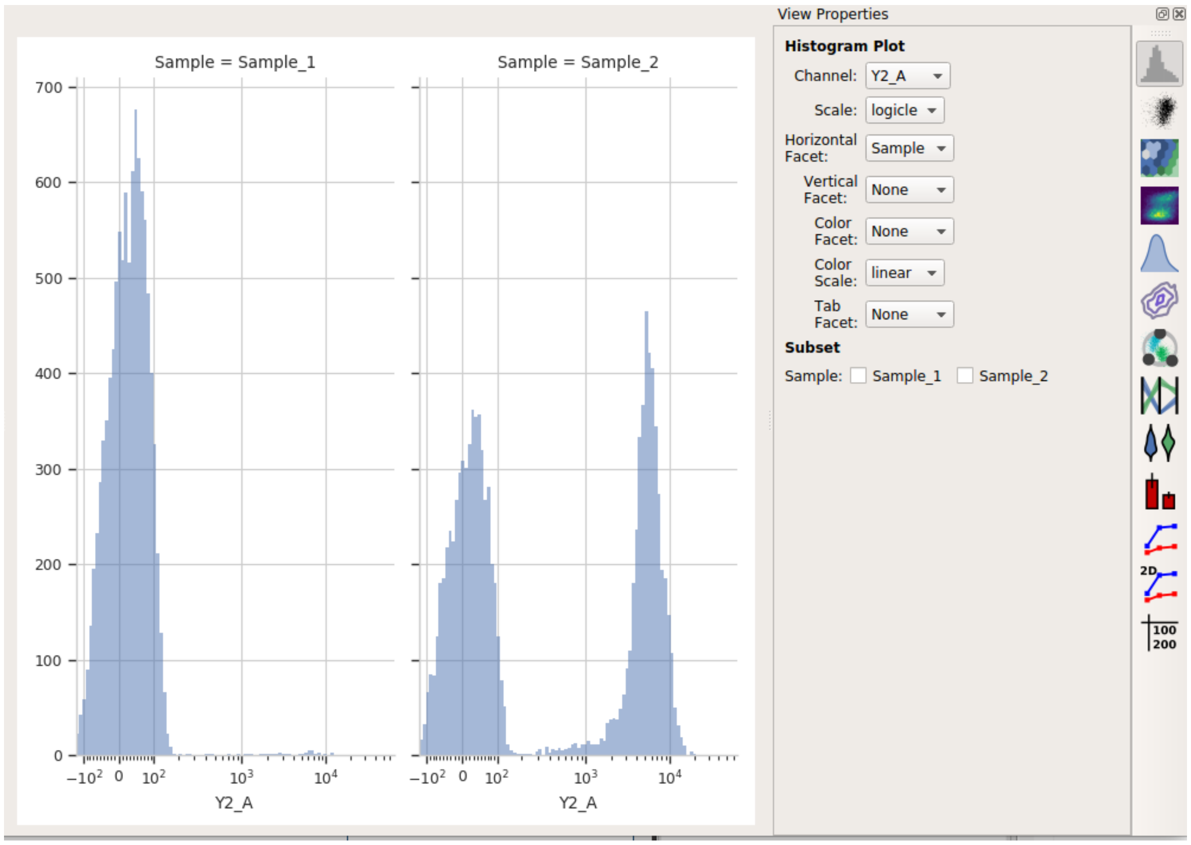
Color Scale: linear

Tab Facet: None

Subset

Sample: ☐ Sample_1 ☐ Sample_2

However, `Cytoflow` also makes it easy to compare these two samples. I can put them on separate plots, side-by-side, by setting `Horizontal Facet` to the *variable* that I want to compare. In this case, it's `Sample`:



View Properties

Histogram Plot

Channel: Y2_A

Scale: logicle

Horizontal Facet: Sample

Vertical Facet: None

Color Facet: None

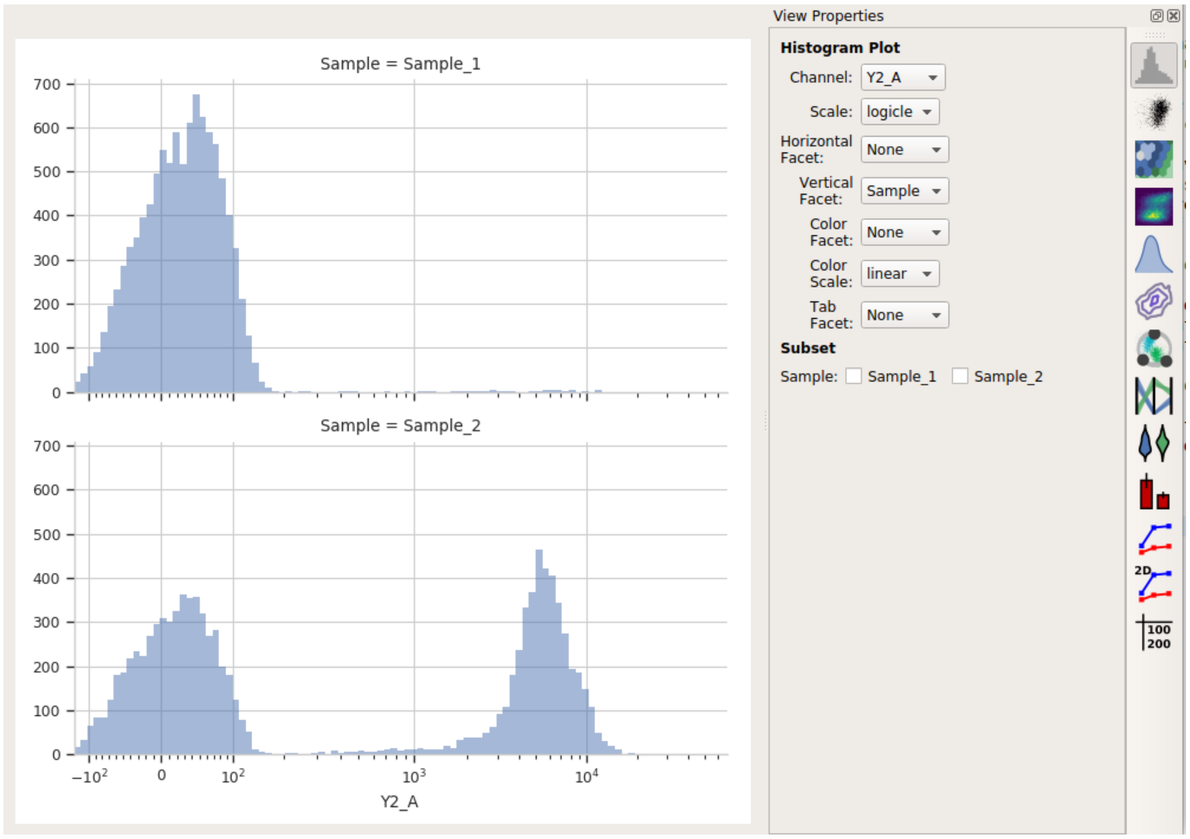
Color Scale: linear

Tab Facet: None

Subset

Sample: ☐ Sample_1 ☐ Sample_2

Note that the title of the plot shows me both which *variable* I'm comparing and what the *value* of that variable is in each plot. Similarly, I can create separate plots but stack them on top of eachother with the `Vertical Facet` setting:



Histogram Plot

Channel: Y2_A

Scale: logicle

Horizontal Facet: None

Vertical Facet: Sample

Color Facet: None

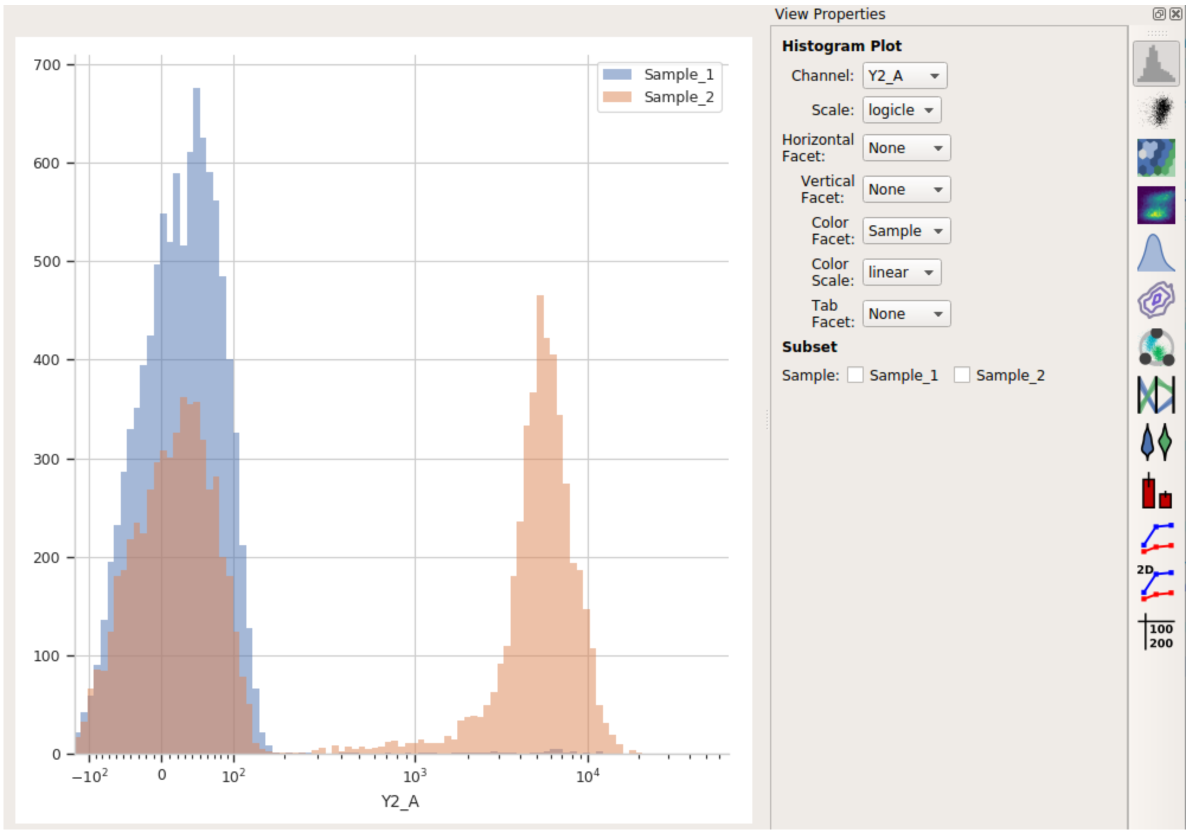
Color Scale: linear

Tab Facet: None

Subset

Sample: ☐ Sample_1 ☐ Sample_2

You can also view them on the same plot but with different colors using the `Color Facet` setting:



Histogram Plot

Channel: Y2_A

Scale: logicle

Horizontal Facet: None

Vertical Facet: None

Color Facet: Sample

Color Scale: linear

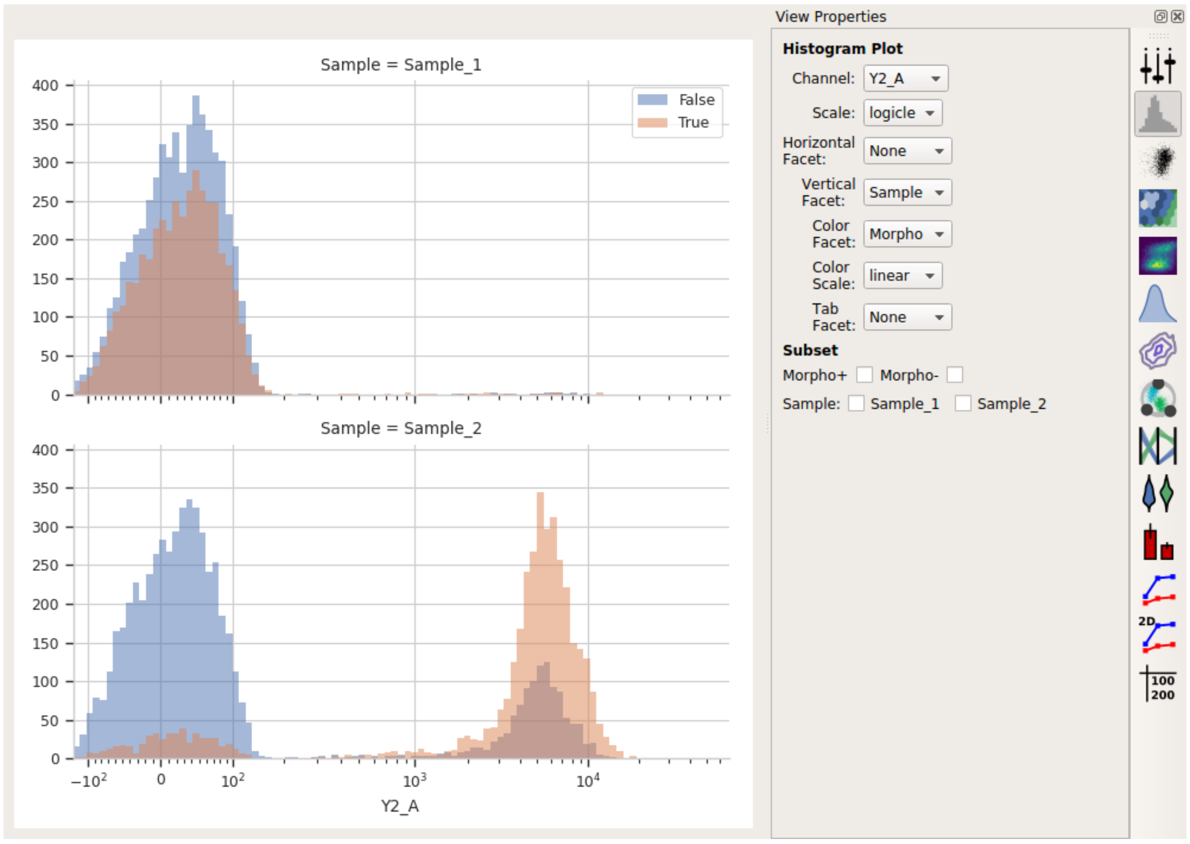
Tab Facet: None

Subset

Sample: ☐ Sample_1 ☐ Sample_2

(No, you can't change the colors, at least not in the point-and-click interface – they're the from default Seaborn qualitative palette. If you'd like to be able to change the colors, submit a feature request or (even better) a patch!)

Finally, if you have multiple variables, you can set multiple facets at once to compare across them all. For example, I've added a `Threshold` gate called `Morpho`, and now I can compare data across different values of both `Sample` and `Morpho`:

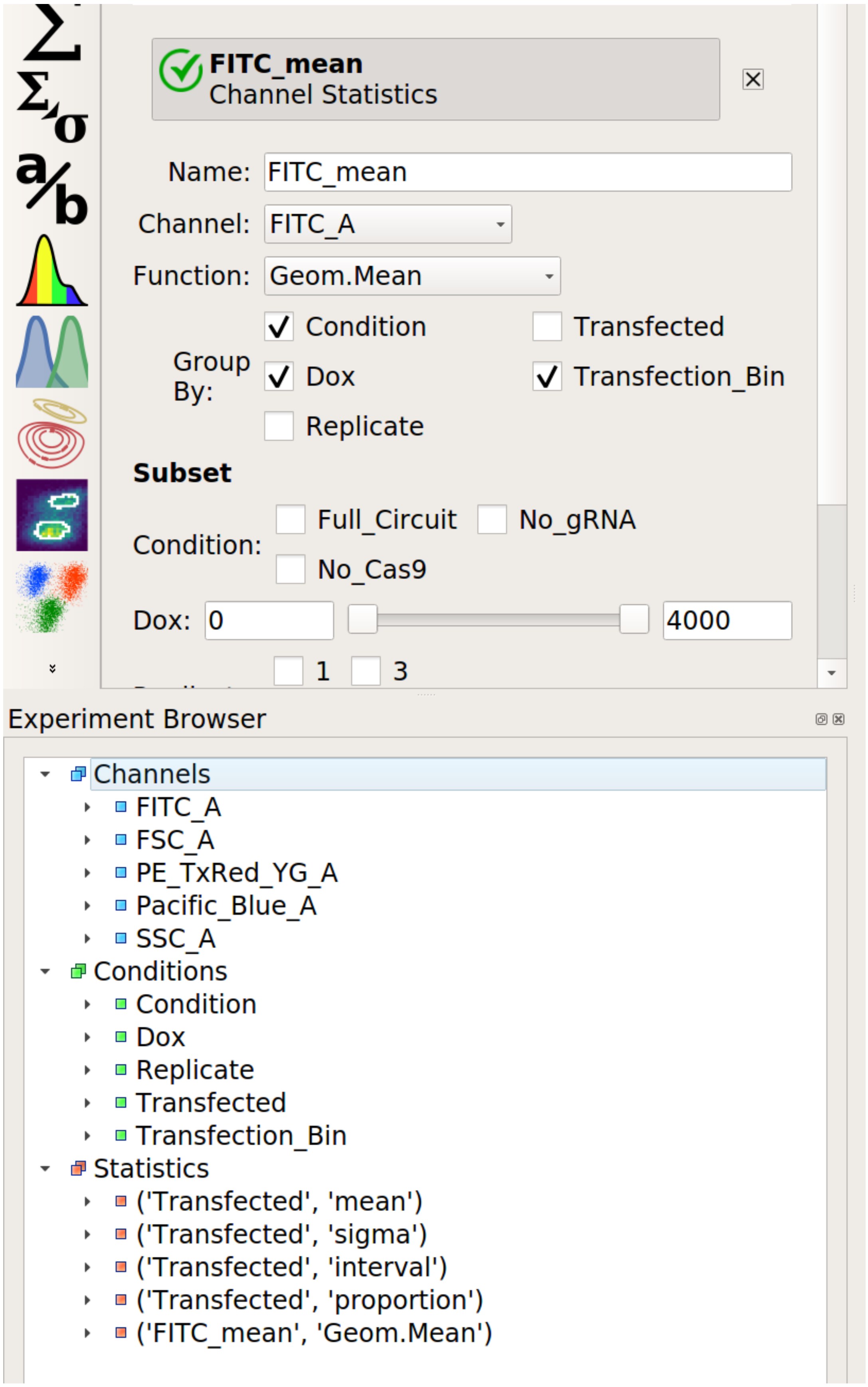


Histogram Plot
Channel: Y2_A
Scale: logicle
Horizontal Facet: None
Vertical Facet: Sample
Color Facet: Morpho
Color Scale: linear
Tab Facet: None
Subset
Morpho+ ☐ Morpho- ☐
Sample: ☐ Sample_1 ☐ Sample_2

5.1.18 Guide: How can I understand the experiment structure?

When creating a complex analysis, it can be easy to lose track of the channels, conditions, and statistics that you've created in your workflow. In order to give users both a quick overview of their experiment and allow them to "drill down" into their experiment structure, Cytoflow includes an **Experiment Browser**.

Consider the experiment outlined in *Tutorial: Synthetic Gene Logic Network* – five channels, three experimental conditions, two conditions added in the analysis, and a number of statistics. It's a lot to keep in your head! However, if you select the last operation in the workflow, the **Experiment Browser** shows this:



FITC_mean
Channel Statistics

Name: FITC_mean

Channel: FITC_A

Function: Geom.Mean

Group By: [✓ Condition] Transfected [✓ Dox] [✓ Transfection_Bin] Replicate

Subset

Condition: Full_Circuit No_gRNA No_Cas9

Dox: 0 4000

1 3

Experiment Browser

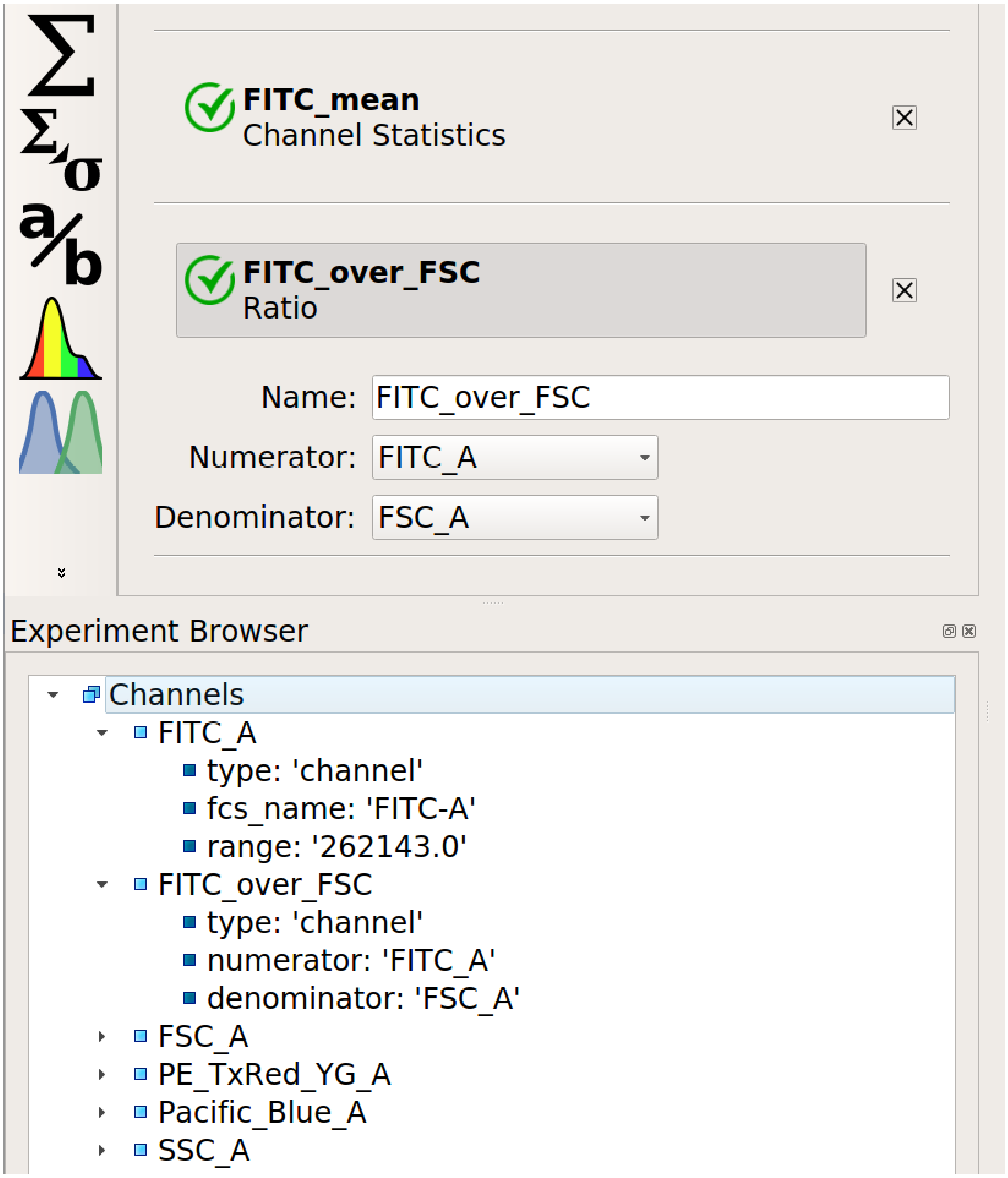
- Channels
  - FITC_A
  - FSC_A
  - PE_TxRed_YG_A
  - Pacific_Blue_A
  - SSC_A
- Conditions
  - Condition
  - Dox
  - Replicate
  - Transfected
  - Transfection_Bin
- Statistics
  - ('Transfected', 'mean')
  - ('Transfected', 'sigma')
  - ('Transfected', 'interval')
  - ('Transfected', 'proportion')
  - ('FITC_mean', 'Geom.Mean')

Note: If the **Experiment Browser** isn't visible, you can show it by selecting it from the **View** menu or clicking the corresponding button on the button bar.

Here, you can see the channels, conditions, and statistics in the experiment in a tree view. If you open one, by clicking the caret or double-clicking the name, you are shown additional information about each. Let's walk through all three below.

Channels

Each *channel* is shown in the experiment browser. When you open a channel, additional metadata about the channel is visible, such as the original name in the `.fcs` file and the range of the channel. Additionally, if the channel has been manipulated in any way, or is a "synthetic" or derived channel, the operations that did so leave some metadata as well. For example, if you create a new channel with the **Ratio** operation, the numerator and demoninator are show.



FITC_mean
Channel Statistics

FITC_over_FSC
Ratio

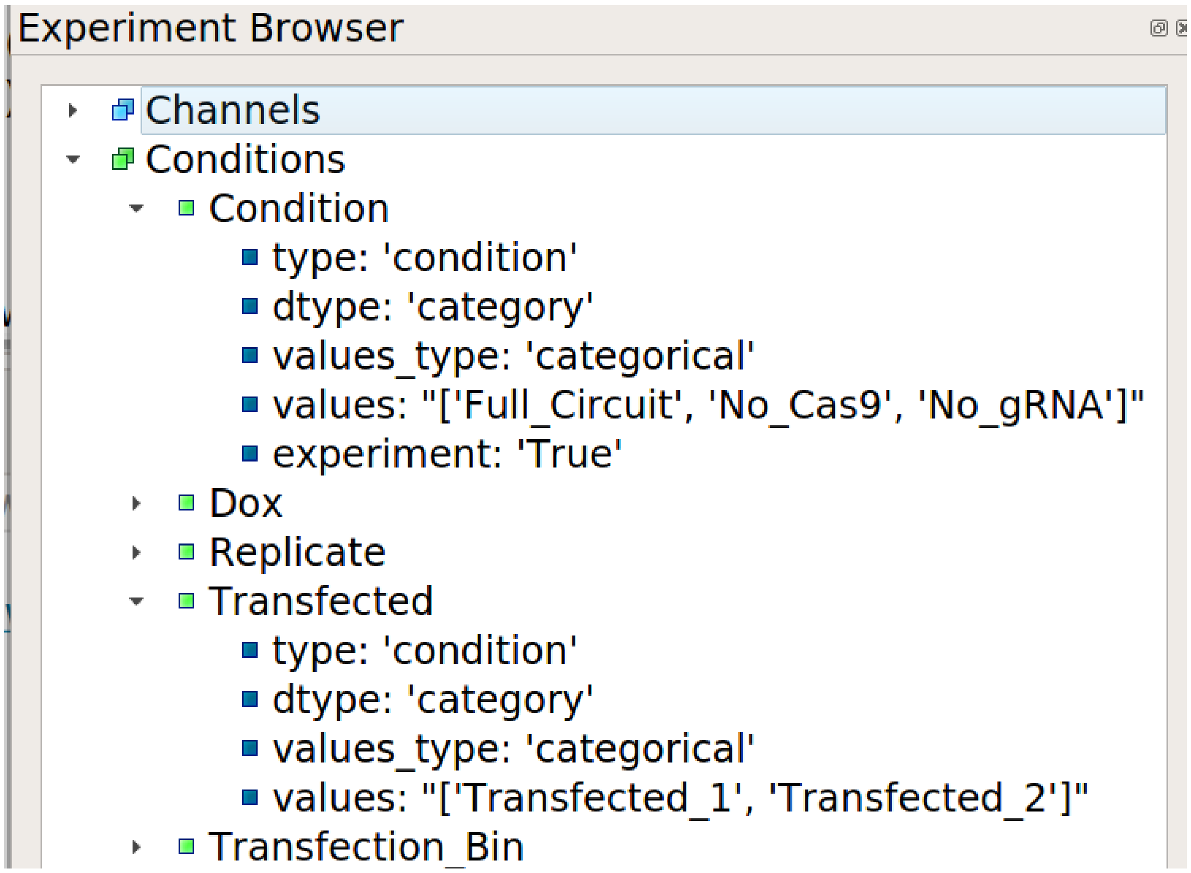
Name: FITC_over_FSC
Numerator: FITC_A
Denominator: FSC_A

Experiment Browser

```
Channels
    FITC_A
        type: 'channel'
        fcs_name: 'FITC-A'
        range: '262143.0'
    FITC_over_FSC
        type: 'channel'
        numerator: 'FITC_A'
        denominator: 'FSC_A'
    FSC_A
    PE_TxRed_YG_A
    Pacific_Blue_A
    SSC_A
```

Conditions

Remember, *conditions* – either experimental conditions or conditions that were added by operations – are how you control the plotting and statistics. You can view the conditions in the browser as well. When you open them, you can see the type of condition, the condition's type (boolean, numeric, categorical), and the values of that condition.

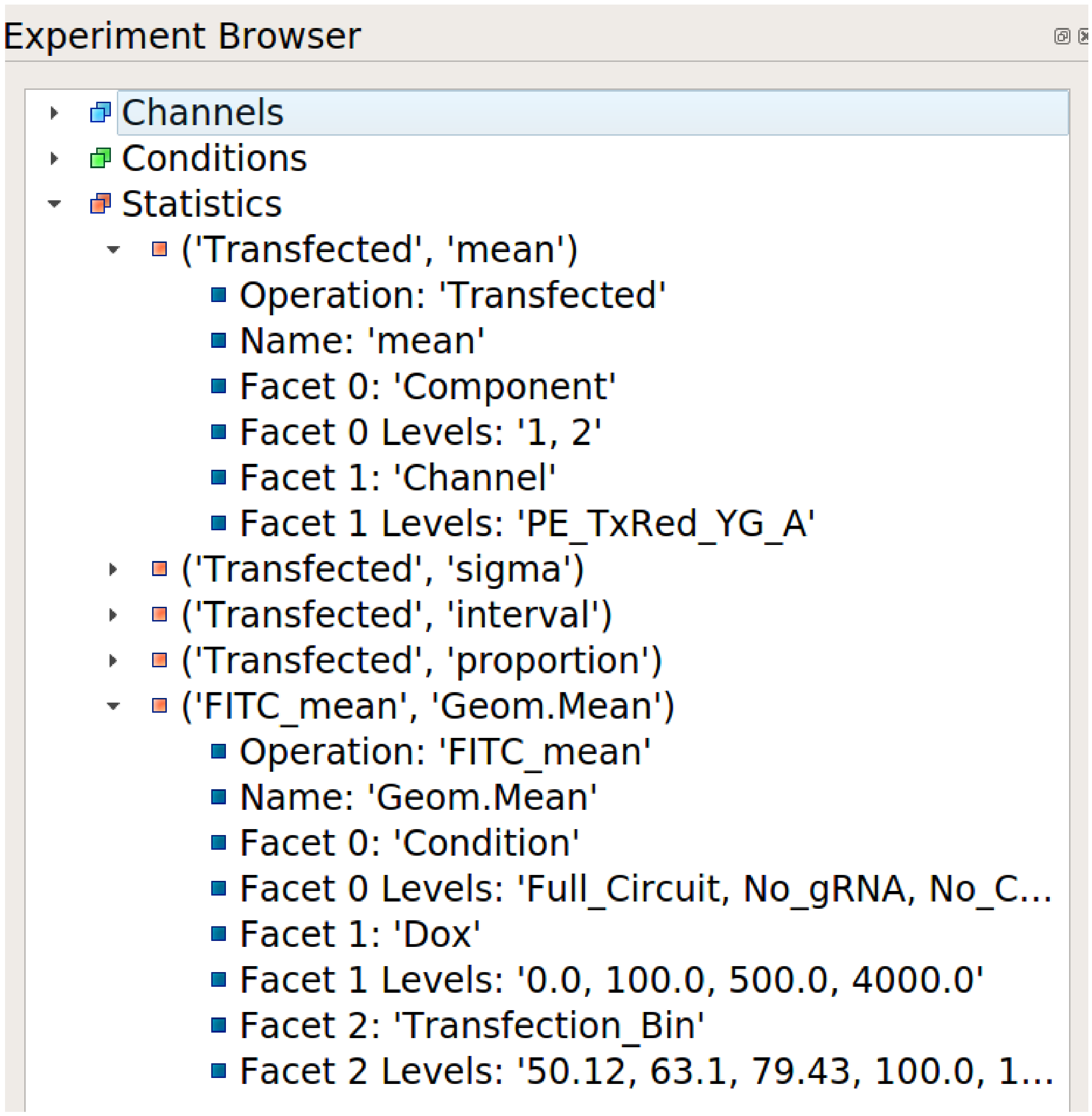


Experiment Browser

- Channels
- Conditions
  - Condition
    - type: 'condition'
    - dtype: 'category'
    - values_type: 'categorical'
    - values: "['Full_Circuit', 'No_Cas9', 'No_gRNA']"
    - experiment: 'True'
  - Dox
  - Replicate
  - Transfected
    - type: 'condition'
    - dtype: 'category'
    - values_type: 'categorical'
    - values: "['Transfected_1', 'Transfected_2']"
  - Transfection_Bin

Statistics

Remember, *statistics* are summaries of the data that are computed by an operation. The statistics that have been added to the experiment are also shown by the browser window, including the facets over which the statistic was computed and the various values of those facets.



Experiment Browser

- Channels
- Conditions
- Statistics
  - ('Transfected', 'mean')
    - Operation: 'Transfected'
    - Name: 'mean'
    - Facet 0: 'Component'
    - Facet 0 Levels: '1, 2'
    - Facet 1: 'Channel'
    - Facet 1 Levels: 'PE_TxRed_YG_A'
  - ('Transfected', 'sigma')
  - ('Transfected', 'interval')
  - ('Transfected', 'proportion')
  - ('FITC_mean', 'Geom.Mean')
    - Operation: 'FITC_mean'
    - Name: 'Geom.Mean'
    - Facet 0: 'Condition'
    - Facet 0 Levels: 'Full_Circuit, No_gRNA, No_C...
    - Facet 1: 'Dox'
    - Facet 1 Levels: '0.0, 100.0, 500.0, 4000.0'
    - Facet 2: 'Transfection_Bin'
    - Facet 2 Levels: '50.12, 63.1, 79.43, 100.0, 1...

4. *Reference documents* for each operation and view in Cytoflow. These are the same documents that appear in the **Help** pane in the user interface.

5.1.19 Operation and View Gallery

Below is a gallery of *Cytoflow*'s operations and views. Each header will take you to the corresponding manual page.

Operations

Autofluorescence correction operations/autofluorescence-1.png	*Bead calibration* operations/bead_calibration-1.png	*Binning* operations/binning-1.png	*Bleedthrough correction* operations/bleedthrough_linear-
Channel Statistics	*Color translation* operations/color_translation-1.png	*Density gate* operations/density-1.png	*FlowPeaks* operations/flowpeaks-1_01.png
1D Gaussian operations/gaussian_1d-1.png	*2D Gaussian* operations/gaussian_2d-1.png	*KMeans Clustering* operations/kmeans-1.png	*Principle Component Analysis*
Polygon Gate operations/polygon-1.png	*Quad Gate* operations/quad-1.png	*Range Gate* operations/range-1.png	*Rectangle Gate* operations/range2d-1.png
Ratio	*TASBE Calibration* operations/tasbe-4.png	*Threshold Gate* operations/threshold-1.png	*Transform Statistic*

Views

Bar Chart views/bar_chart-1.png	*Density Map* views/density-1.png	*Export FCS*	*Histogram* views/histogram-1.png
2D Histogram views/histogram_2d-1.png	*Kernel Density Estimate* views/kde_1d-1.png	*2D Kernel Density Estimate* views/kde_2d-1.png	*Parallel Coordinates Plot* views/parallel_coords-1.png
RadViz Plot views/radviz-1.png	*Scatterplot* views/scatterplot-1.png	*1D Statistics Plot* views/stats_1d-1.png	*2D Statistics Plot* views/stats_2d-1.png
Table View views/table-1.png	*Violin Plot* views/violin-1.png		

Operations

Autofluorescence correction

Apply autofluorescence correction to a set of fluorescence channels.

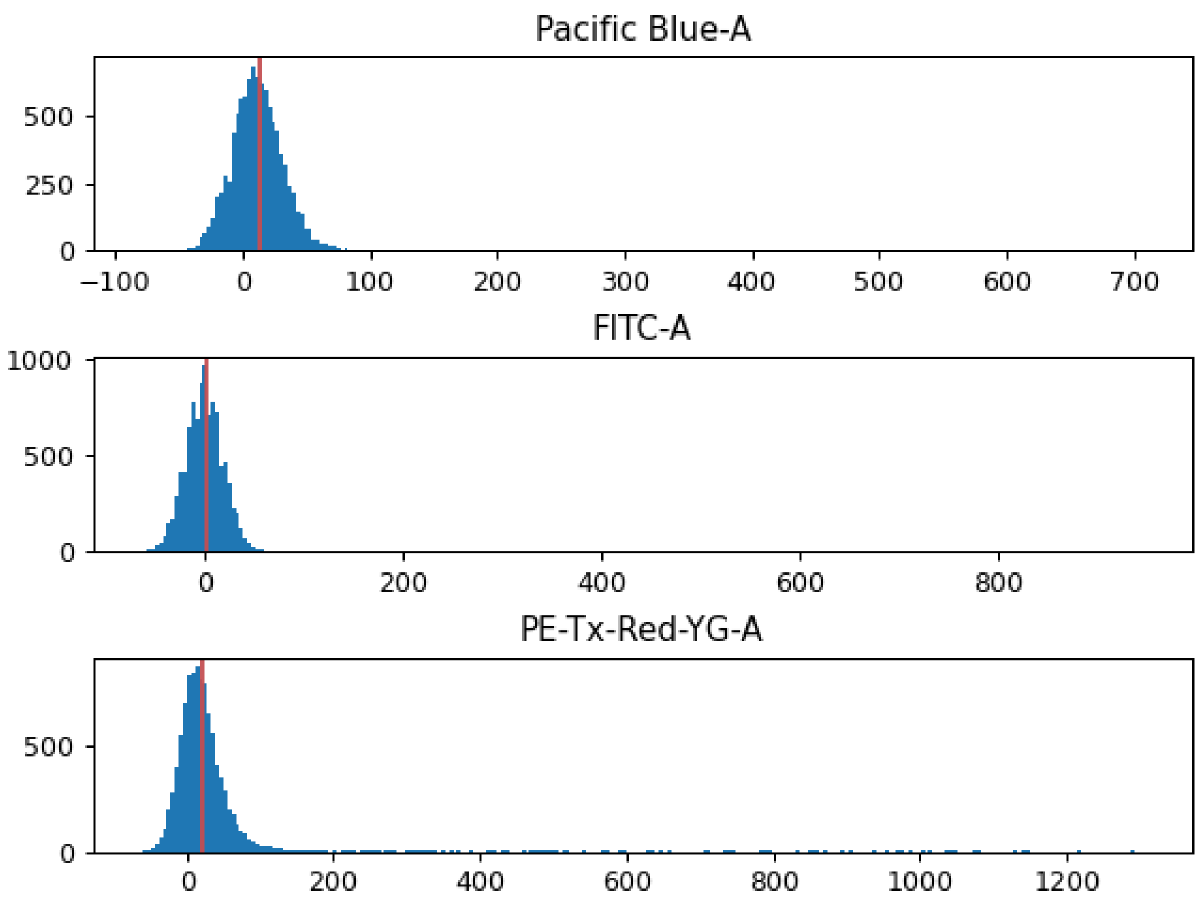
This module estimates the arithmetic median fluorescence from cells that are not fluorescent, then subtracts the median from the experimental data.

Check the diagnostic plot to make sure that the sample is actually non-fluorescent, and that the module found the population median.

Channels
The channels to correct

Blank file
The FCS file containing measurements of blank cells.

Note: You cannot have any operations before this one which estimate model parameters based on experimental conditions. (Eg, you can't use a **Density Gate** to choose morphological parameters and set *by* to an experimental condition.) If you need this functionality, you can access it using the Python module interface.



Bead Calibration

Calibrate arbitrary channels to molecules-of-fluorophore using fluorescent beads (eg, the **Spherotech RCP-30-5A** rainbow beads.)

Computes a log-linear calibration function that maps arbitrary fluorescence units to physical units (ie molecules equivalent fluorophore, or *MEF*).

To use, set **Beads** to the beads you calibrated with (check the lot!) and **Beads File** to an FCS file containing events collected *using the same cytometer settings as the data you're calibrating*. Then, click **Add a channel** to add the channels to calibrate, and set both the channel name and the units you want calibrate to. Click **Estimate**, and *make sure you check the diagnostic plot to see that the correct peaks were found.*

If it didn't find all the peaks (or found too many), try tweaking **Peak Quantile**, **Peak Threshold** and **Peak Cutoff**. If you can't make the peak finding work by tweaking , please submit a bug report!

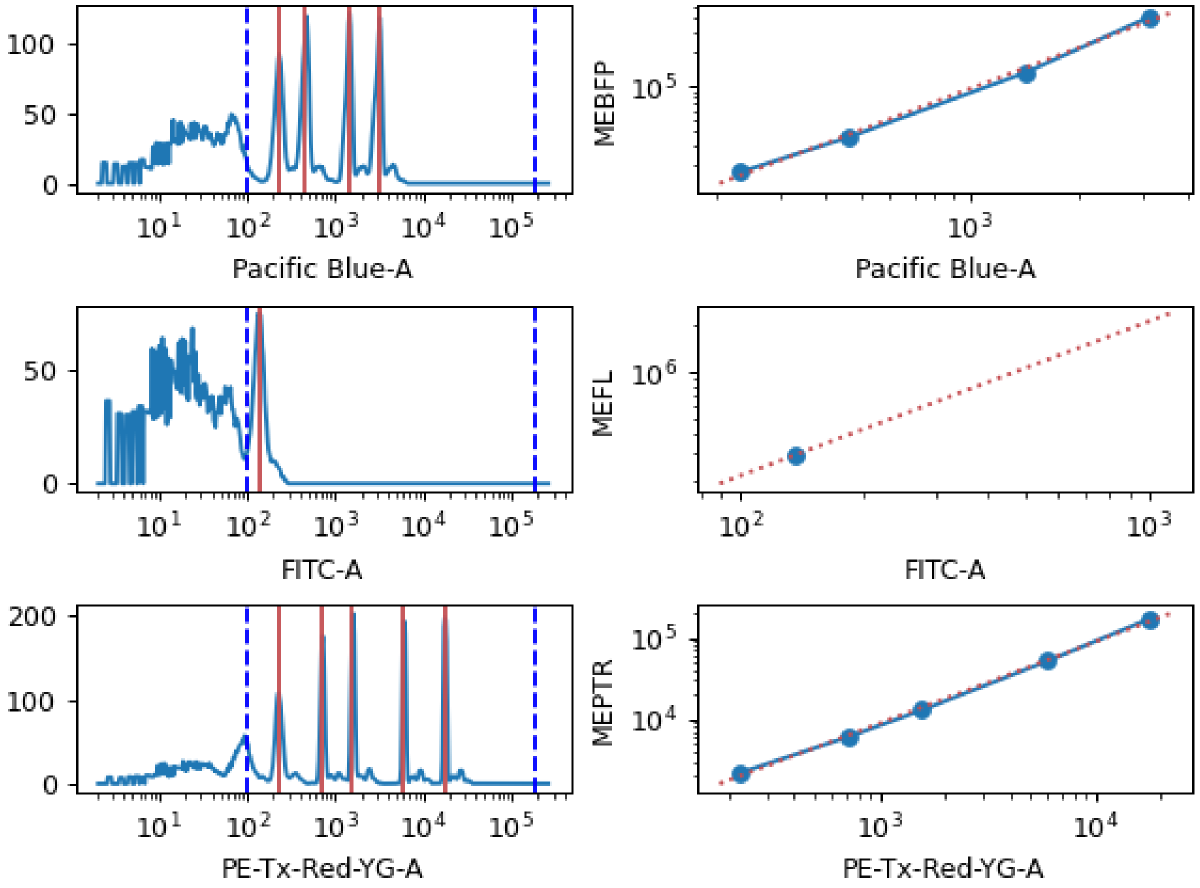
`Beads`
: The beads you're calibrating with. Make sure to check the lot number!

`Beads file`
: A file containing the FCS events from the beads.

`Channels`
A list of the channels you want calibrated and the units you want them calibrated in.

`Peak Quantile`
Peaks must be at least this quantile high to be considered. Default = 80.

`Peak Threshold`
Don't search for peaks below this brightness. Default = 100.



Note: It works best to choose units of a fluorophore that is spectrally matched to the channel that you're calibrating. Here are the beads that `Cytoflow` knows about (including the laser and filter sets used to characterize the beads):

- Spherotech ACP 30-2K
- Spherotech RCP-30-5A Lot AN04, AN03, AN02, AN01, AM02, AM01, AL01, AK04, AK03 & AK02**
- Spherotech RCP-30-5A (Euroflow) Lot EAM02 & EAM01
- Sphreotech RCP-30-5A (Euroflow) Lot EAK01, EAG01, EAE01 & EAF01
- Spherotech RCP-30-5A Lot AK01, AJ01, AH02, AH01, AF02, AF01, AD04 & AE01
- Spherotech RCP-30-5A Lot AG01
- Spherotech RCP-30-5A Lot AA01, AA02, AA03, AA04, AB01, AB02, AC01 & GAA01-R
- Spherotech RCP-30-5A Lot AC02, AC03 & AD01

- Spherotech RCP-30-5A Lot Z02 and Z03
- Spherotech RCP-30-5 Lot AA01, AB01, AB02, AC01 & AD01
- Spherotech RCP-30-5 Lot AM02, AM01, AL01, AH01, AG01, AF01 & AD03
- Spherotech RCP-60-5
- Spherotech URCP 38-2K
- Spherotech URCP 38-2K Lot AN01, AM01, AL02, AL01, AK03, AK02, AK01, AJ02 & AJ03
- Spherotech URCP 50-2K Lot AM01 & AJ01
- Spherotech URCP 50-2K

The Spherotech fluorophores labels and the laser / filter sets (that I know about) are:

- **MECSB** (Cascade Blue, 405 –> 450/50)
- **MEBFP** (BFP, 405 –> 530/40)
- **MEFL** (Fluroscein, 488 –> 530/40)
- **MEPE** (Phycoerythrin, 488 –> 575/25)
- **MEPTR** (PE-Texas Red, 488 –> 613/20)
- **MECY** (Cy5, 488 –> 680/30)
- **MEPCY7** (PE-Cy7, 488 –> 750 LP)
- **MEAP** (APC, 633 –> 665/20)
- **MEAPCY7** (APC-Cy7, 635 –> 750 LP)

Binning

Bin data along an axis.

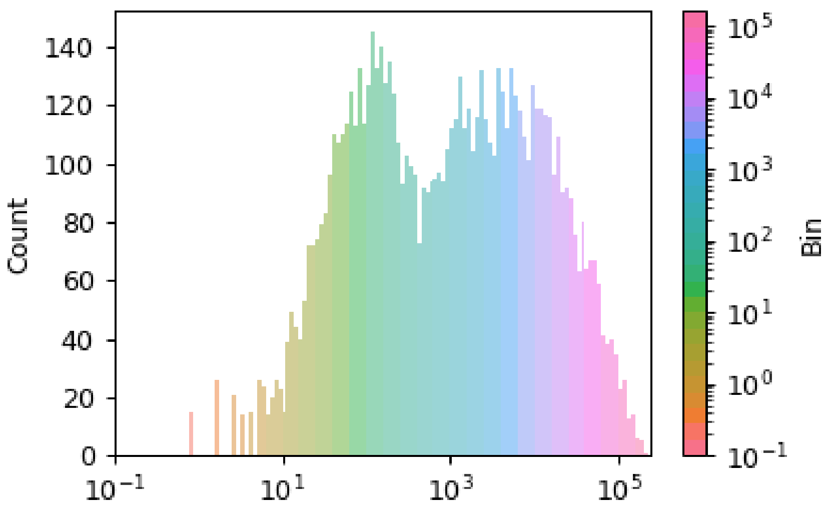
This operation creates equally spaced bins (in linear or log space) along an axis and adds a condition assigning each event to a bin. The value of the event's condition is the left end of the bin's interval in which the event is located.

`Name`
: The name of the new condition created by this operation.

`Channel`
: The channel to apply the binning to.

`Scale`
: The scale to apply to the channel before binning.

`Bin Width`
: How wide should each bin be? Can only set if **Scale** is *linear* or *log* (in which case, **Bin Width** is in log10-units.)



Linear Bleedthrough Compensation

Apply matrix-based bleedthrough correction to a set of fluorescence channels.

This is a traditional matrix-based compensation for bleedthrough. For each pair of channels, the module estimates the proportion of the first channel that bleeds through into the second, then performs a matrix multiplication to compensate the raw data.

This works best on data that has had autofluorescence removed first; if that is the case, then the autofluorescence will be subtracted from the single-color controls too.

To use, specify the single-color control files and which channels they should be measured in, then click **Estimate**. Check the diagnostic plot to make sure the estimation looks good. There must be at least two channels corrected.

`Add Control, Remove Control`
Add or remove single-color controls.

Note: You cannot have any operations before this one which estimate model parameters based on experimental conditions. (Eg, you can't use a **Density Gate** to choose morphological parameters and set *by* to an experimental condition.) If you need this functionality, you can access it using the Python module interface.

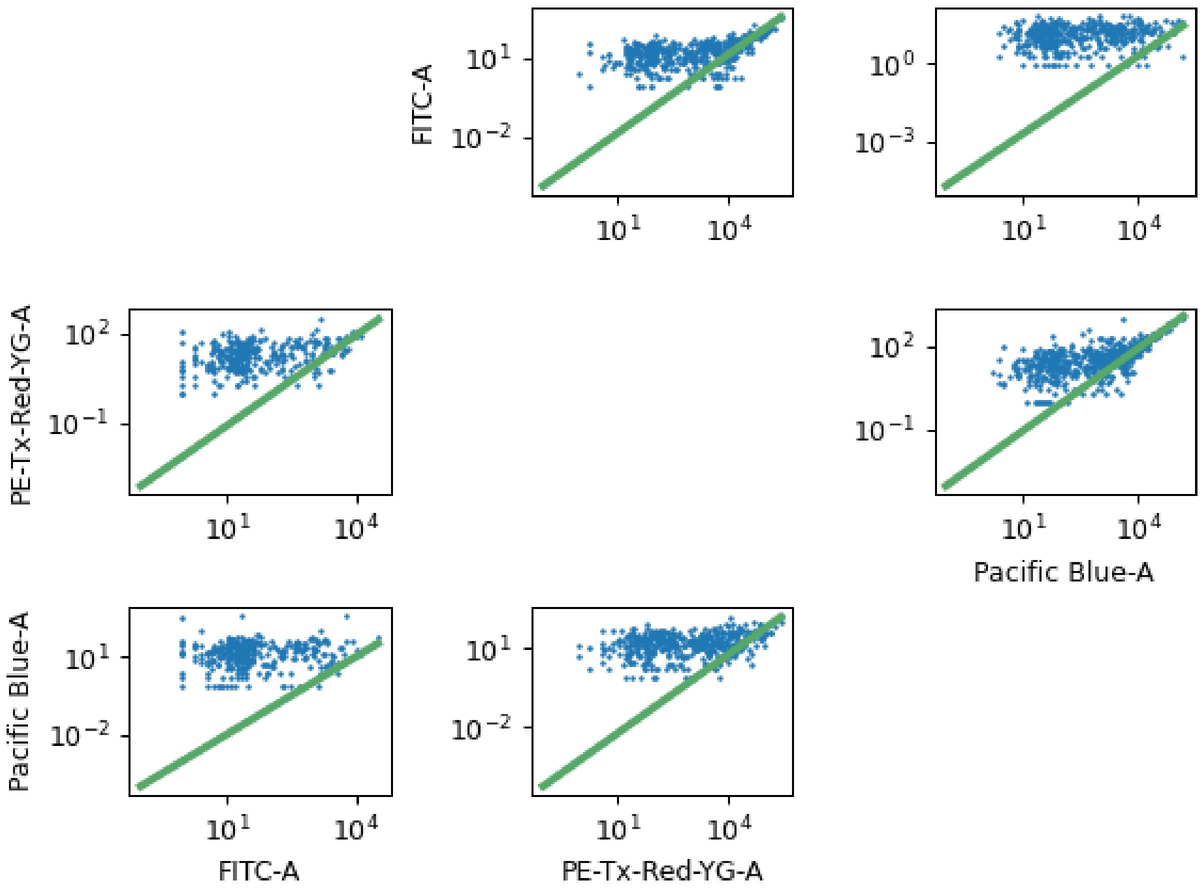
Channel statistic

Apply a function to subsets of a data set, and add it as a statistic to the experiment.

First, the module groups the data by the unique values of the variables in **By**, then applies **Function** to the **Channel** in each group.

`Name`
The operation name. Becomes the first part of the new statistic's name.

`Channel`
The channel to apply the function to.


FITC-A
PE-Tx-Red-YG-A
Pacific Blue-A
FITC-A
PE-Tx-Red-YG-A
Pacific Blue-A
10^1
10^4
10^{-2}
10^0
10^{-3}
10^2
10^{-1}

`Function`
The function to compute on each group.

`Subset`
Only apply the function to a subset of the data. Useful if the function is very slow.

Color Translation

Translate measurements from one color's scale to another, using a two-color or three-color control.

To use, set up the **Controls** list with the channels to convert and the FCS files to compute the mapping. Click **Estimate** and make sure to check that the diagnostic plots look good.

`Add Control, Remove Control`
Add and remove controls to compute the channel mappings.

`Use mixture model?`
If `True`, try to model the **from** channel as a mixture of expressing cells and non-expressing cells (as you would get with a transient transfection), then weight the regression by the probability that the the cell is from the top (transfected) distribution. Make sure you check the diagnostic plots to see that this worked!

Note: You cannot have any operations before this one which estimate model parameters based on experimental conditions. (Eg, you can't use a **Density Gate** to choose morphological parameters and set *by* to an experimental condition.) If you need this functionality, you can access it using the Python module interface.

Density Gate

Computes a gate based on a 2D density plot. The user chooses what proportion of events to keep, and the module creates a gate that selects those events in the highest-density bins of a 2D density histogram.

A single gate may not be appropriate for an entire experiment. If this is the case, you can use **By** to specify metadata by which to aggregate the data before computing and applying the gate.

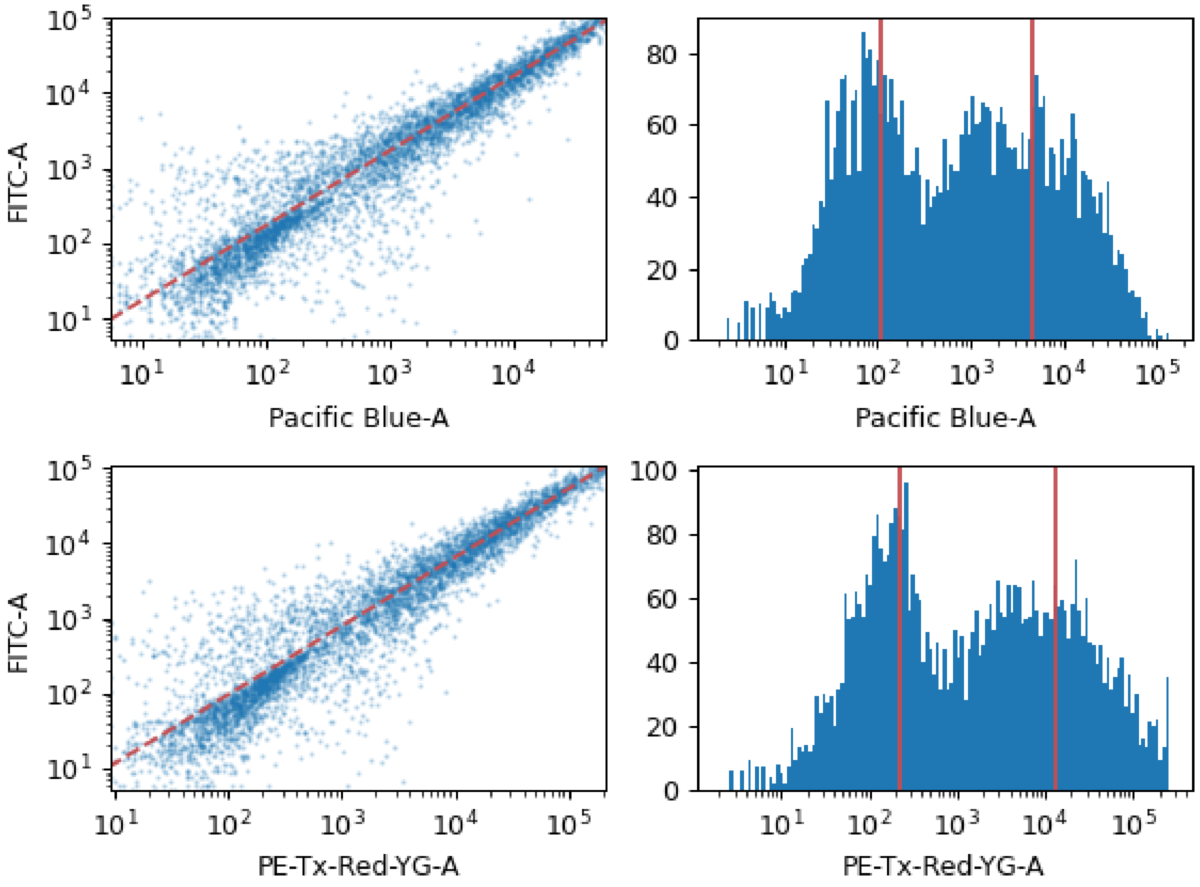
`Name`
The operation name; determines the name of the new metadata

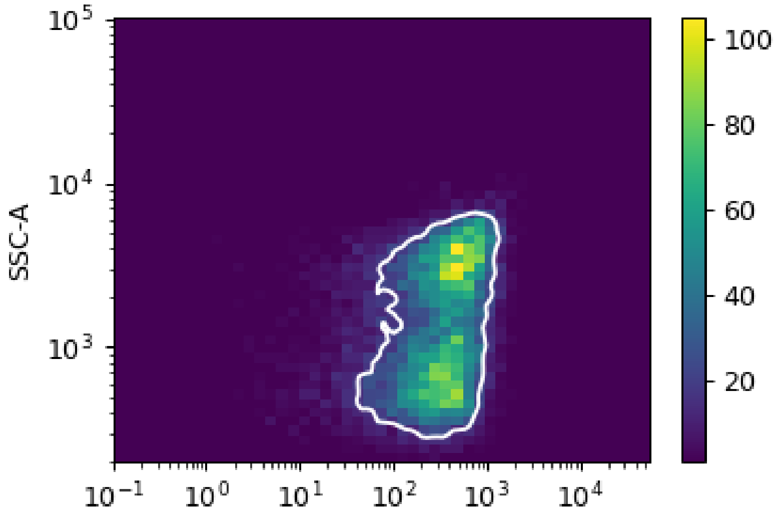
`X Channel, Y Channel`
The channels to apply the mixture model to.

`X Scale, Y Scale`
Re-scale the data in **Channel** before fitting.

`Keep`
The proportion of events to keep in the gate. Defaults to 0.9.

`By`
A list of metadata attributes to aggregate the data before estimating the model. For example, if the experiment has two pieces of metadata, `Time` and `Dox`, setting **by** to `["Time", "Dox"]` will fit the model separately to each subset of the data with a unique combination of `Time` and `Dox`.


FITC-A
Pacific Blue-A
FITC-A
PE-Tx-Red-YG-A
Pacific Blue-A
PE-Tx-Red-YG-A


SSC-A

FlowPeaks Clustering

This module uses the **flowPeaks** algorithm to assign events to clusters in an unsupervized manner.

`Name`
: The operation name; determines the name of the new metadata

`X Channel, Y Channel`
: The channels to apply the mixture model to.

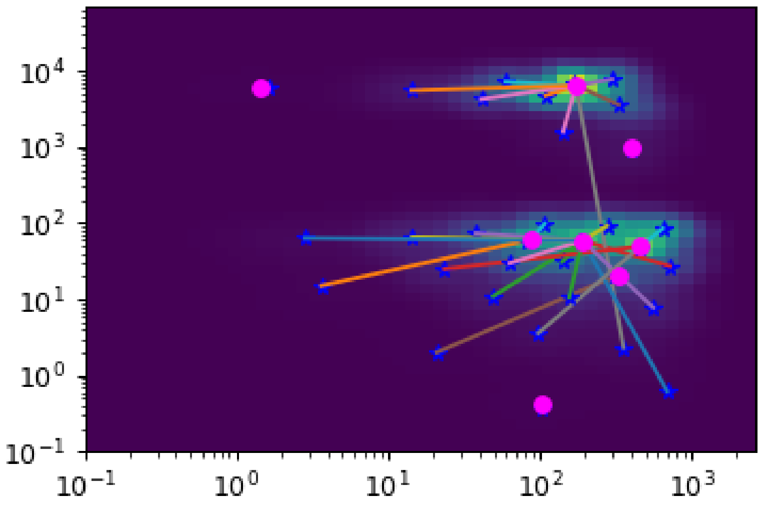
`X Scale, Y Scale`
: Re-scale the data in **Channel** before fitting.

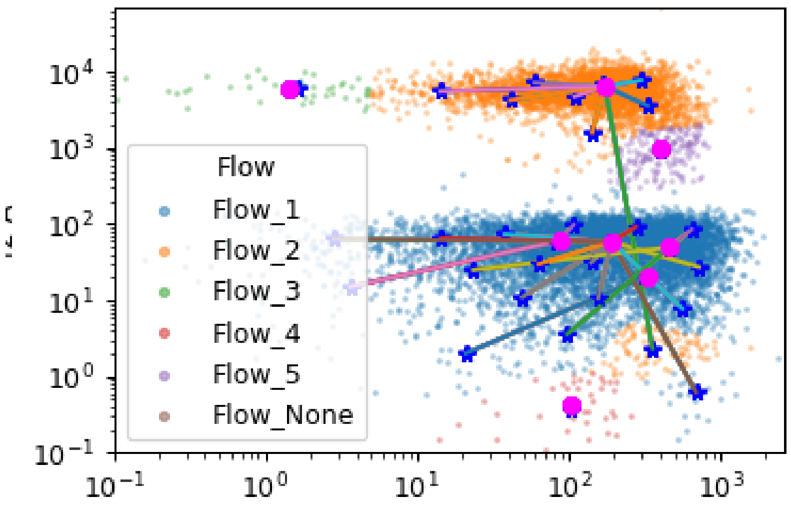
`h, h0`
: Scalar values that control the smoothness of the estimated distribution. Increasing **h** makes it "rougher," while increasing **h0** makes it smoother.

`tol`
: How readily should clusters be merged? Must be between 0 and 1.

`Merge Distance`
: How far apart can clusters be before they are merged?

`By`
: A list of metadata attributes to aggregate the data before estimating the model. For example, if the experiment has two pieces of metadata, `Time` and `Dox`, setting **By** to `["Time", "Dox"]` will fit the model separately to each subset of the data with a unique combination of `Time` and `Dox`.





Gaussian Mixture Model (1D)

Fit a Gaussian mixture model with a specified number of components to one channel.

If **Num Components** is greater than 1, then this module creates a new categorical metadata variable named **Name**, with possible values `{name}_1` .... `name_n` where `n` is the number of components. An event is assigned to `name_i` category if it has the highest posterior probability of having been produced by component `i`. If an event has a value that is outside the range of one of the channels' scales, then it is assigned to `{name}_None`.

Additionally, if **Sigma** is greater than 0, this module creates new boolean metadata variables named `{name}_1` ... `{name}_n` where `n` is the number of components. The column `{name}_i` is `True` if the event is less than **Sigma** standard deviations from the mean of component `i`. If **Num Components** is `1`, **Sigma** must be greater than 0.

Finally, the same mixture model (mean and standard deviation) may not be appropriate for every subset of the data. If this is the case, you can use **By** to specify metadata by which to aggregate the data before estimating and applying a mixture model.

Note: **Num Components** and **Sigma** withh be the same for each subset.

`Name`
The operation name; determines the name of the new metadata

`Channel`
The channels to apply the mixture model to.

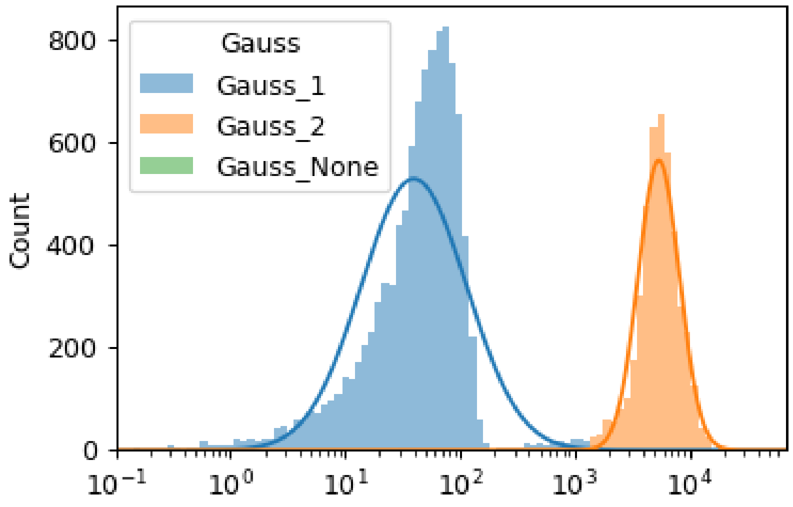
`Scale`
Re-scale the data in **Channel** before fitting.

`Num Components`
How many components to fit to the data? Must be a positive integer.

`Sigma`
How many standard deviations on either side of the mean to include in the boolean variable `{name}_i`? Must be `None` or > `0.0`. If **Num Components** is 1, must be > `0`.

By

A list of metadata attributes to aggregate the data before estimating the model. For example, if the experiment has two pieces of metadata, `Time` and `Dox`, setting **By** to `["Time", "Dox"]` will fit the model separately to each subset of the data with a unique combination of `Time` and `Dox`.



Gaussian Mixture Model (2D)

Fit a Gaussian mixture model with a specified number of components to two channels.

If **Num Components** is greater than 1, then this module creates a new categorical metadata variable named **Name**, with possible values `{name}_1` .... `name_n` where `n` is the number of components. An event is assigned to `name_i` category if it has the highest posterior probability of having been produced by component `i`. If an event has a value that is outside the range of one of the channels' scales, then it is assigned to `{name}_None`.

Additionally, if **Sigma** is greater than 0, this module creates new boolean metadata variables named `{name}_1` ... `{name}_n` where `n` is the number of components. The column `{name}_i` is `True` if the event is less than **Sigma** standard deviations from the mean of component `i`. If **Num Components** is 1, **Sigma** must be greater than 0.

Finally, the same mixture model (mean and standard deviation) may not be appropriate for every subset of the data. If this is the case, you can use **By** to specify metadata by which to aggregate the data before estimating and applying a mixture model.

Note: **Num Components** and **Sigma** will be the same for each subset.

Name

The operation name; determines the name of the new metadata

X Channel, Y Channel

The channels to apply the mixture model to.

X Scale, Y Scale

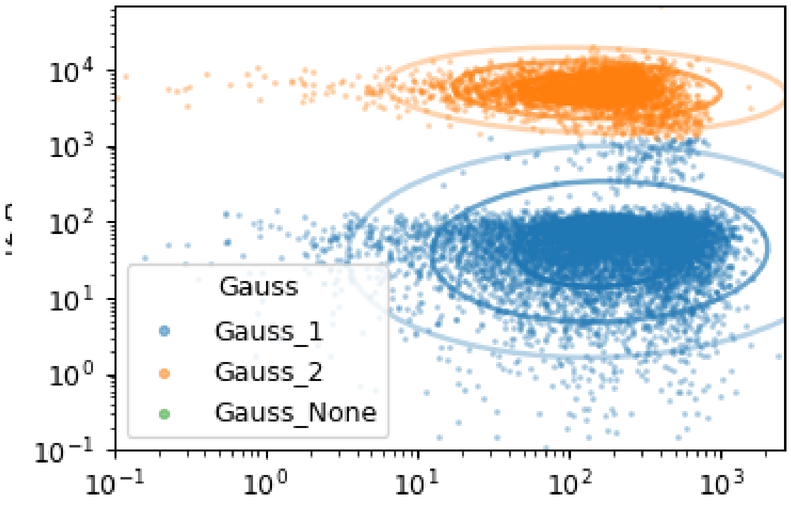
Re-scale the data in **Channel** before fitting.

Num Components

How many components to fit to the data? Must be a positive integer.

`Sigma`
How many standard deviations on either side of the mean to include in the boolean variable `{name}_i`? Must be >= `0.0`. If **Num Components** is 1, must be > `0`.

`By`
A list of metadata attributes to aggregate the data before estimating the model. For example, if the experiment has two pieces of metadata, `Time` and `Dox`, setting **By** to `["Time", "Dox"]` will fit the model separately to each subset of the data with a unique combination of `Time` and `Dox`.



Import Files

Import FCS files and associate them with experimental conditions (metadata.)

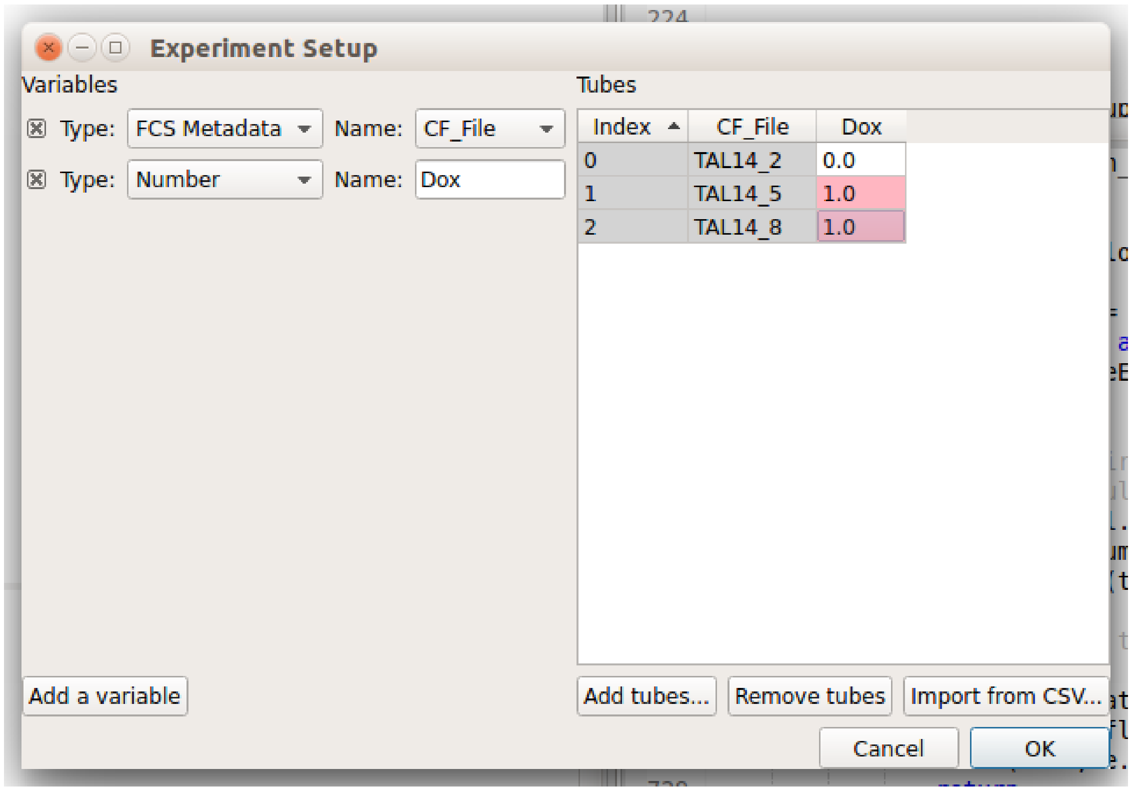
`Channels`
Here, you can rename channels to use names that are more informative, or remove channels you don't need. Names must be valid Python identifiers (must contain only letters, numbers and underscores and must start with a letter or underscore.)

`Reset channel names`
Reset the channels and channel names.

`Events per sample`
For very large data sets, *Cytoflow*'s interactive operation may be too slow. By setting **Events per sample**, you can tell *Cytoflow* to import a smaller number of events from each FCS file, which will make interactive data exploration much faster. When you're done setting up your workflow, set **Events per sample** to empty or 0 and *Cytoflow* will re-run your workflow with the entire data set.

`Set up experiment....`
Open the sample editor dialog box.

The sample editor dialog



Type: FCS Metadata Name: CF_File

Type: Number Name: Dox

Index	CF_File	Dox
0	TAL14_2	0.0
1	TAL14_5	1.0
2	TAL14_8	1.0

Allows you to specify FCS files in the experiment, and the experimental conditions that each tube (or well) was subject to.

Note: You can select sort the table by clicking on a row header.

Note: You can select multiple entries in a column by clicking one, holding down *Shift*, and clicking another (to select a range); or, by holding down *Ctrl* and clicking multiple additional cells in the table. If multiple cells are selected, typing a value will update all of them.

Note: **Each tube must have a unique set of experimental conditions.** If a tube's conditions are not unique, the row is red and you will not be able to click "OK".

Add tubes
Opens a file selector to add tubes.

KMeans

This module uses the **KMeans** algorithm to assign events to clusters in an unsupervized manner.

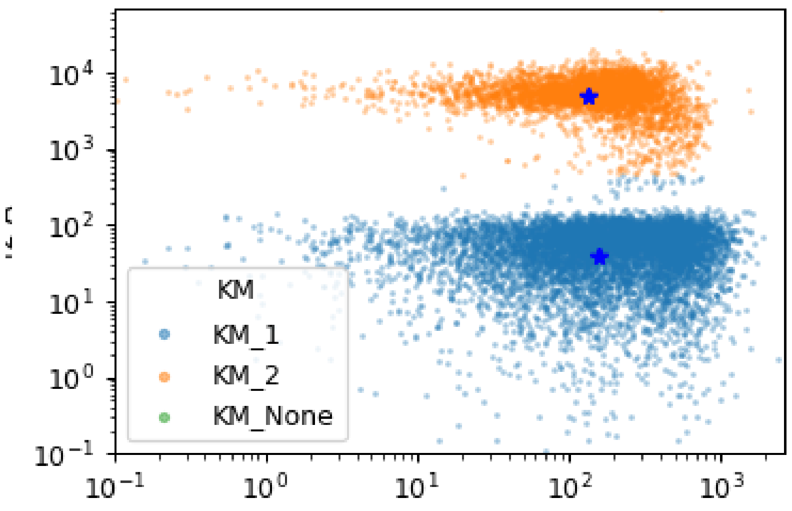
`Name`
: The operation name; determines the name of the new metadata

`X Channel, Y Channel`
: The channels to apply the mixture model to.

`X Scale, Y Scale`
: Re-scale the data in **Channel** before fitting.

`Num Clusters`
: How many clusters to assign the data to.

`By`
: A list of metadata attributes to aggregate the data before estimating the model. For example, if the experiment has two pieces of metadata, `Time` and `Dox`, setting **By** to `["Time", "Dox"]` will fit the model separately to each subset of the data with a unique combination of `Time` and `Dox`.



Principal Component Analysis

Use principal components analysis (PCA) to decompose a multivariate data set into orthogonal components that explain a maximum amount of variance.

Creates new "channels" named `{name}_1 ... {name}_n`, where `name` is the **Name** attribute and `n` is **Num components**.

The same decomposition may not be appropriate for different subsets of the data set. If this is the case, you can use the **By** attribute to specify metadata by which to aggregate the data before estimating (and applying) a model. The PCA parameters such as the number of components and the kernel are the same across each subset, though.

`Name`
: The operation name; determines the name of the new columns.

`Channels`
: The channels to apply the decomposition to.

`Scale`
Re-scale the data in the specified channels before fitting.

`Num components`
How many components to fit to the data? Must be a positive integer.

`By`
A list of metadata attributes to aggregate the data before estimating the model. For example, if the experiment has two pieces of metadata, `Time` and `Dox`, setting **By** to `["Time", "Dox"]` will fit the model separately to each subset of the data with a unique combination of `Time` and `Dox`.

`Whiten`
Scale each component to unit variance? May be useful if you will be using unsupervized clustering (such as K-means).

Polygon Gate

Draw a polygon gate. To add vertices, use a single-click; to close the polygon, click the first vertex a second time.

`Name`
The operation name. Used to name the new metadata field that's created by this module.

`X Channel`
The name of the channel on the gate's X axis.

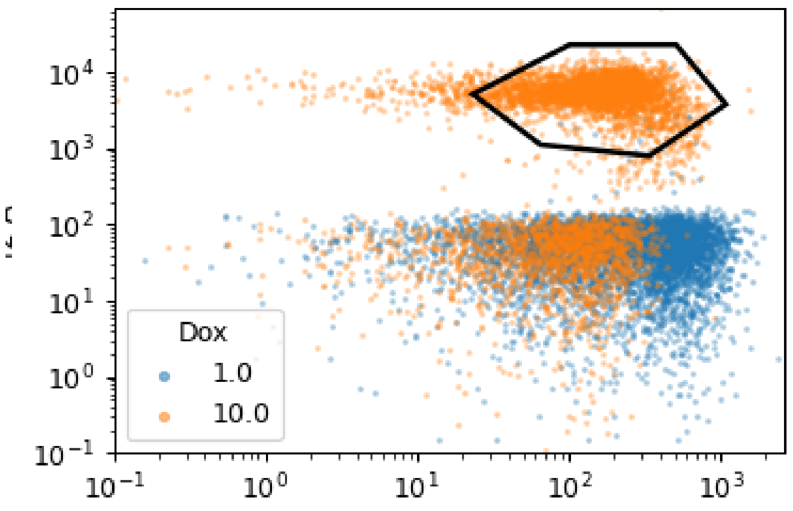
`Y Channel`
The name of the channel on the gate's Y axis.

`X Scale`
The scale of the X axis for the interactive plot.

`Y Scale`
The scale of the Y axis for the interactive plot

`Hue facet`
Show different experimental conditions in different colors.

`Subset`
Show only a subset of the data.



Quadrant Gate

Draw a "quadrant" gate. To create a new gate, just click where you'd like the intersection to be. Creates a new metadata column named *name*, with values `name_1` (upper-left quadrant), `name_2` (upper-right), `name_3` (lower-left), and `name_4` (lower-right).

Note: This matches the order of FACSDiva quad gates.

Name
The operation name. Used to name the new metadata field that's created by this operation.

X channel
The name of the channel on the X axis.

X threshold
The threshold in the X channel.

Y channel
The name of the channel on the Y axis.

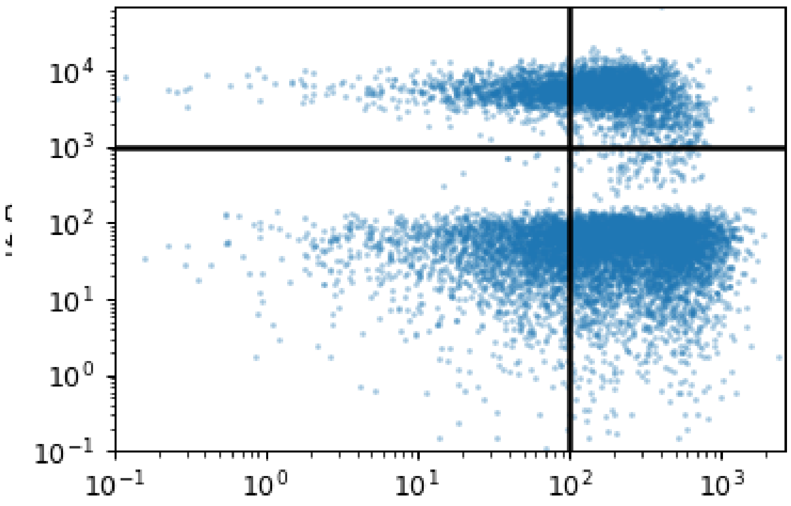
Y threshold
The threshold in the Y channel.

X Scale
The scale of the X axis for the interactive plot.

Y Scale
The scale of the Y axis for the interactive plot

Hue facet
Show different experimental conditions in different colors.

Subset
Show only a subset of the data.



Range Gate

Draw a range gate. To draw a new range, click-and-drag across the plot.

Name
The operation name. Used to name the new metadata field that's created by this module.

Channel
The name of the channel to apply the gate to.

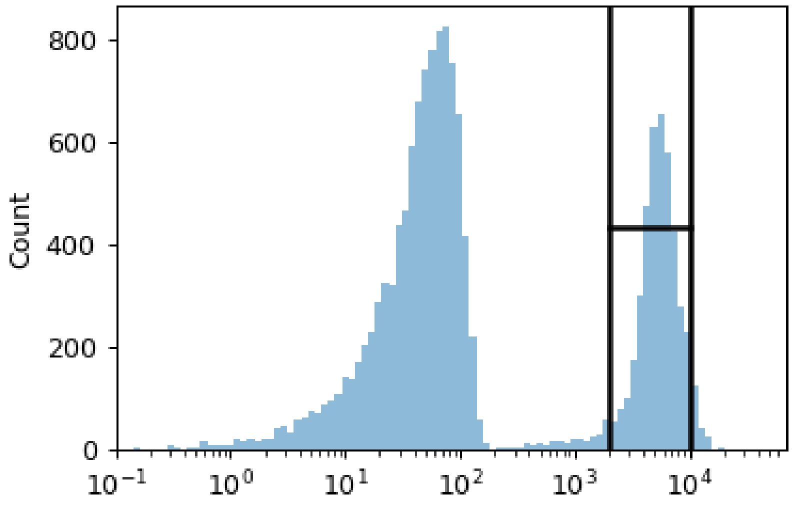
Low
The low threshold of the gate.

High
The high threshold of the gate.

Scale
The scale of the axis for the interactive plot

Hue facet
Show different experimental conditions in different colors.

Subset
Show only a subset of the data.



2D Range Gate

Draw a 2-dimensional range gate (eg, a rectangle). To set the gate, click-and-drag on the plot.

Name
The operation name. Used to name the new metadata field that's created by this operation.

X channel
The name of the channel on the X axis.

X Low
The low threshold in the X channel.

`X High`
: The high threshold in the X channel.

`Y channel`
: The name of the channel on the Y axis.

`Y Low`
: The low threshold in the Y channel.

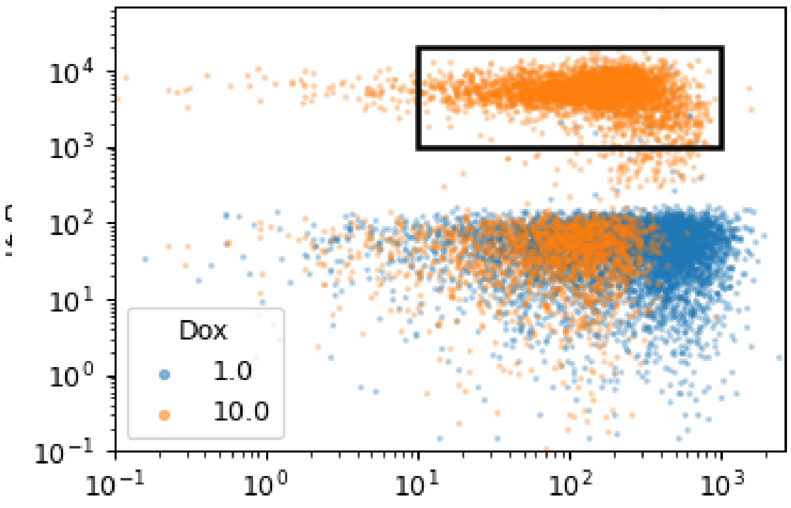
`Y High`
: The high threshold in the Y channel.

`X Scale`
: The scale of the X axis for the interactive plot.

`Y Scale`
: The scale of the Y axis for the interactive plot

`Hue facet`
: Show different experimental conditions in different colors.

`Subset`
: Show only a subset of the data.



Ratio

Adds a new "channel" to the workflow, where the value of the channel is the ratio of two other channels.

`Name`
: The name of the new channel.

`Numerator`
: The numerator for the ratio.

`Denominator`
: The denominator for the ratio.

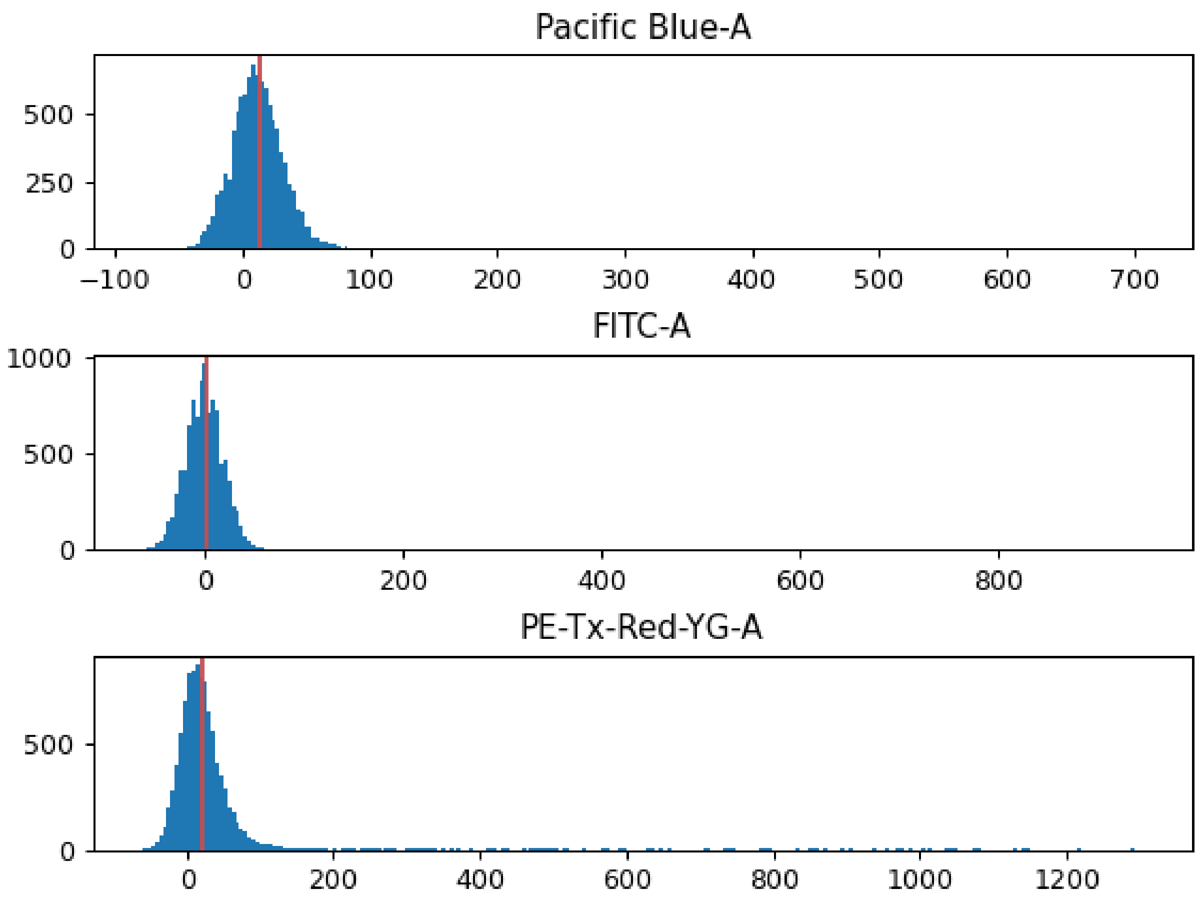
TASBE Calibrated Flow Cytometry

This module combines all of the other calibrated flow cytometry modules (autofluorescence, bleedthrough compensation, bead calibration, and channel translation) into one easy-use-interface.

`Channels`
Which channels are you calibrating?

`Autofluorescence`

`Blank File`
The FCS file with the blank (unstained or untransformed) cells, for autofluorescence correction.



`Bleedthrough Correction`
A list of single-color controls to use in bleedthrough compensation. There's one entry per channel to compensate.

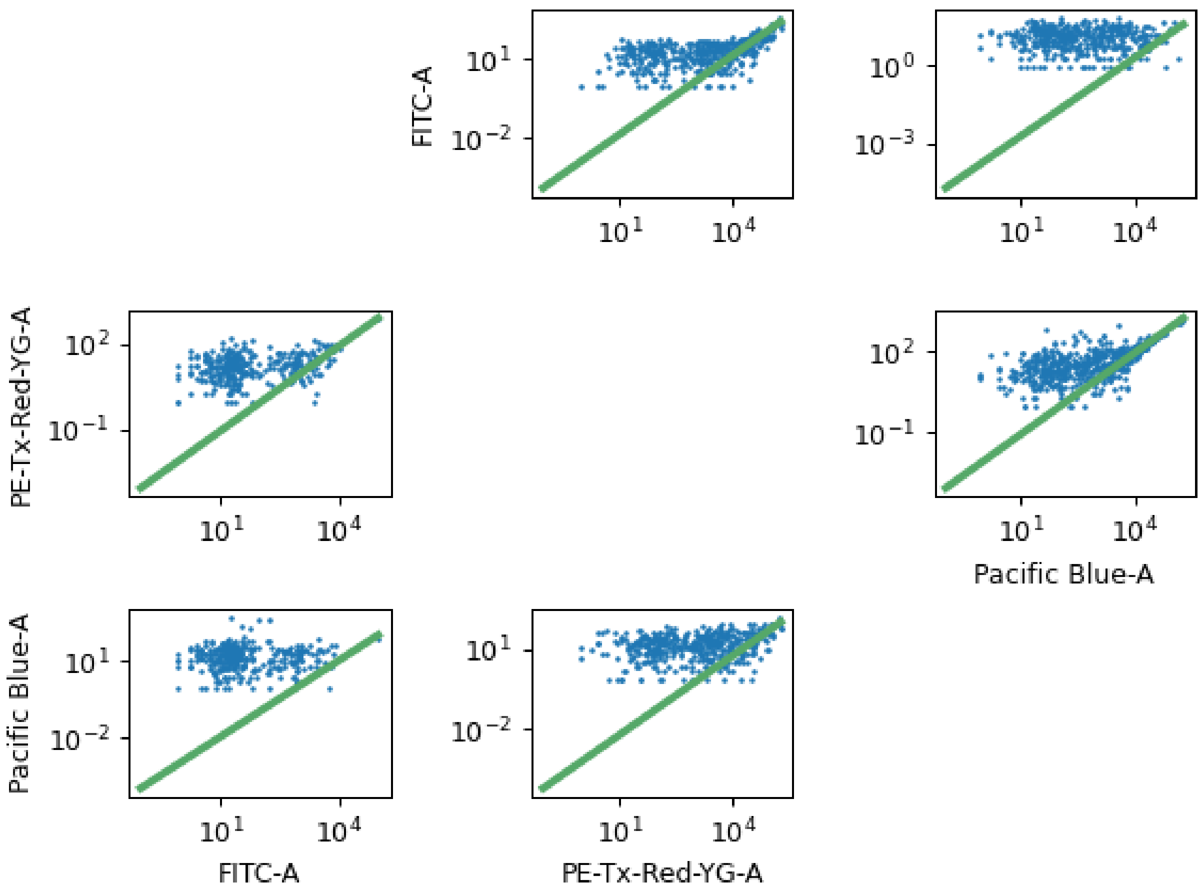
`Channel`

The channel that this file is the single-color control for.

`File`

The FCS file containing the single-color control data.

`Bead Calibration`
The beads that you used for calibration. Make sure to check the lot number as well!


FITC-A
PE-Tx-Red-YG-A
Pacific Blue-A
FITC-A
PE-Tx-Red-YG-A
Pacific Blue-A
10^1
10^{-2}
10^0
10^{-3}
10^2
10^{-1}
10^1
10^4

The FCS file containing the bead data.

The unit (such as *MEFL*) to calibrate to.

`Peak Quantile`

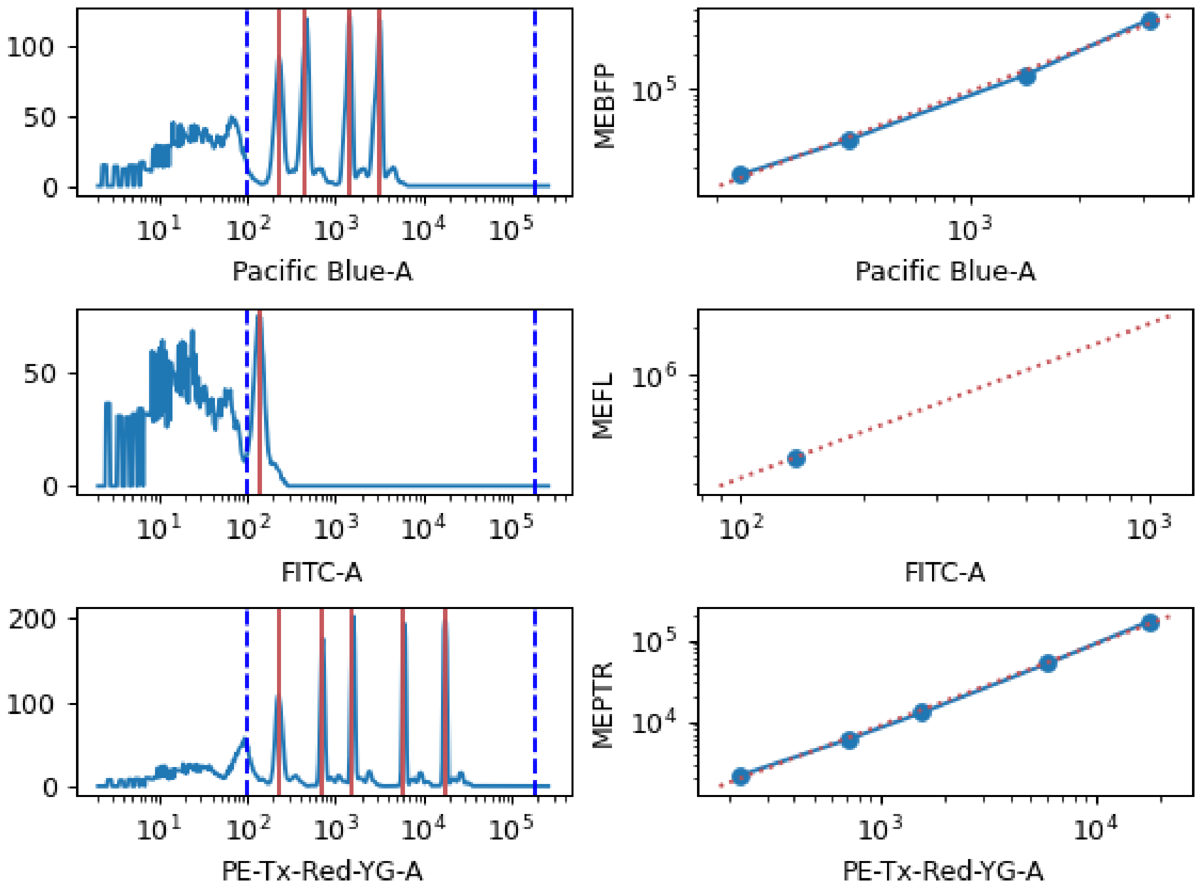
The minimum quantile required to call a peak in the bead data. Check the diagnostic plot: if you have peaks that aren't getting called, decrease this. If you have "noise" peaks that are getting called incorrectly, increase this.

`Peak Threshold`

The minumum brightness where the module will call a peak.

`Peak Cutoff`

The maximum brightness where the module will call a peak. Use this to remove peaks that are saturating the detector.



`Color Translation`

`To Channel`

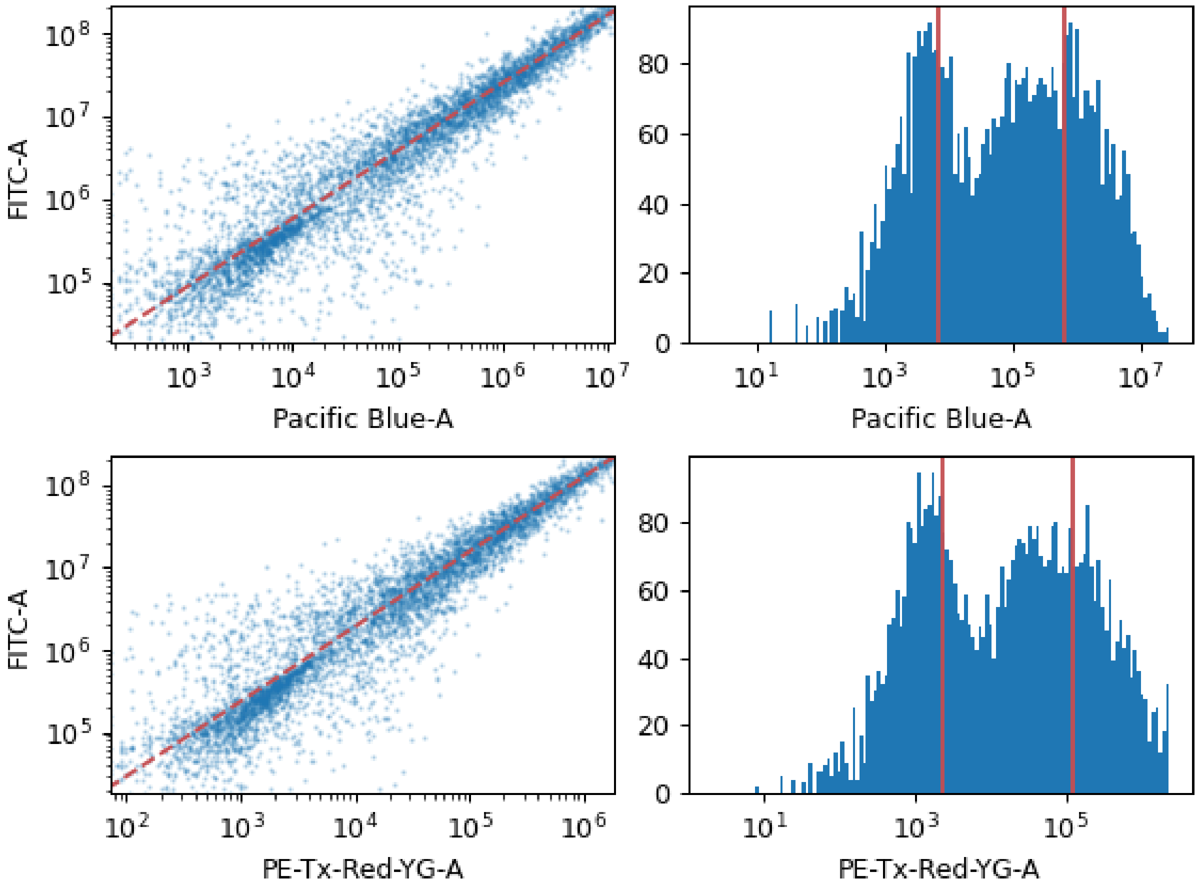
Which channel should we rescale all the other channels to?

`Use mixture model?`

If this is set, the module will try to separate the data using a mixture-of-Gaussians, then only compute the translation using the higher population. This is the kind of behavior that you see in a transient transfection in mammalian cells, for example.

`Translation list`

Each pair of channels must have a multi-color control from which to compute the scaling factor.



Threshold Gate

Draw a threshold gate. To set a new threshold, click on the plot.

`Name`
The operation name. Used to name the new metadata field that's created by this module.

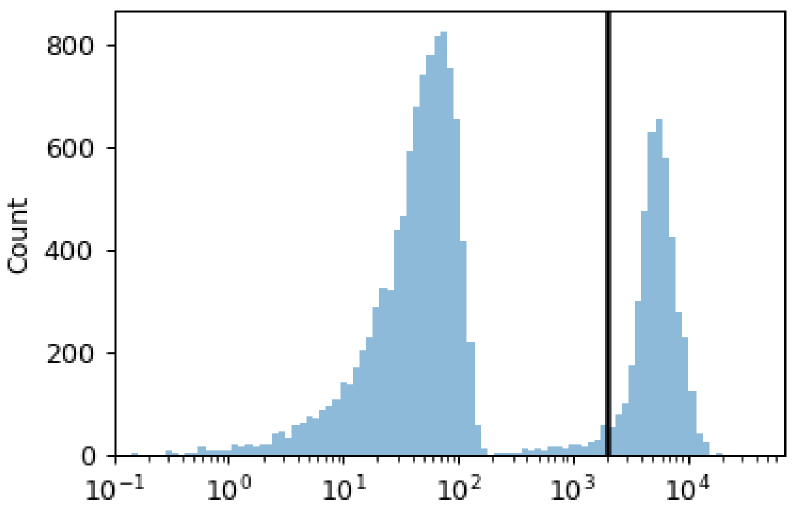
`Channel`
The name of the channel to apply the gate to.

`Threshold`
The threshold of the gate.

`Scale`
The scale of the axis for the interactive plot

`Hue facet`
Show different experimental conditions in different colors.

`Subset`
Show only a subset of the data.



Transform statistic

Apply a function to a statistic, and add it as a statistic to the experiment.

First, the module groups the data by the unique values of the variables in **By**, then applies **Function** to the statistic in each group.

Note: Statistics are a central part of *Cytoflow*. More documentation is forthcoming.

`Name`
The operation name. Becomes the first part of the new statistic's name.

`Statistic`
The statistic to apply the function to.

`Function`
The function to compute on each group.

`Subset`
Only apply the function to a subset of the input statistic. Useful if the function is very slow.

Views

Bar Chart

Plots a bar chart of a statistic.

Each variable in the statistic (ie, each variable chosen in the statistic operation's **Group By**) must be set as **Variable** or as a facet.

`Statistic`
Which statistic to plot.

`Variable`
The statistic variable to use as the major bar groups.

`Scale`
: How to scale the statistic plot.

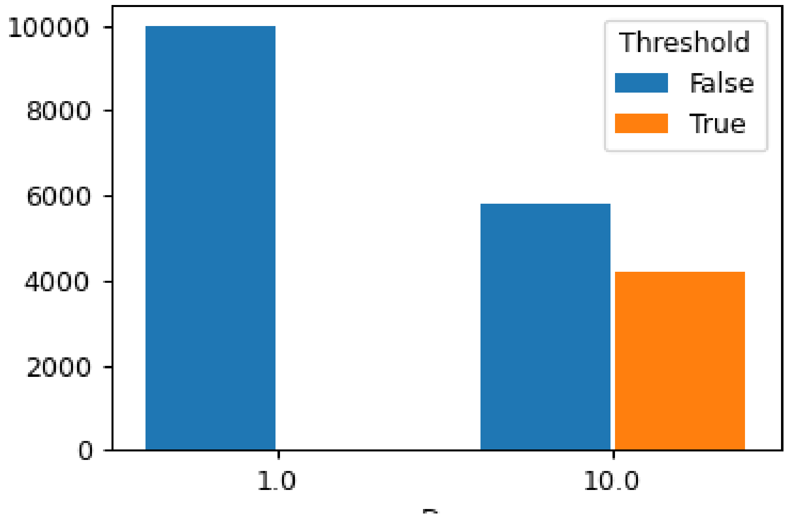
`Horizontal Facet`
: Make muliple plots, with each column representing a subset of the statistic with a different value for this variable.

`Vertical Facet`
: Make multiple plots, with each row representing a subset of the statistic with a different value for this variable.

`Hue Facet`
: Make multiple bars with different colors; each color represents a subset of the statistic with a different value for this variable.

`Error Statistic`
: A statistic to use to make the error bars. Must have the same variables as the statistic in **Statistic**.

`Subset`
: Plot only a subset of the statistic.



Density Plot

Plots a 2-dimensional density plot.

`X Channel, Y Channel`
: The channels to plot on the X and Y axes.

`X Scale, Y Scale`
: How to scale the X and Y axes of the plot.

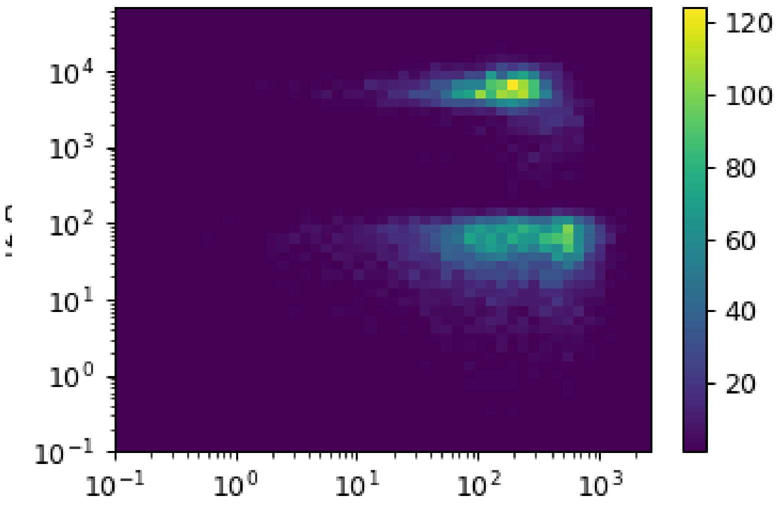
`Horizonal Facet`
: Make multiple plots. Each column has a unique value of this variable.

`Vertical Facet`
: Make multiple plots. Each row has a unique value of this variable.

`Color Scale`
: Scale the color palette and the color bar

`Tab Facet`
Make multiple plots in differen tabs; each tab's plot has a unique value of this variable.

`Subset`
Plot only a subset of the data in the experiment.



Export FCS

Exports FCS files from after this operation. Only really useful if you've done a calibration step or created derivative channels using the ratio option. As you set the options, the main plot shows a table of the files that will be created.

`Base`
`The prefix of the FCS file names`

`By`
A list of metadata attributes to aggregate the data before exporting. For example, if the experiment has two pieces of metadata, `Time` and `Dox`, setting **By** to `["Time", "Dox"]` will export one file for each subset of the data with a unique combination of `Time` and `Dox`.

`Keywords`
If you want to add more keywords to the FCS files' TEXT segment, specify them here.

`Export...`
Choose a folder and export the FCS files.

Histogram

Plots a histogram.

`Channel`
: The channel for the plot.

`Scale`
: How to scale the X axis of the plot.

`Horizonal Facet`
: Make multiple plots. Each column has a unique value of this variable.

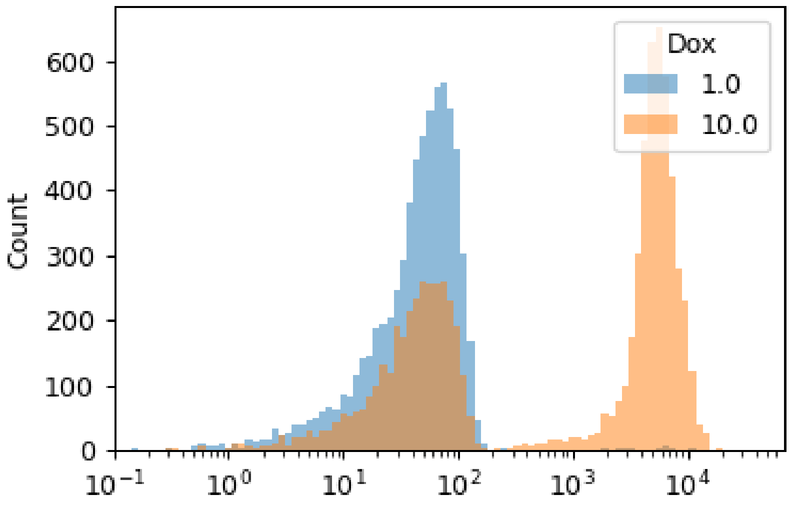
`Vertical Facet`
: Make multiple plots. Each row has a unique value of this variable.

`Color Facet`
: Plot with multiple colors. Each color has a unique value of this variable.

`Color Scale`
: If **Color Facet** is a numeric variable, use this scale for the color bar.

`Tab Facet`
: Make multiple plots in differen tabs; each tab's plot has a unique value of this variable.

`Subset`
: Plot only a subset of the data in the experiment.



2D Histogram

Plots a 2-dimensional histogram. Similar to a density plot, but the number of events in a bin change the bin's opacity, so you can use different colors.

`X Channel, Y Channel`
: The channels to plot on the X and Y axes.

`X Scale, Y Scale`
: How to scale the X and Y axes of the plot.

`Horizonal Facet`
: Make multiple plots. Each column has a unique value of this variable.

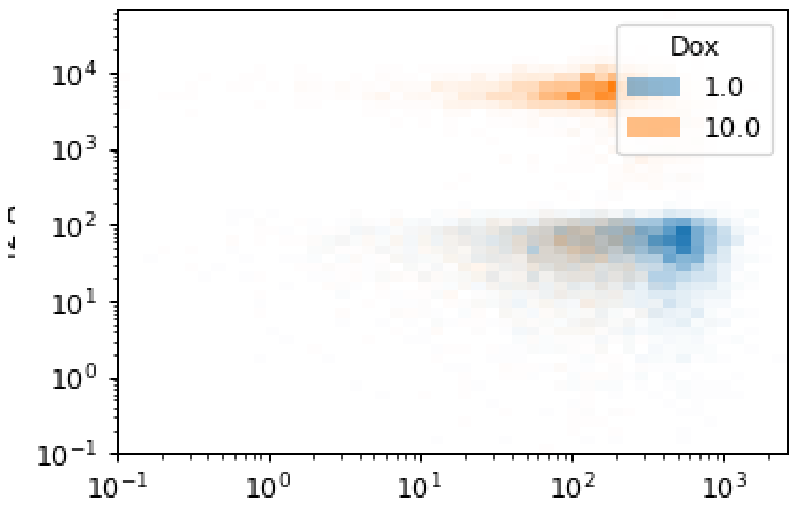
`Vertical Facet`
: Make multiple plots. Each row has a unique value of this variable.

`Color Facet`
: Plot with multiple colors. Each color has a unique value of this variable.

`Color Scale`
: If **Color Facet** is a numeric variable, use this scale for the color bar.

`Tab Facet`
: Make multiple plots in differen tabs; each tab's plot has a unique value of this variable.

`Subset`
: Plot only a subset of the data in the experiment.



1D Kernel Density Estimate

Plots a "smoothed" histogram.

`Channel`
The channel for the plot.

`Scale`
How to scale the X axis of the plot.

`Horizonal Facet`
Make multiple plots. Each column has a unique value of this variable.

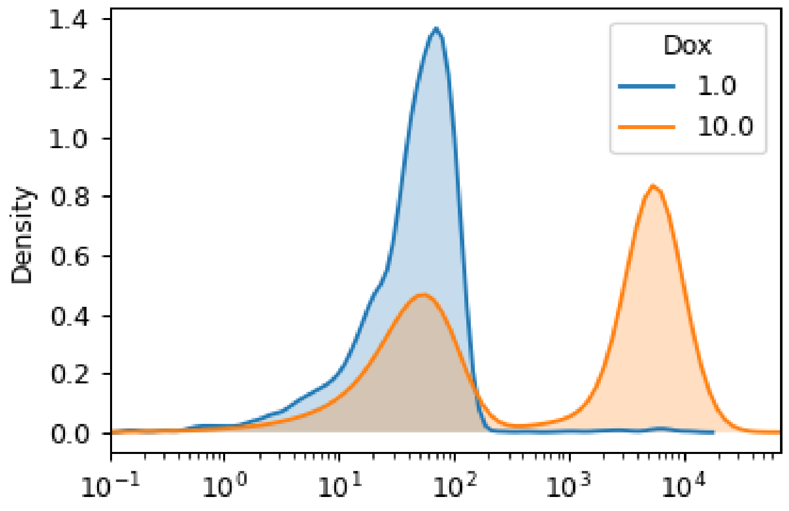
`Vertical Facet`
Make multiple plots. Each row has a unique value of this variable.

`Color Facet`
Plot with multiple colors. Each color has a unique value of this variable.

`Color Scale`
If **Color Facet** is a numeric variable, use this scale for the color bar.

`Tab Facet`
Make multiple plots in differen tabs; each tab's plot has a unique value of this variable.

`Subset`
Plot only a subset of the data in the experiment.



2D Kernel Density Estimate

Plots a 2-d kernel-density estimate. Sort of like a smoothed histogram. The density is visualized with a set of isolines.

`X Channel, Y Channel`
The channels to plot on the X and Y axes.

`X Scale, Y Scale`
How to scale the X and Y axes of the plot.

`Horizonal Facet`
Make multiple plots. Each column has a unique value of this variable.

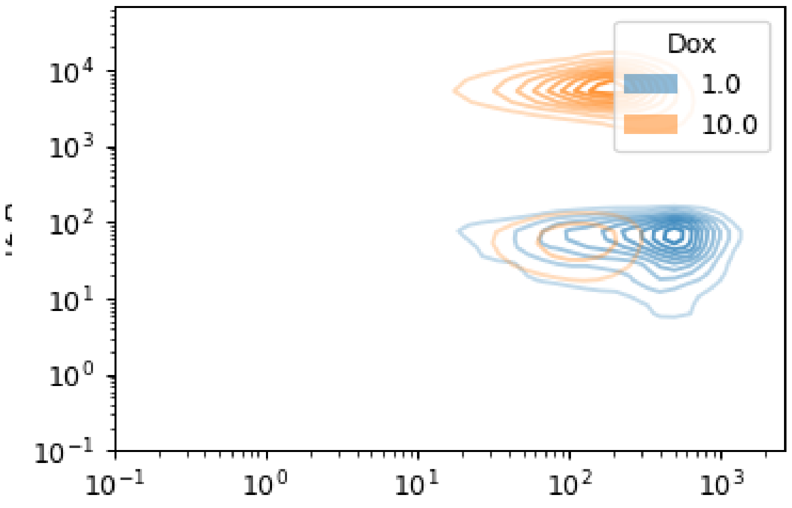
`Vertical Facet`
Make multiple plots. Each row has a unique value of this variable.

`Color Facet`
Plot with multiple colors. Each color has a unique value of this variable.

`Color Scale`
If **Color Facet** is a numeric variable, use this scale for the color bar.

`Tab Facet`
Make multiple plots in differen tabs; each tab's plot has a unique value of this variable.

`Subset`
Plot only a subset of the data in the experiment.



Parallel Coordinates Plot

Plots a parallel coordinates plot. PC plots are good for multivariate data; each vertical line represents one attribute, and one set of connected line segments represents one data point.

`Channels`
: The channels to plot, and their scales. Drag the blue dot to re-order.

`Add Channel, Remove Channel`
: Add or remove a channel

`Horizonal Facet`
: Make multiple plots. Each column has a unique value of this variable.

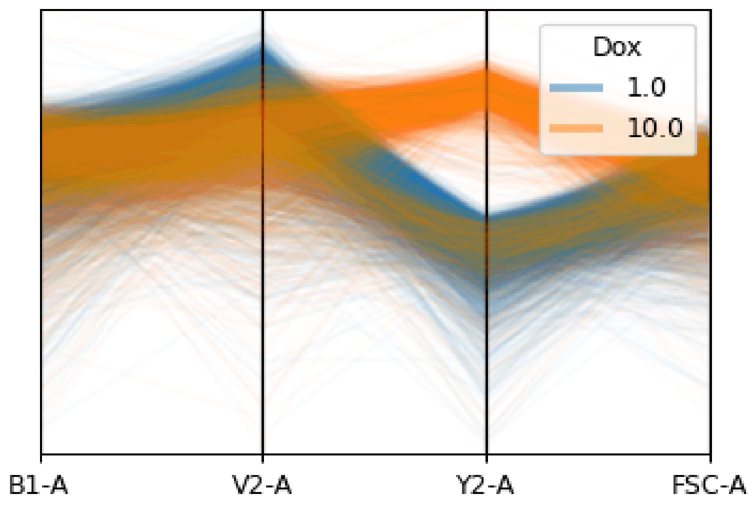
`Vertical Facet`
: Make multiple plots. Each row has a unique value of this variable.

`Color Facet`
: Plot different values of a condition with different colors.

`Color Scale`
: Scale the color palette and the color bar

`Tab Facet`
: Make multiple plots in differen tabs; each tab's plot has a unique value of this variable.

`Subset`
: Plot only a subset of the data in the experiment.



Radviz Plot

Plots a radviz plot. Radviz plots project multivariate plots into two dimensions. Good for looking for clusters.

`Channels`
: The channels to plot, and their scales. Drag the blue dot to re-order.

`Add Channel, Remove Channel`
: Add or remove a channel

`Horizonal Facet`
: Make multiple plots. Each column has a unique value of this variable.

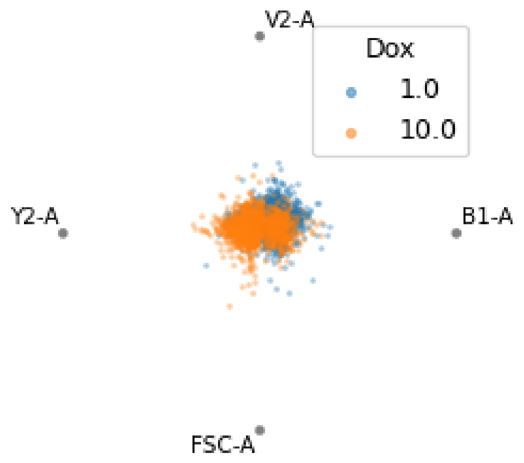
`Vertical Facet`
: Make multiple plots. Each row has a unique value of this variable.

`Color Facet`
: Plot different values of a condition with different colors.

`Color Scale`
: Scale the color palette and the color bar

`Tab Facet`
: Make multiple plots in differen tabs; each tab's plot has a unique value of this variable.

`Subset`
: Plot only a subset of the data in the experiment.



Scatterplot

Plot a scatterplot.

`X Channel, Y Channel`
The channels to plot on the X and Y axes.

`X Scale, Y Scale`
How to scale the X and Y axes of the plot.

`Horizonal Facet`
Make multiple plots. Each column has a unique value of this variable.

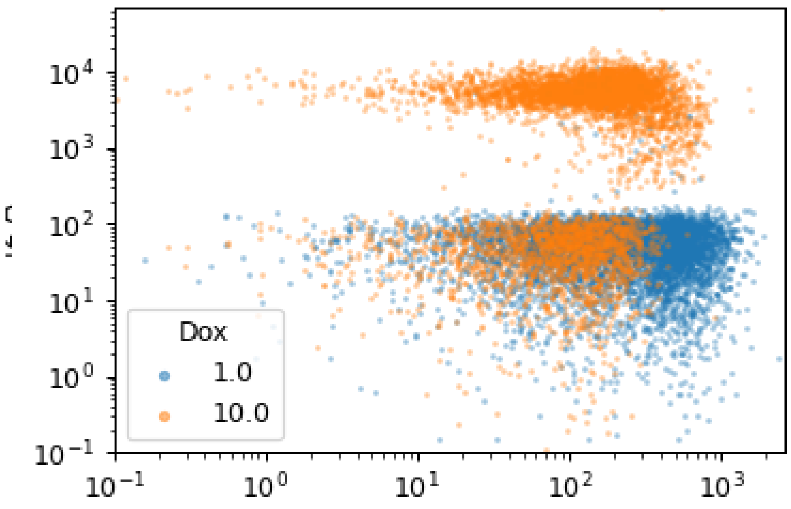
`Vertical Facet`
Make multiple plots. Each row has a unique value of this variable.

`Color Facet`
Plot with multiple colors. Each color has a unique value of this variable.

`Color Scale`
If **Color Facet** is a numeric variable, use this scale for the color bar.

`Tab Facet`
Make multiple plots in differen tabs; each tab's plot has a unique value of this variable.

`Subset`
Plot only a subset of the data in the experiment.



1D Statistics Plot

Plots a line plot of a statistic.

Each variable in the statistic (ie, each variable chosen in the statistic operation's **Group By**) must be set as **Variable** or as a facet.

`Statistic`
: Which statistic to plot.

`Variable`
: The statistic variable put on the X axis. Must be numeric.

`X Scale, Y Scale`
: How to scale the X and Y axes.

`Horizontal Facet`
: Make muliple plots, with each column representing a subset of the statistic with a different value for this variable.

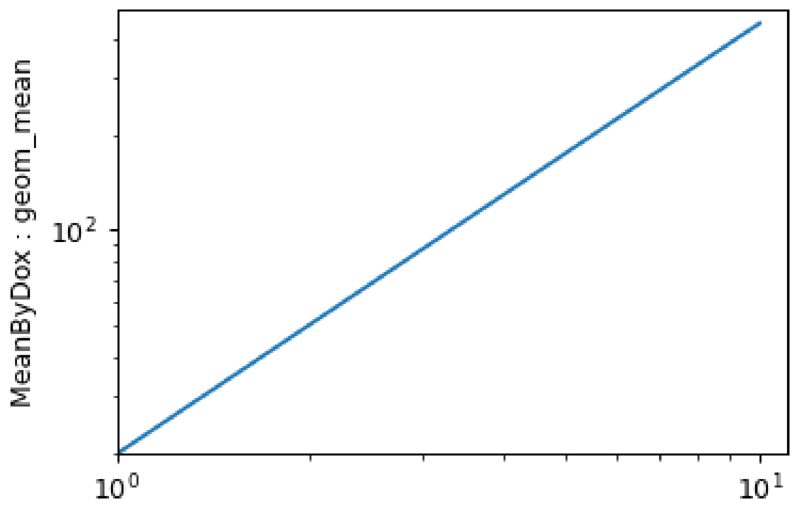
`Vertical Facet`
: Make multiple plots, with each row representing a subset of the statistic with a different value for this variable.

`Hue Facet`
: Make multiple bars with different colors; each color represents a subset of the statistic with a different value for this variable.

`Color Scale`
: If **Color Facet** is a numeric variable, use this scale for the color bar.

`Error Statistic`
: A statistic to use to make the error bars. Must have the same variables as the statistic in **Statistic**.

`Subset`
: Plot only a subset of the statistic.



2D Statistics Plot

Plot two statistics on a scatter plot. A point (X,Y) is drawn for every pair of elements with the same value of **Variable**; the X value is from ** X statistic** and the Y value is from **Y statistic**.

`X Statistic`
: Which statistic to plot on the X axis.

`Y Statistic`
: Which statistic to plot on the Y axis. Must have the same indices as **X Statistic**.

`X Scale, Y Scale`
: How to scale the X and Y axes.

`Variable`
: The statistic variable to put on the plot.

`Horizontal Facet`
: Make muliple plots, with each column representing a subset of the statistic with a different value for this variable.

`Vertical Facet`
: Make multiple plots, with each row representing a subset of the statistic with a different value for this variable.

`Color Facet`
: Make lines on the plot with different colors; each color represents a subset of the statistic with a different value for this variable.

`Color Scale`
: If **Color Facet** is a numeric variable, use this scale for the color bar.

`X Error Statistic`
: A statistic to use to make error bars in the X direction. Must have the same indices as the statistic in **X Statistic**.

`Y Error Statistic`
: A statistic to use to make error bars in the Y direction. Must have the same indices as the statistic in **Y Statistic**.

`Subset`
: Plot only a subset of the statistic.

Table

Make a table out of a statistic. The table can then be exported.

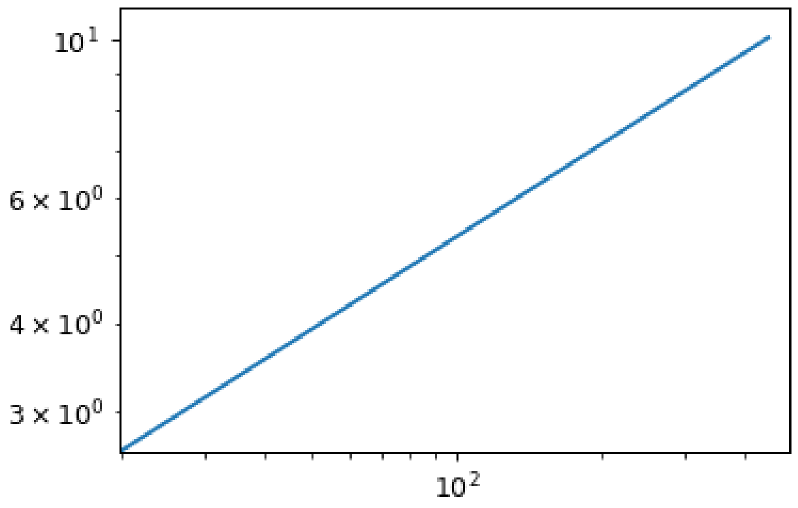
`Statistic`
: Which statistic to view.

`Rows`
: Which variable to use for the rows

`Subrows`
: Which variable to use for subrows.

`Columns`
: Which variable to use for the columns.

`Subcolumns`
: Which variable to use for the subcolumns.



Export
Export the table to a CSV file.

Violin Plot

Plots a violin plot, which is a nice way to compare several distributions.

X Variable
The variable to compare on the X axis.

Y Channel
The channel to plot on the Y axis.

Y Channel Scale
How to scale the Y axis of the plot.

Horizonal Facet
Make multiple plots. Each column has a unique value of this variable.

Vertical Facet
Make multiple plots. Each row has a unique value of this variable.

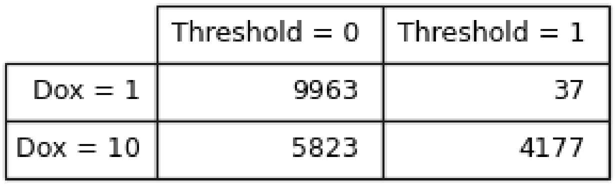
Color Facet
Plot with multiple colors. Each color has a unique value of this variable.

Color Scale
If **Color Facet** is a numeric variable, use this scale for the color bar.

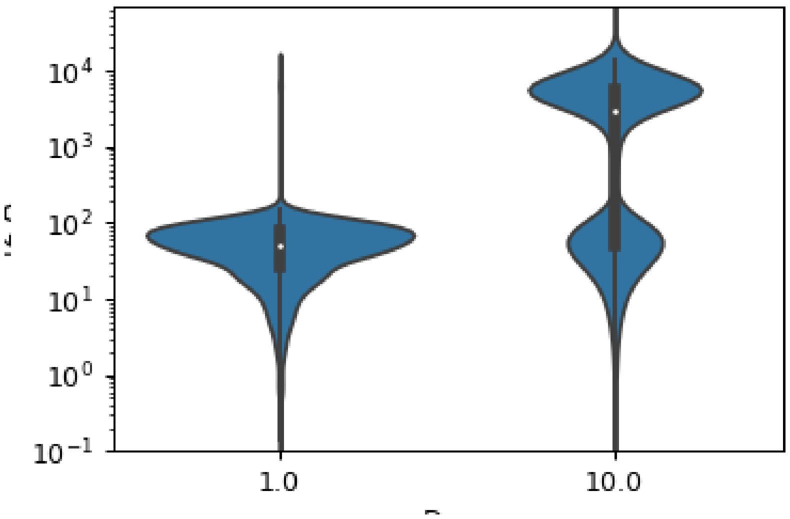
Tab Facet
Make multiple plots in differen tabs; each tab's plot has a unique value of this variable.

Subset
Plot only a subset of the data in the experiment.

There are also a few odds and ends:

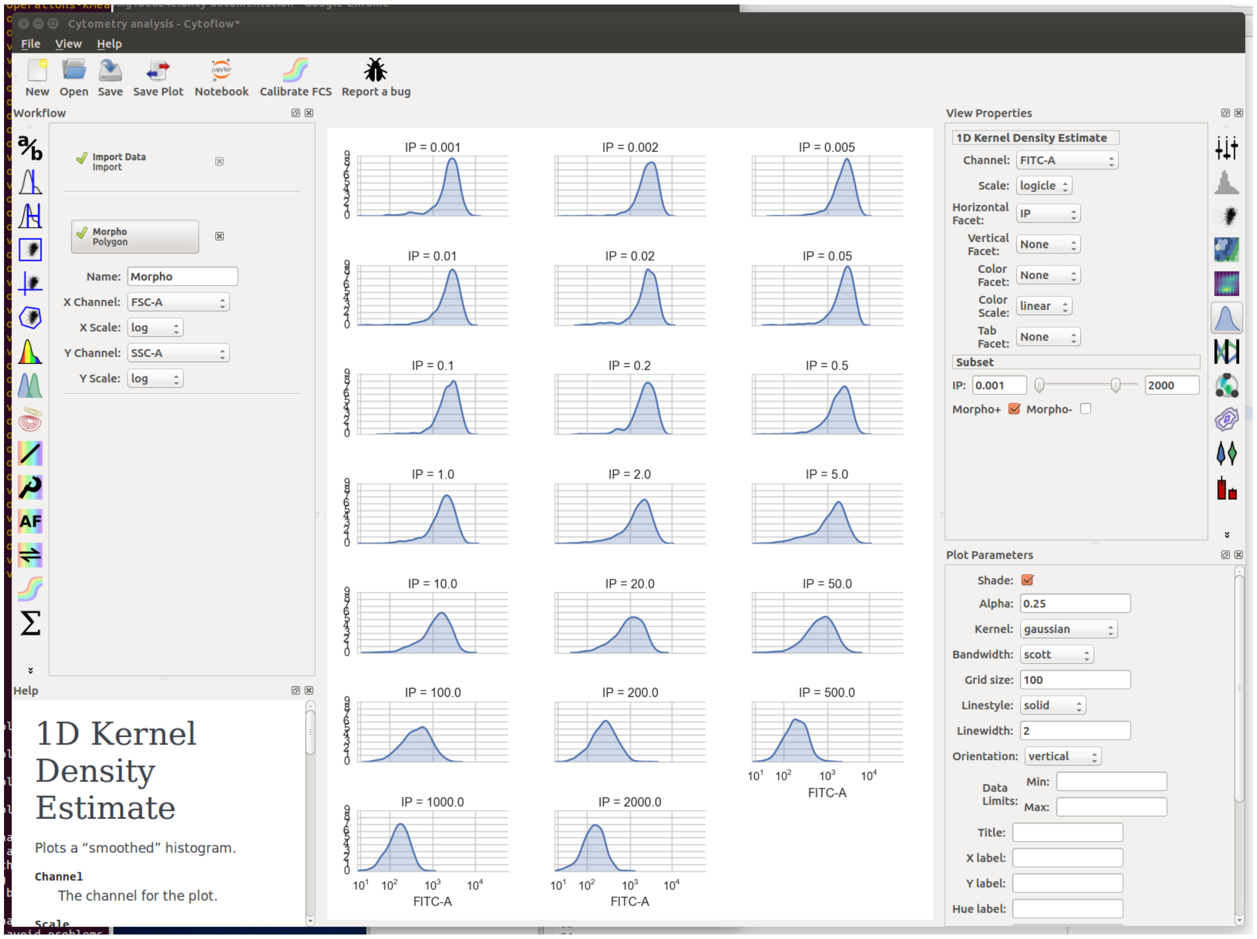


	Threshold = 0	Threshold = 1
Dox = 1	9963	37
Dox = 10	5823	4177



5.1.20 Screenshots of the Cytoflow GUI

The main GUI.



Import Data
Import

Morpho
Polygon

Name: Morpho
X Channel: FSC-A
X Scale: log
Y Channel: SSC-A
Y Scale: log

1D Kernel Density Estimate
Channel: FITC-A
Scale: logicle
Horizontal Facet: IP
Vertical Facet: None
Color Facet: None
Color Scale: linear
Tab Facet: None

Subset
IP: 0.001 2000
Morpho+ ✓ Morpho- ☐

Shade: ✓
Alpha: 0.25
Kernel: gaussian
Bandwidth: scott
Grid size: 100
Linestyle: solid
Linewidth: 2
Orientation: vertical
Data Limits: Min: Max:
Title:
X label:
Y label:
Hue label:

1D Kernel Density Estimate

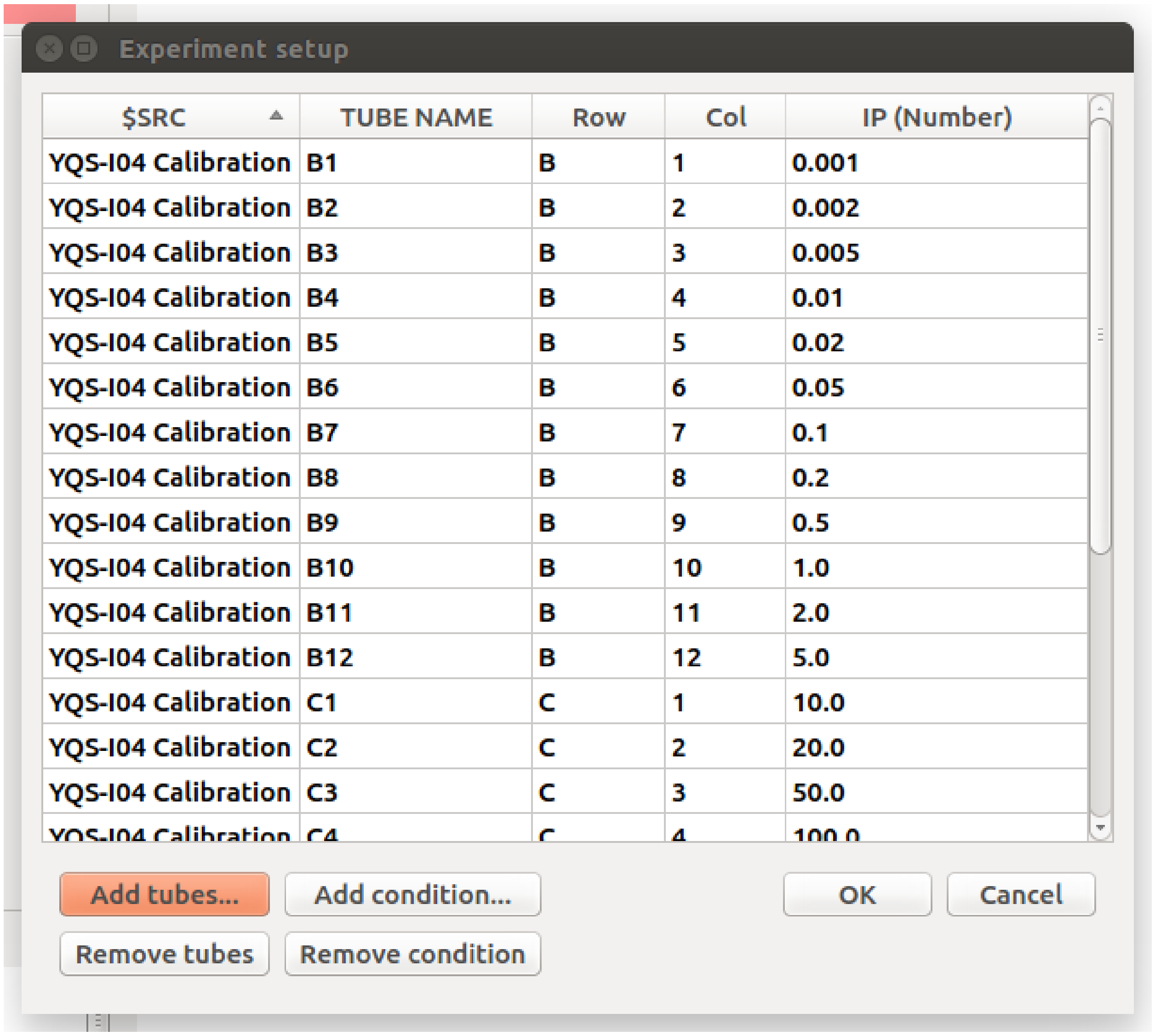
Plots a “smoothed” histogram.

Channel

The channel for the plot.

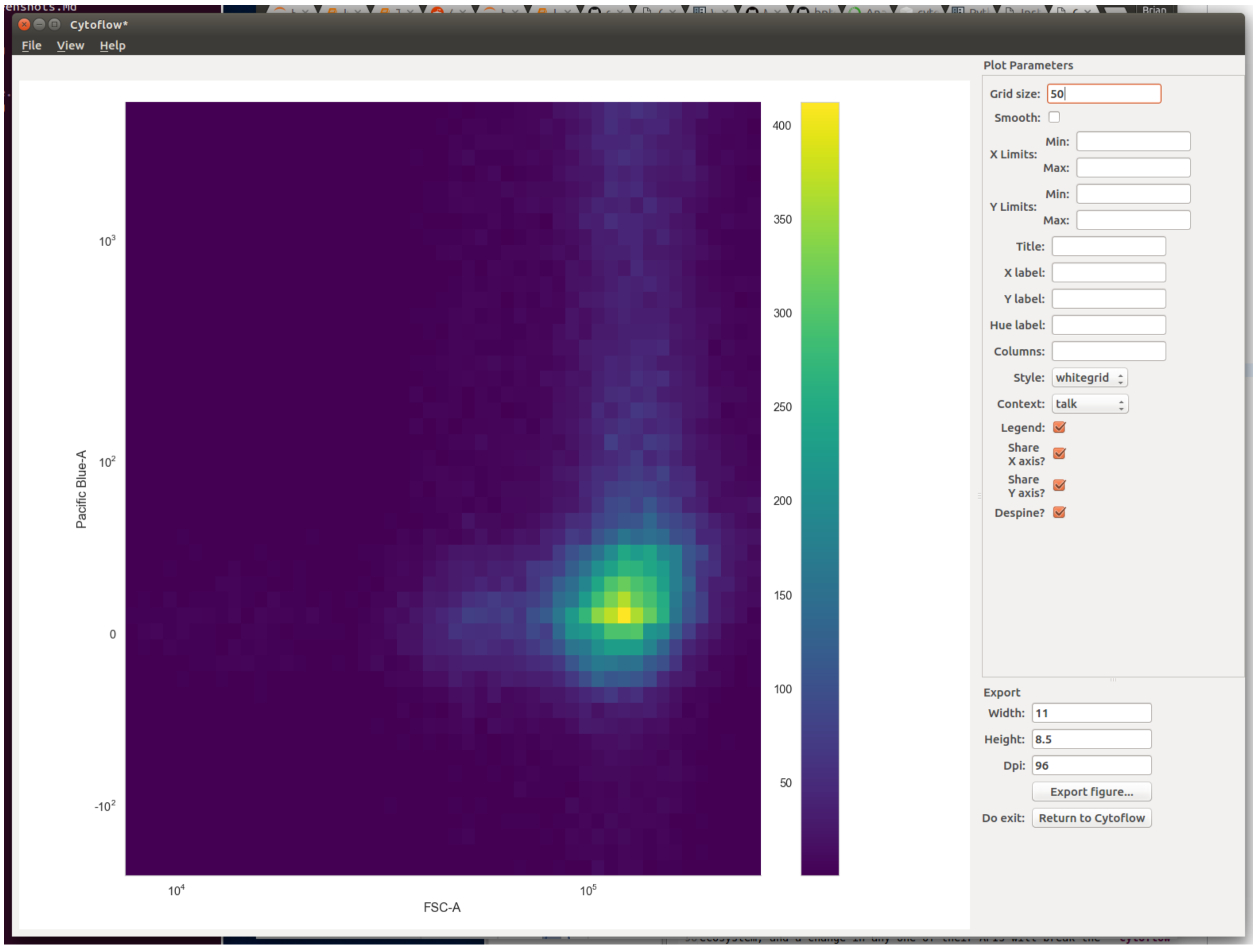
Scale

The experiment editor.



$SRC	TUBE NAME	Row	Col	IP (Number)
YQS-I04 Calibration	B1	B	1	0.001
YQS-I04 Calibration	B2	B	2	0.002
YQS-I04 Calibration	B3	B	3	0.005
YQS-I04 Calibration	B4	B	4	0.01
YQS-I04 Calibration	B5	B	5	0.02
YQS-I04 Calibration	B6	B	6	0.05
YQS-I04 Calibration	B7	B	7	0.1
YQS-I04 Calibration	B8	B	8	0.2
YQS-I04 Calibration	B9	B	9	0.5
YQS-I04 Calibration	B10	B	10	1.0
YQS-I04 Calibration	B11	B	11	2.0
YQS-I04 Calibration	B12	B	12	5.0
YQS-I04 Calibration	C1	C	1	10.0
YQS-I04 Calibration	C2	C	2	20.0
YQS-I04 Calibration	C3	C	3	50.0
YQS-I04 Calibration	C4	C	4	100.0

The plot editor.



Plot Parameters

Grid size: 50

Smooth: ☐

X Limits: Min: Max:

Y Limits: Min: Max:

Title:

X label:

Y label:

Hue label:

Columns:

Style: whitegrid

Context: talk

Legend: ✓

Share X axis? ✓

Share Y axis? ✓

Despine? ✓

Export

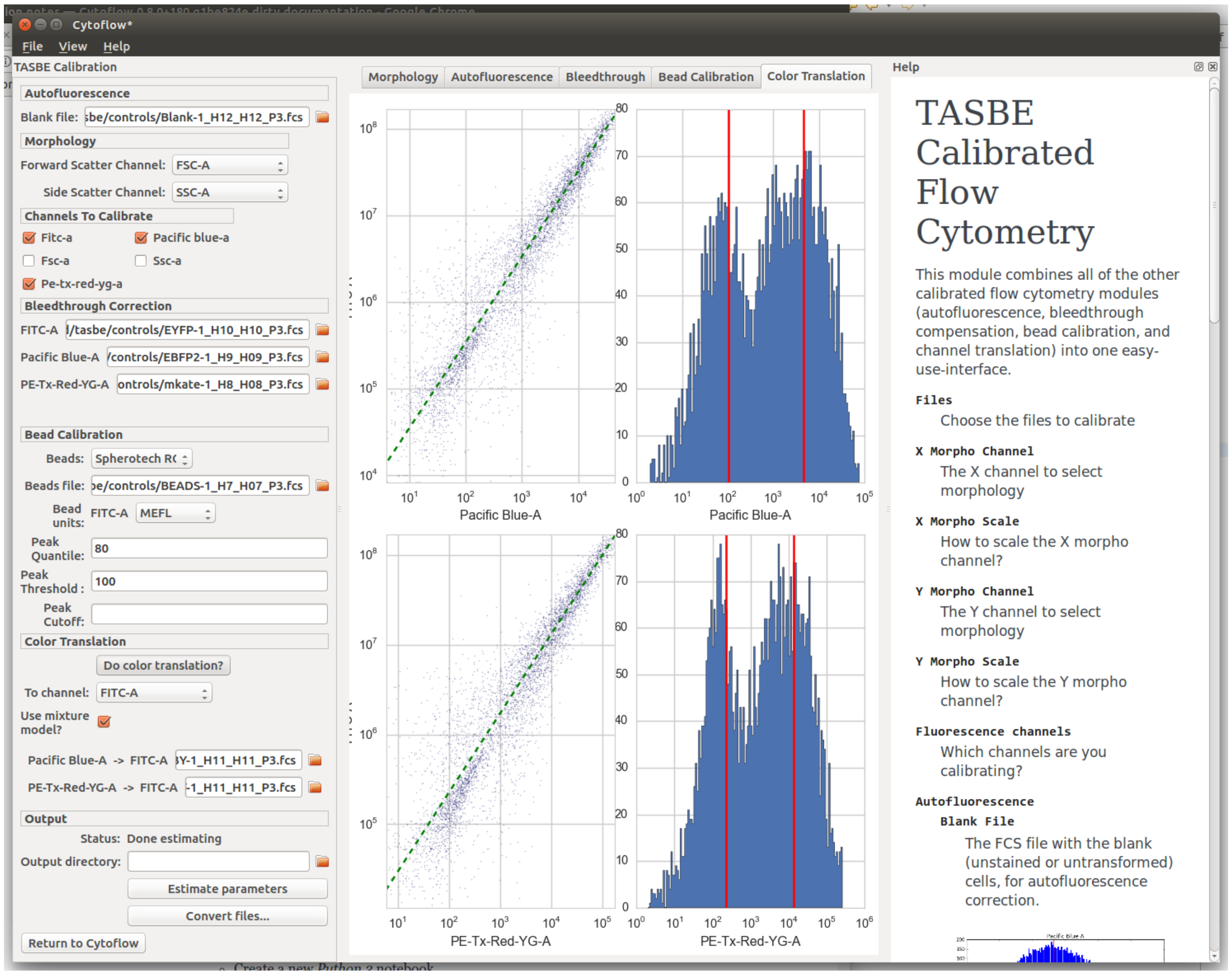
Width: 11

Height: 8.5

Dpi: 96

Do exit:

Convert FCS files to calibrated units.



TASBE Calibration

Autofluorescence

Blank file: sbe/controls/Blank-1_H12_H12_P3.fcs

Morphology

Forward Scatter Channel: FSC-A

Side Scatter Channel: SSC-A

Channels To Calibrate

[✓ Fitc-a] [✓ Pacific blue-a] Fsc-a Ssc-a [✓ Pe-tx-red-yg-a]

Bleedthrough Correction

FITC-A /tasbe/controls/EYFP-1_H10_H10_P3.fcs

Pacific Blue-A /controls/EBFP2-1_H9_H09_P3.fcs

PE-Tx-Red-YG-A ontrols/mkate-1_H8_H08_P3.fcs

Bead Calibration

Beads: Spherotech RC

Beads file: be/controls/BEADS-1_H7_H07_P3.fcs

Bead units: FITC-A MEFL

Peak Quantile: 80

Peak Threshold : 100

Peak Cutoff:

Color Translation

Do color translation?

To channel: FITC-A

Use mixture model? [✓]

Pacific Blue-A -> FITC-A Y-1_H11_H11_P3.fcs

PE-Tx-Red-YG-A -> FITC-A -1_H11_H11_P3.fcs

Output

Status: Done estimating

Output directory:

Estimate parameters

Convert files...

Return to Cytoflow

TASBE Calibrated Flow Cytometry

This module combines all of the other calibrated flow cytometry modules (autofluorescence, bleedthrough compensation, bead calibration, and channel translation) into one easy-use-interface.

Files
Choose the files to calibrate

X Morpho Channel
The X channel to select morphology

X Morpho Scale
How to scale the X morpho channel?

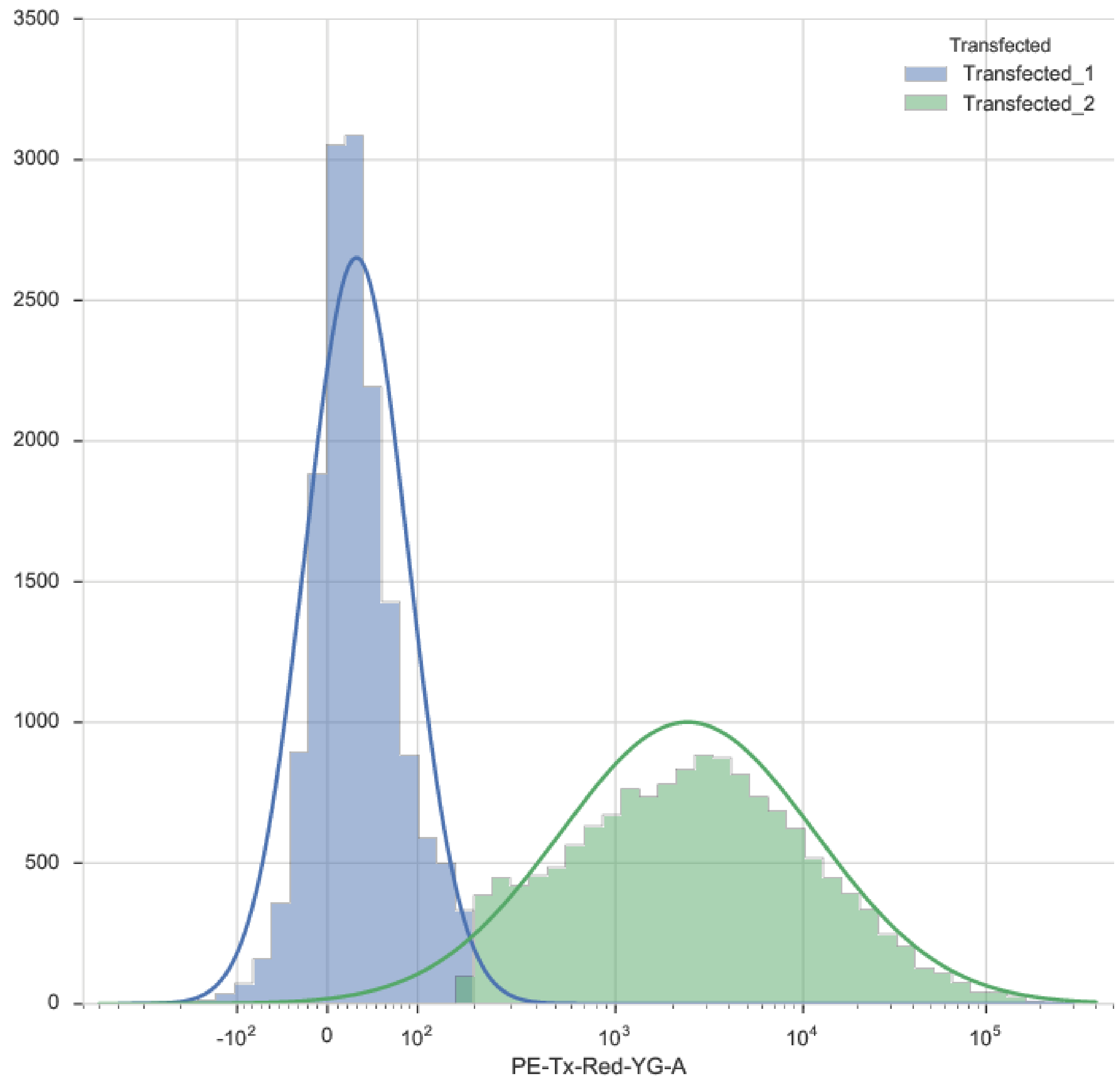
Y Morpho Channel
The Y channel to select morphology

Y Morpho Scale
How to scale the Y morpho channel?

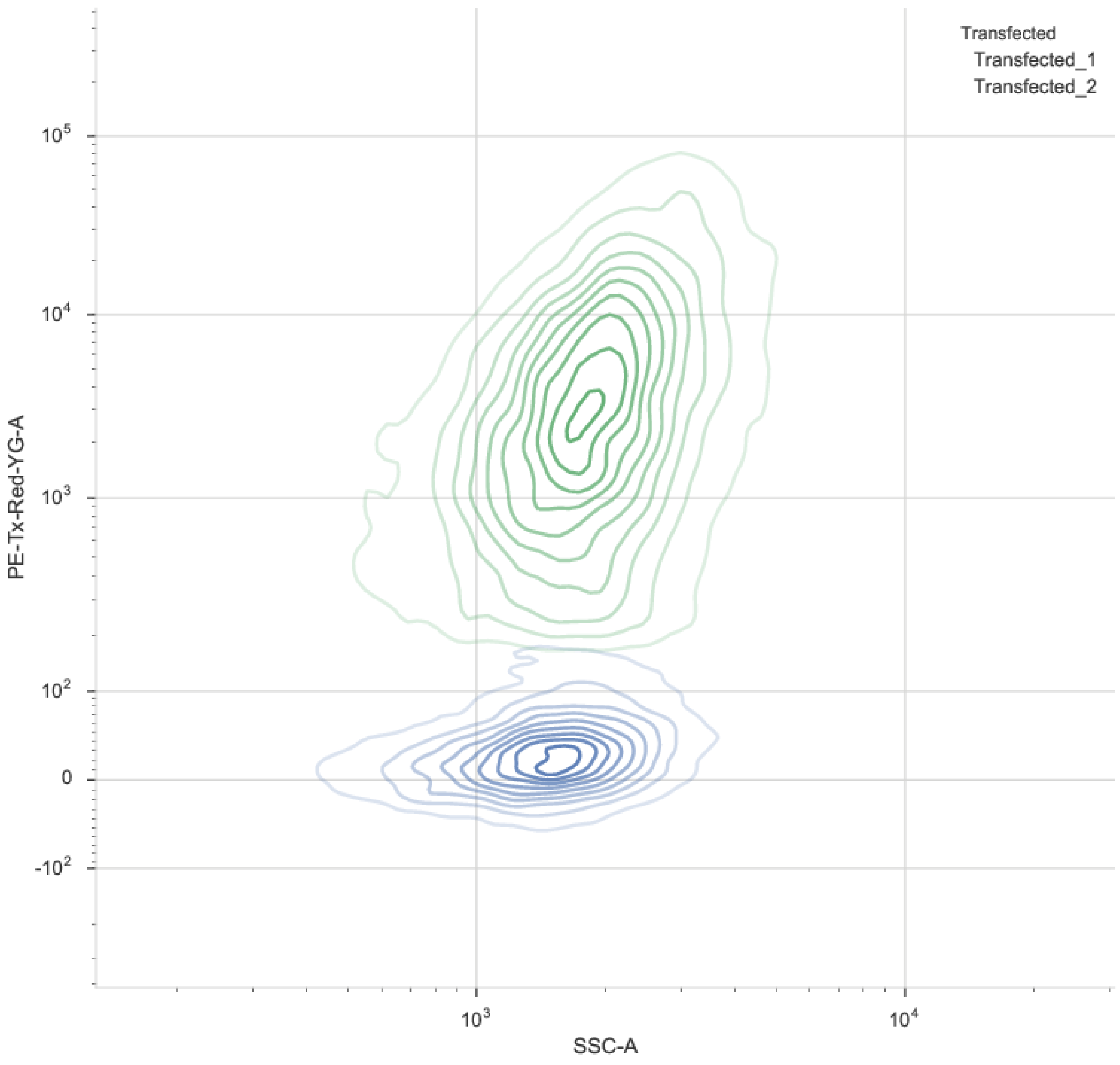
Fluorescence channels
Which channels are you calibrating?

Autofluorescence
Blank File
The FCS file with the blank (unstained or untransformed) cells, for autofluorescence correction.

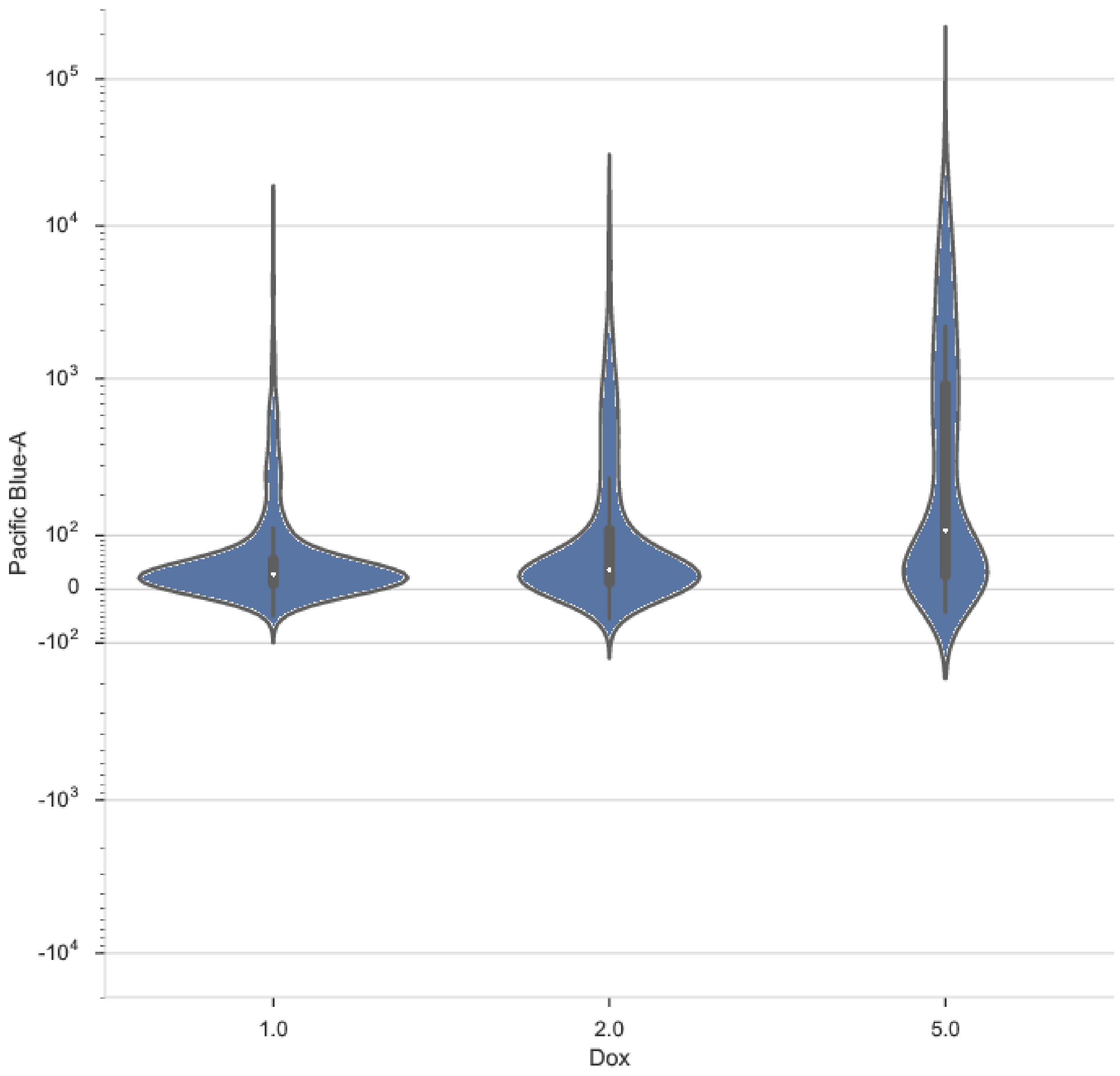
One-dimensional Gaussian mixture models.



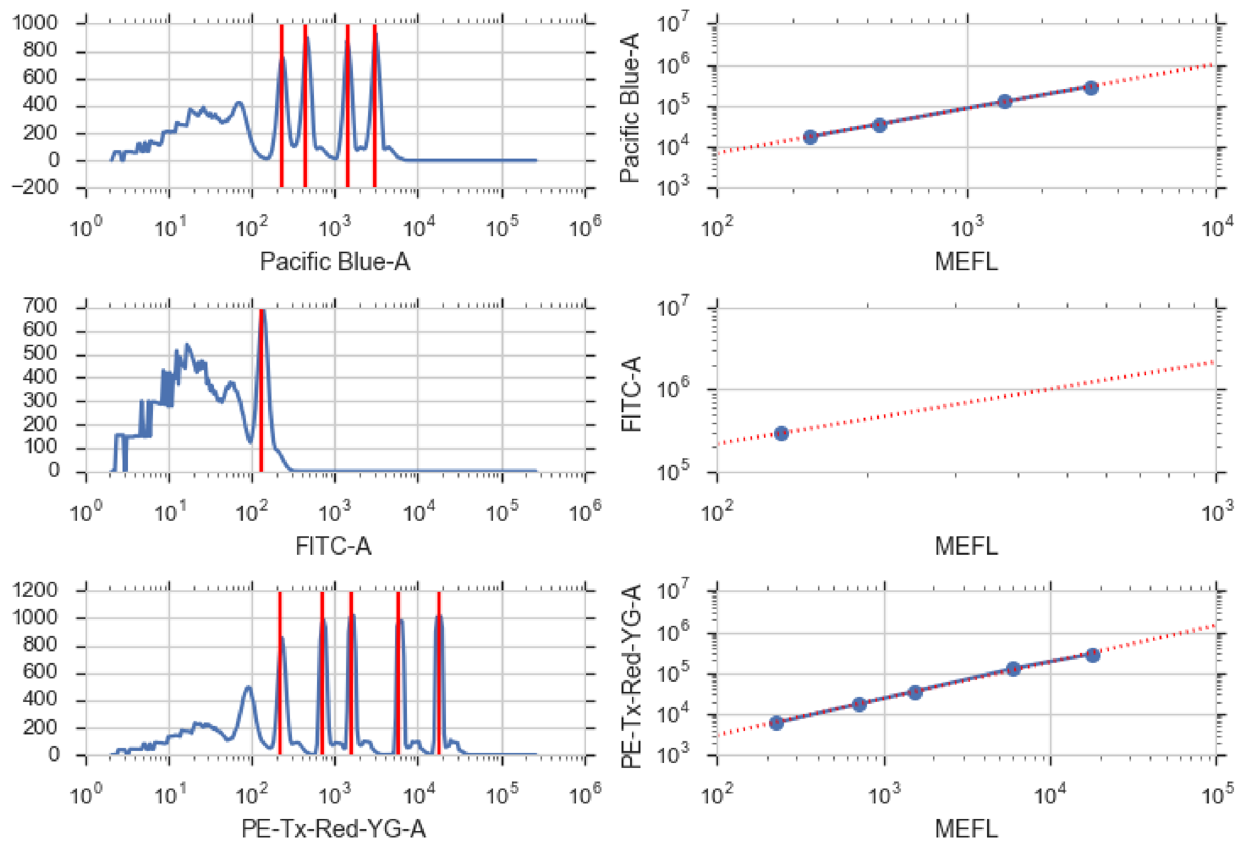
Two-dimensional kernel density estimates.



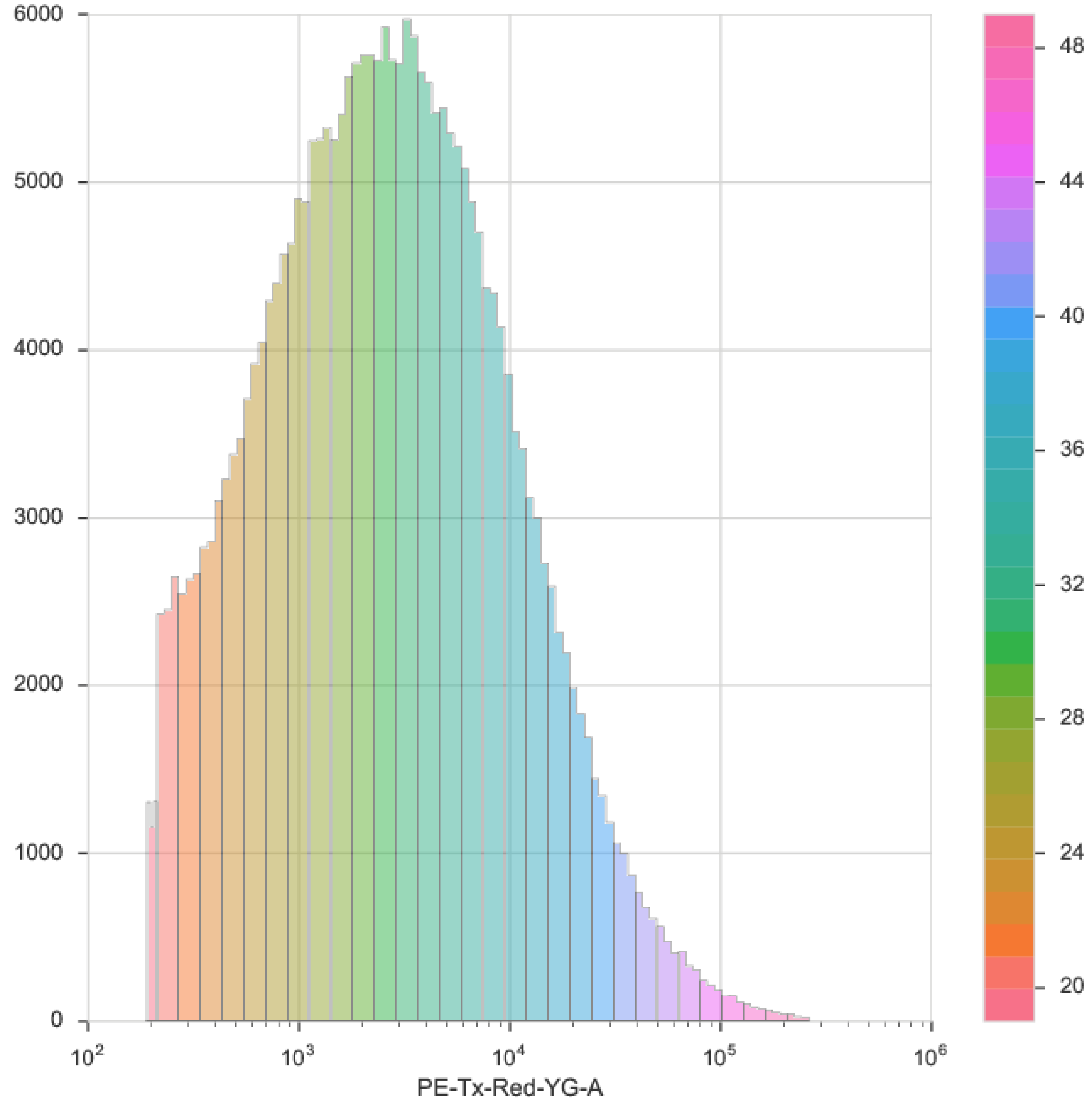
Violin plots to compare distributions.



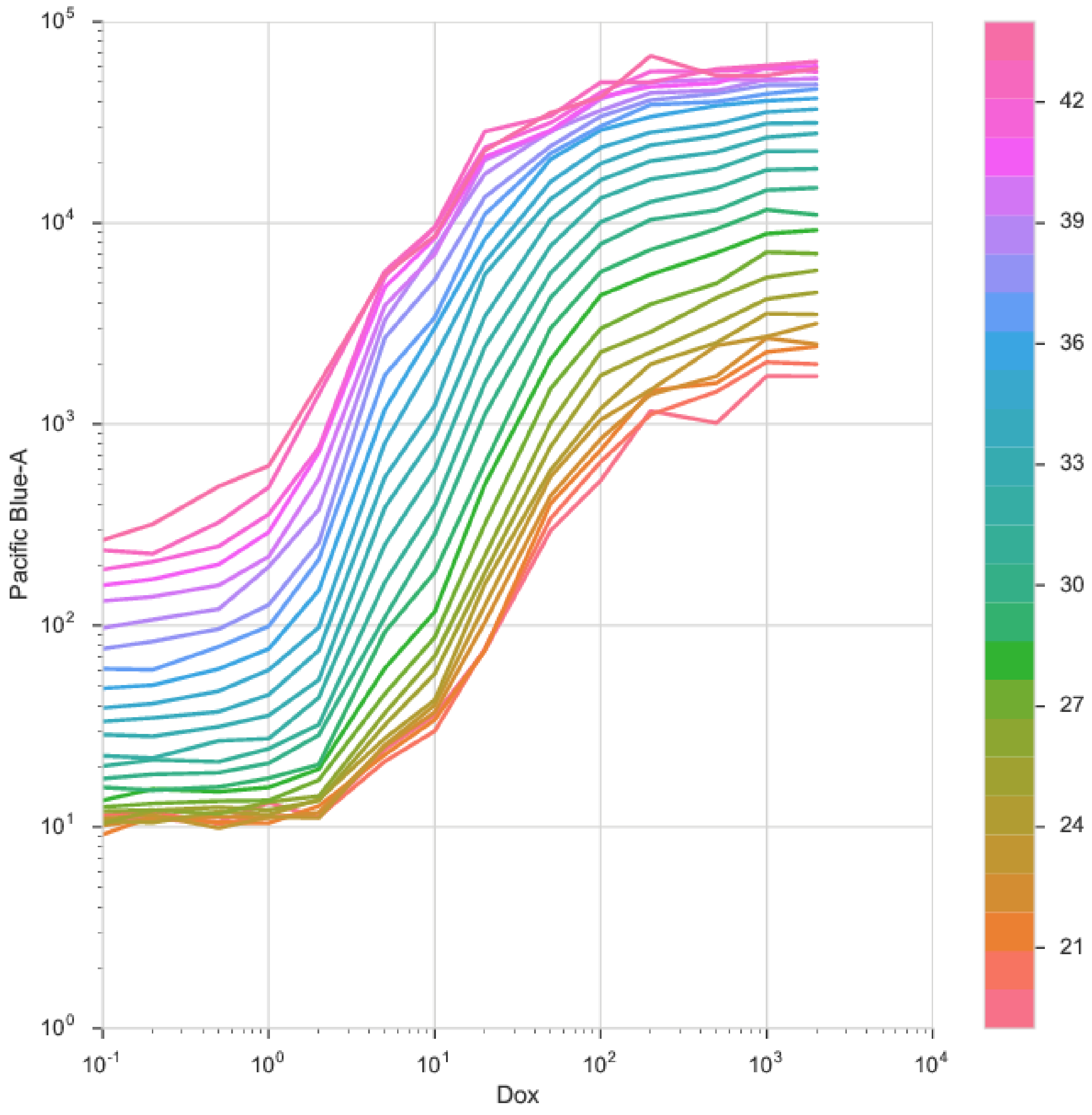
Calibrate raw data to physical calibrants like beads.



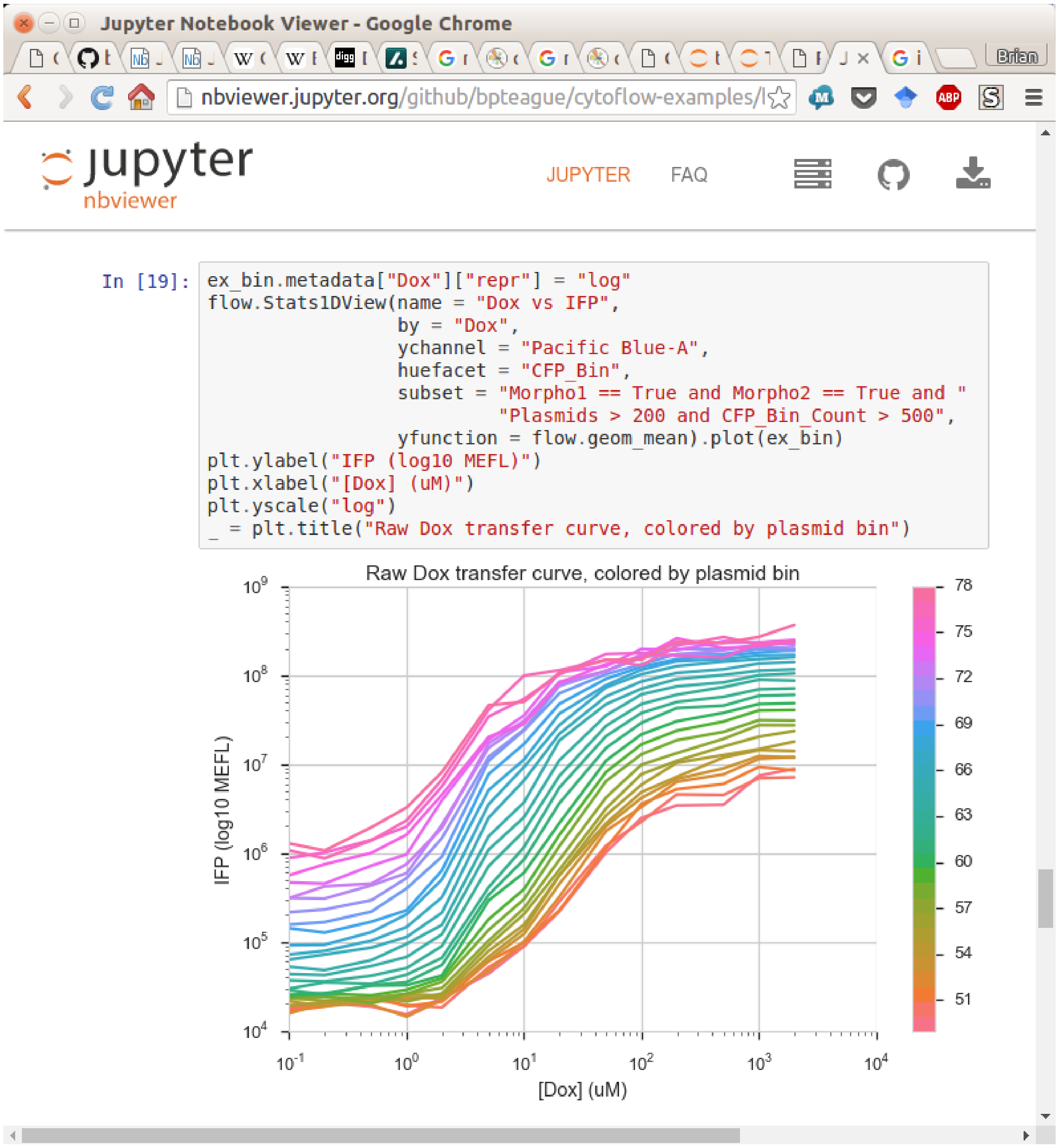
Bin data to analyze subsets.



Summarize data with statistics plots.



Export straight to a Jupyter notebook.


Jupyter Notebook Viewer - Google Chrome
nbviewer.jupyter.org/github/bpteague/cytoflow-examples/
jupyter nbviewer
JUPYTER
FAQ
Raw Dox transfer curve, colored by plasmid bin
IFP (log10 MEFL)
[Dox] (uM)

```
In [19]: ex_bin.metadata["Dox"]["repr"] = "log"
         flow.Stats1DView(name = "Dox vs IFP",
                          by = "Dox",
                          ychannel = "Pacific Blue-A",
                          huefacet = "CFP_Bin",
                          subset = "Morpho1 == True and Morpho2 == True and "
                                   "Plasmids > 200 and CFP_Bin_Count > 500",
                          yfunction = flow.geom_mean).plot(ex_bin)
         plt.ylabel("IFP (log10 MEFL)")
         plt.xlabel("[Dox] (uM)")
         plt.yscale("log")
         _ = plt.title("Raw Dox transfer curve, colored by plasmid bin")
```

5.1.21 Frequently Asked Questions

These are some questions that are commonly asked about Cytoflow's GUI, especially by new users.

I've made one of the panels (eg, the View settings panel) disappear. How do I make it come back?

You can restore them by going to the `View` menu and toggling the checkbox next to the panel that you made disappear.

I have some information in the FCS files' metadata that I'd like to use as experimental variables. How do I "import" them?

In the **Experiment Setup** dialog:

1. Click **Add a variable**
2. Select **FCS Metadata** as the variable type
3. Choose the metadata you want from the **Name** dropdown
4. *Change* the **Type** to something appropriate – probably **Number** or **Category**

5.1.22 Tutorials

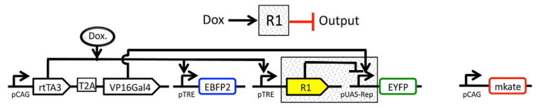
The following are step-by-step examples to get you started using Cytoflow. They include both some basic tutorials as well as examples of some more advanced analyses.

Easy tutorials

Advanced tutorials

Tutorial: Transcriptional Repressor Characterization

This example demonstrates using `cytoflow` to use calibrated flow cytometry to characterize a transcriptional repressor in a mammalian multi-plasmid system. The implementation closely follows that described in Beal et al and its implementation in Davidsohn et al. The experiment whose data we'll be analyzing characterizes a TALE transcriptional repressor (TAL14, from Li et al) using a multi-plasmid transient transfection in mammalian cells, depicted below:

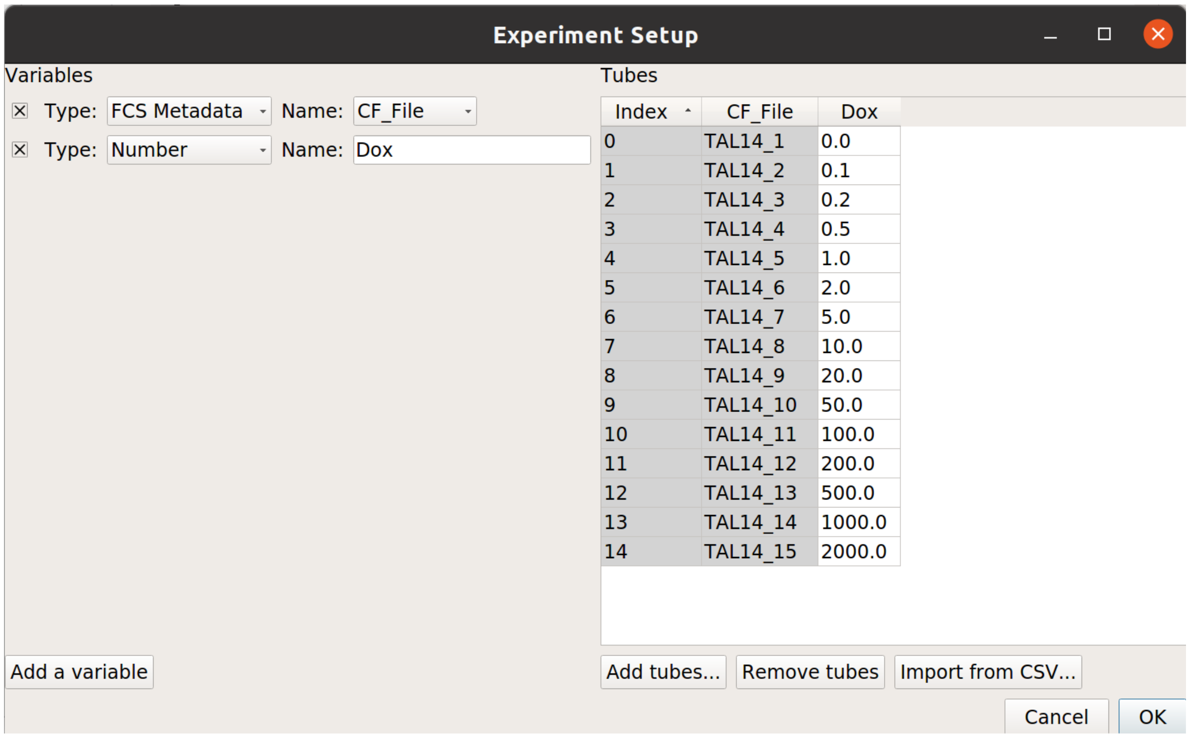


The small molecule doxycycline ("Dox") drives the transcriptional activator rtTA3 to activate the transcriptional repressor ("R1" in the diagram), which then represses output of the yellow fluorescent protein EYFP. rtTA3 also drives expression of a blue fluorescent protein, eBFP, which serves as a proxy for the amount of repressor. Finally, since we're doing transient transfection, there's a huge amount of variability in the level of transfection; we measure transfection level with a constitutively expressed red fluorescent protein, mKate.

If you'd like to follow along, you can do so by downloading one of the **cytoflow-#####-examples-advanced.zip** files from the Cytoflow releases page on GitHub. The files are in the **tasbe/** subdirectory.

Setup

- Start Cytoflow. Under the **Import Data** operation, choose **Set up experiment. . .**
- Add one variable, **Dox** – make it a **Number**.
- Load the `TAL14....fcs` files from the `tasbe/` subdirectory, and fill in the `Dox` concentrations as per the table below:



Experiment Setup

Variables

Type: FCS Metadata Name: CF_File

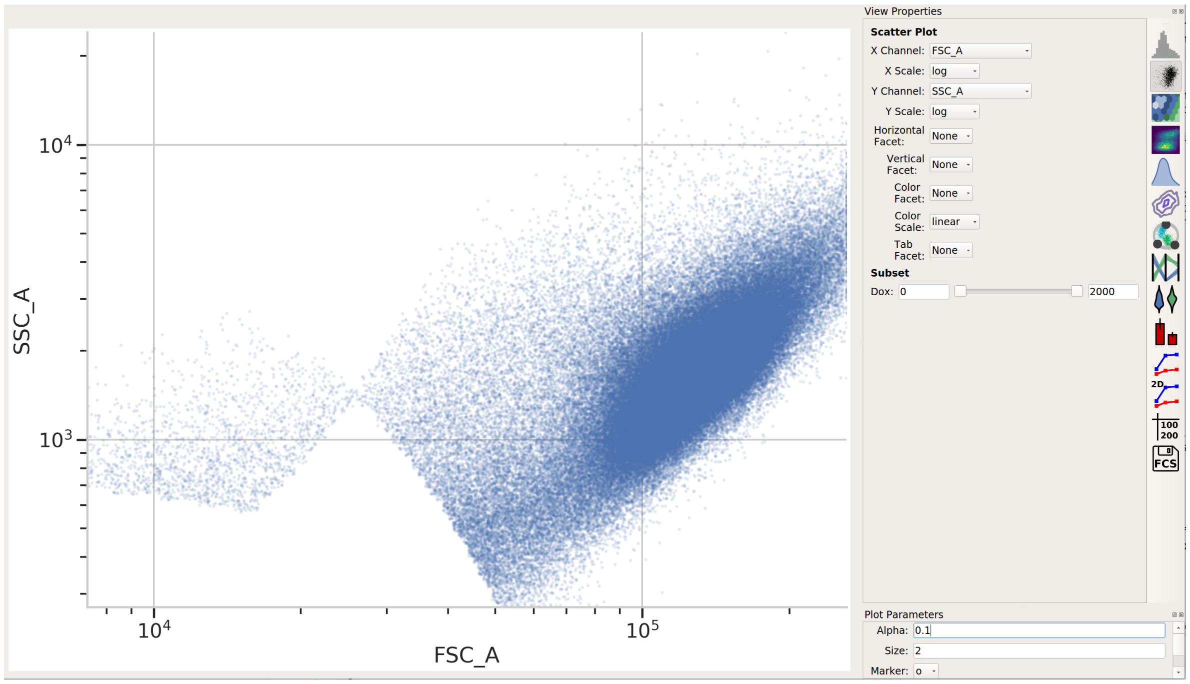
Type: Number Name: Dox

Tubes

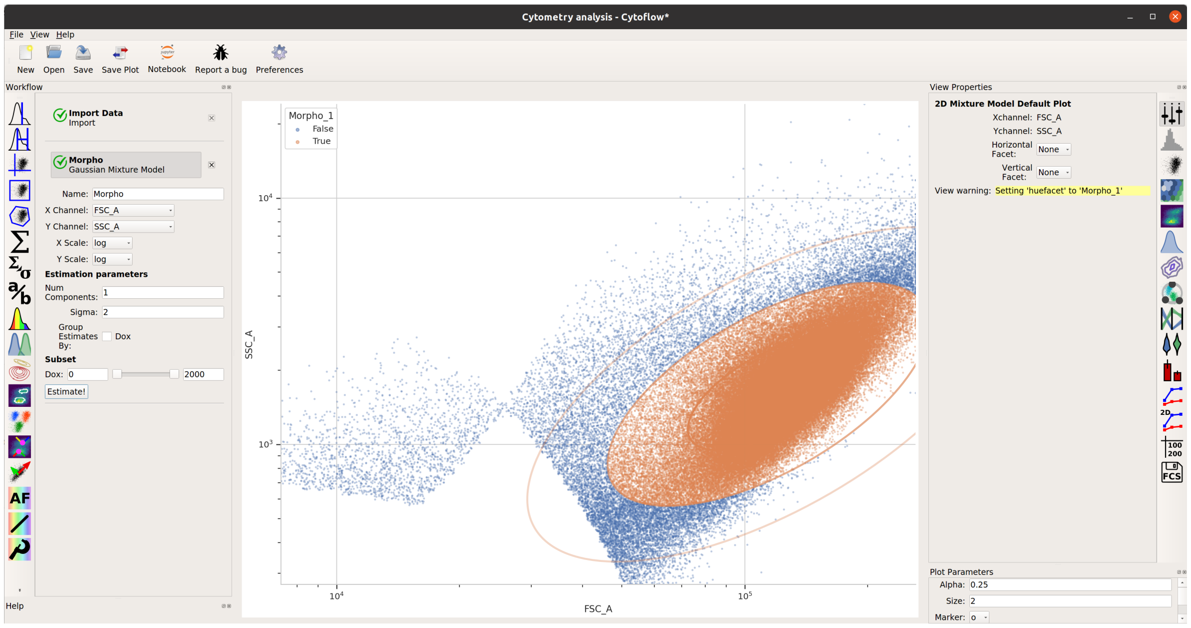
Index	CF_File	Dox
0	TAL14_1	0.0
1	TAL14_2	0.1
2	TAL14_3	0.2
3	TAL14_4	0.5
4	TAL14_5	1.0
5	TAL14_6	2.0
6	TAL14_7	5.0
7	TAL14_8	10.0
8	TAL14_9	20.0
9	TAL14_10	50.0
10	TAL14_11	100.0
11	TAL14_12	200.0
12	TAL14_13	500.0
13	TAL14_14	1000.0
14	TAL14_15	2000.0

Gate out the debris

- Use a scatter plot to look at the `FSC_A` and `SSC_A` channels:

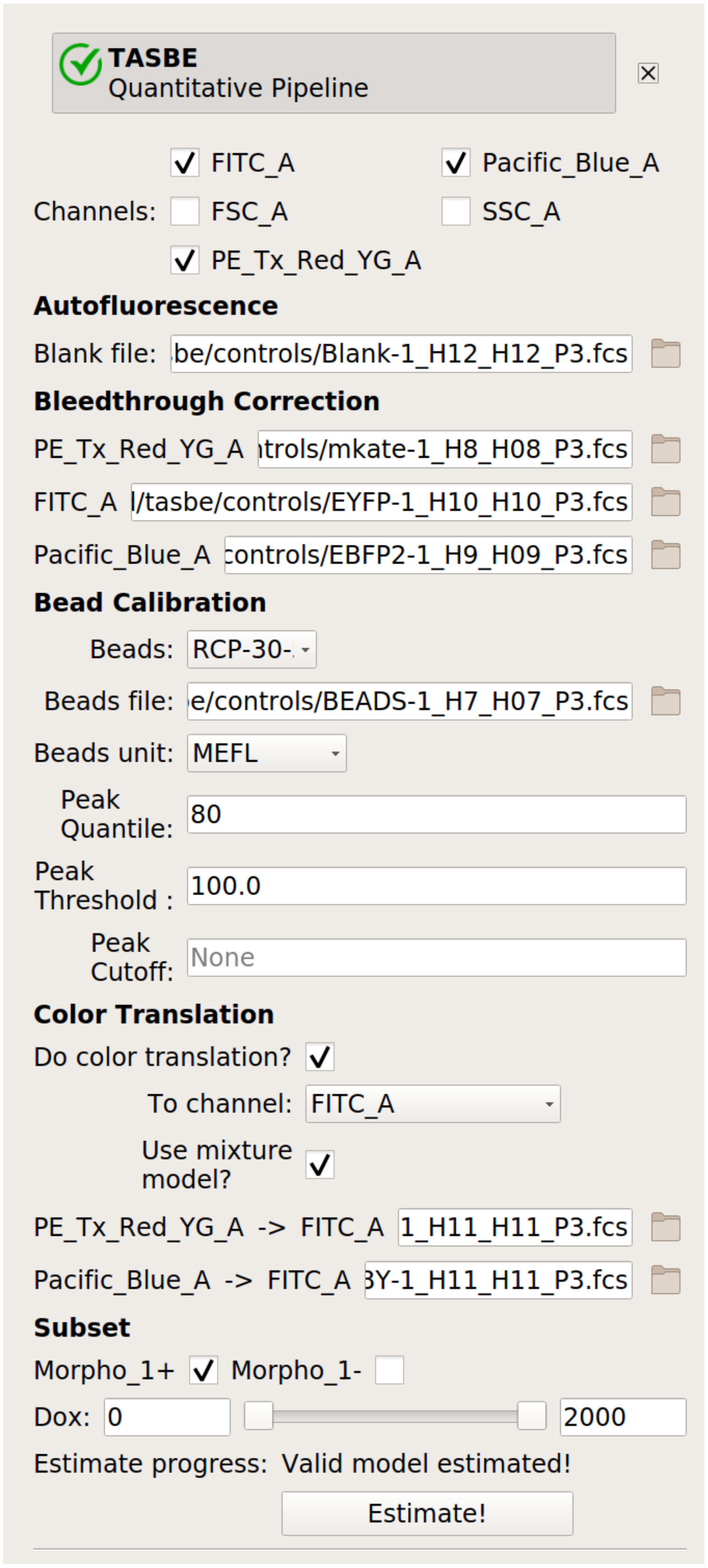


- This looks pretty good – a nice tight distribution. Let's use a 2D gaussian model to take 2 standard deviations around the centroid.



TASBE Calibration

- Use the TASBE calibration module to calibrate the `FITC_A`, `Pacific_Blue_A`, and `PE_Tx_Red_YG_A` channels. Settings:
  - Autofluorescence / Blank file: `controls/Blank-1_H12_H12_P3.fcs`
  - Bleedthrough / PE_Tx_Red_YG_A file: `controls/mkate-1_H8_H08_P3.fcs`
  - Bleedthrough / FITC_A file: `controls/EYFP-1_H10_H10_P3.fcs`
  - Bleedthrough / Pacific_Blue_A file: `controls/EBFP2-1_H9_H09_P3.fcs`
  - Beads: `Spherotech RCP-30-5A Lot AA01-AA04, AB01, AB02, AC01, GAA01-R`
  - Beads file: `controls/BEADS-1_H7_H07_P3.fcs`
  - Beads unit: `MEFL`
  - Remaining beads parameters: left at default
  - Color translation / Do color translation? : `True`
  - Color translation / To channel: `FITC_A`
  - Color translation / Use mixture model? `True`
  - Color translation / PE_Tx_Red_YG_A -> FITC_A: `controls/RBY-1_H11_H11_P3.fcs`
  - Color translation / Pacific_Blue_A -> FITC_A: `controls/RBY-1_H11_H11_P3.fcs`
  - Subset / Morpho_1+: `True`



TASBE
Quantitative Pipeline

Channels: ✓ FITC_A ✓ Pacific_Blue_A ☐ FSC_A ☐ SSC_A ✓ PE_Tx_Red_YG_A

Autofluorescence

Blank file: be/controls/Blank-1_H12_H12_P3.fcs

Bleedthrough Correction

PE_Tx_Red_YG_A ıtrols/mkate-1_H8_H08_P3.fcs

FITC_A l/tasbe/controls/EYFP-1_H10_H10_P3.fcs

Pacific_Blue_A controls/EBFP2-1_H9_H09_P3.fcs

Bead Calibration

Beads: RCP-30-

Beads file: e/controls/BEADS-1_H7_H07_P3.fcs

Beads unit: MEFL

Peak Quantile: 80

Peak Threshold : 100.0

Peak Cutoff: None

Color Translation

Do color translation? ✓

To channel: FITC_A

Use mixture model? ✓

PE_Tx_Red_YG_A -> FITC_A 1_H11_H11_P3.fcs

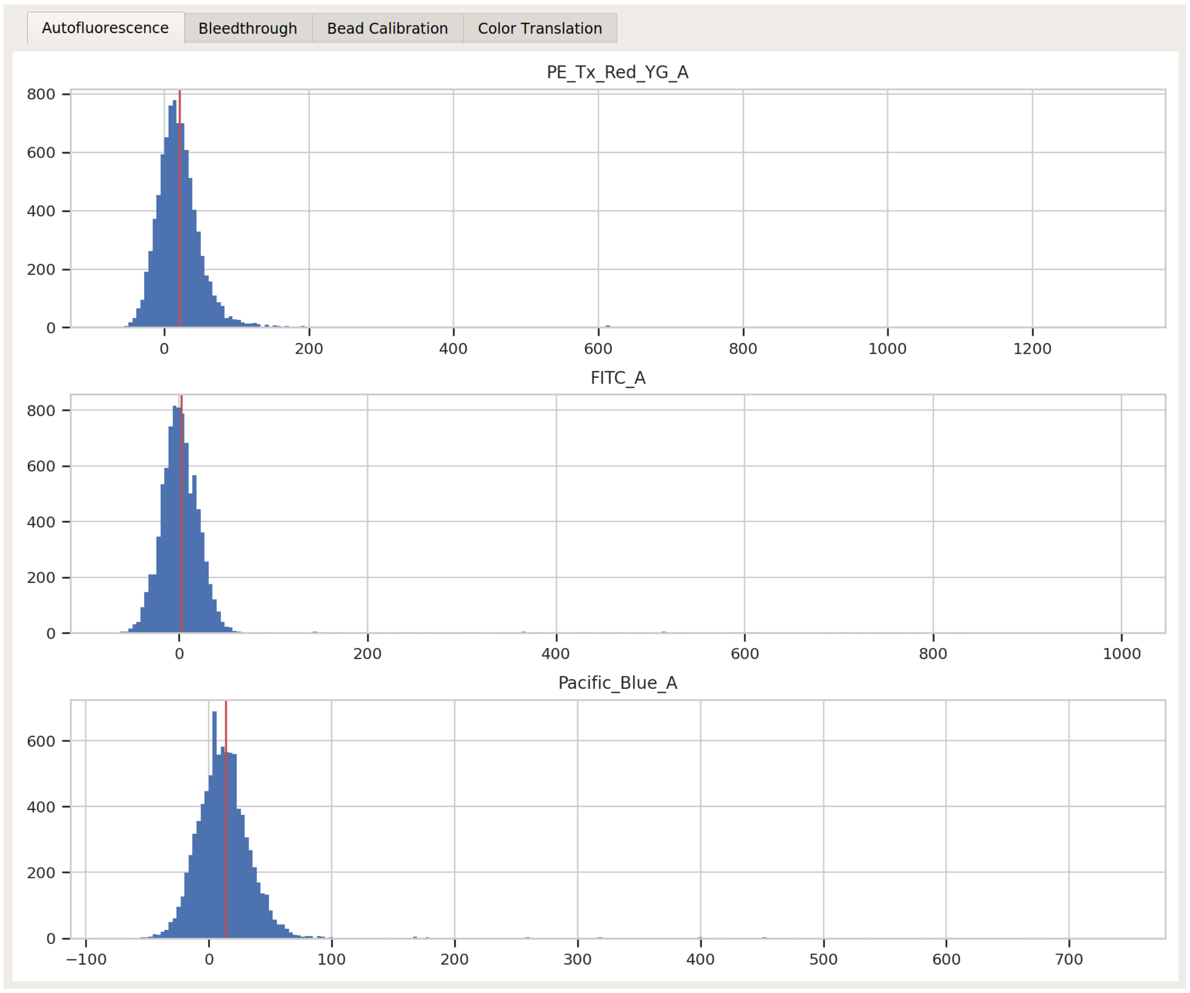
Pacific_Blue_A -> FITC_A 3Y-1_H11_H11_P3.fcs

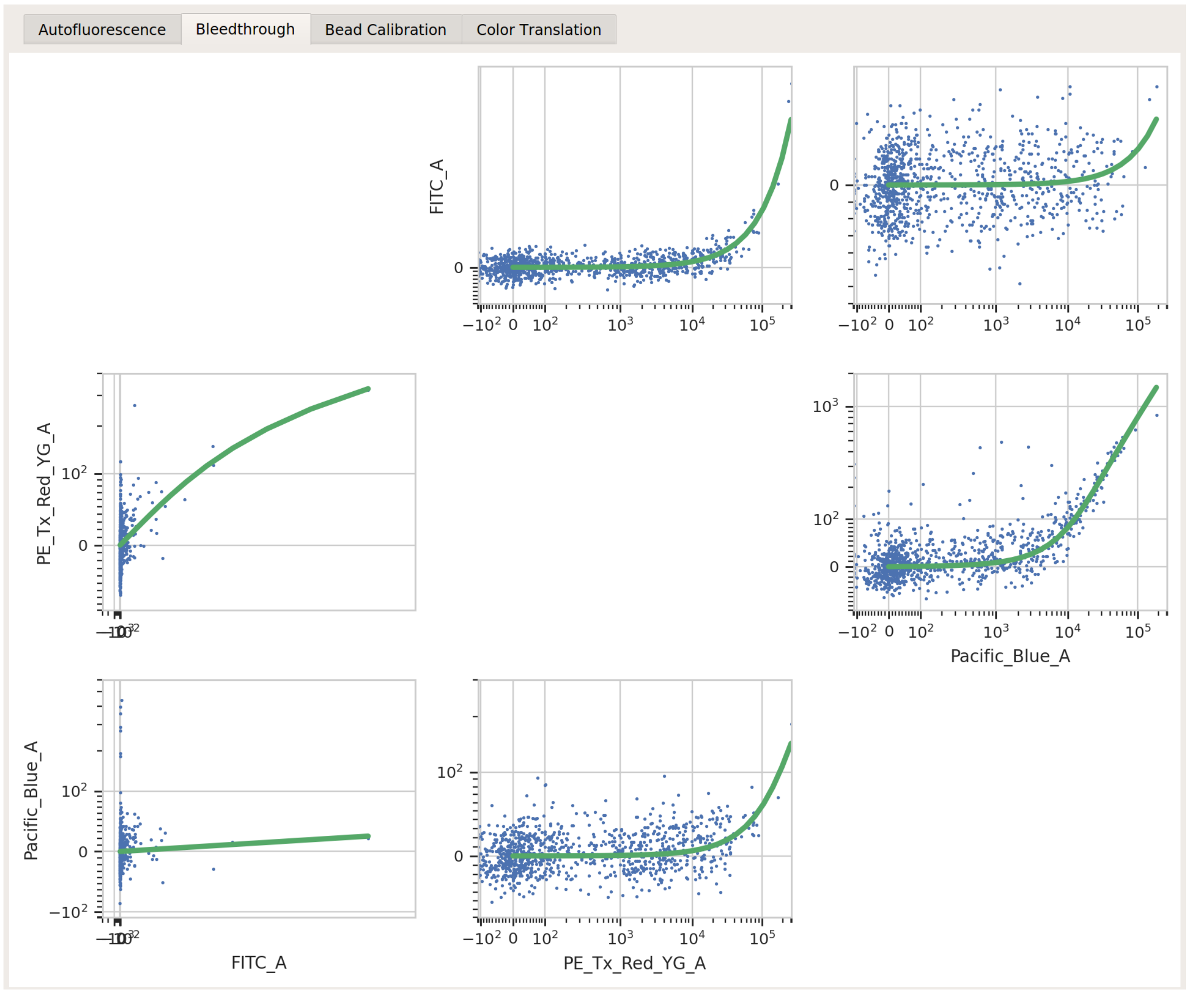
Subset

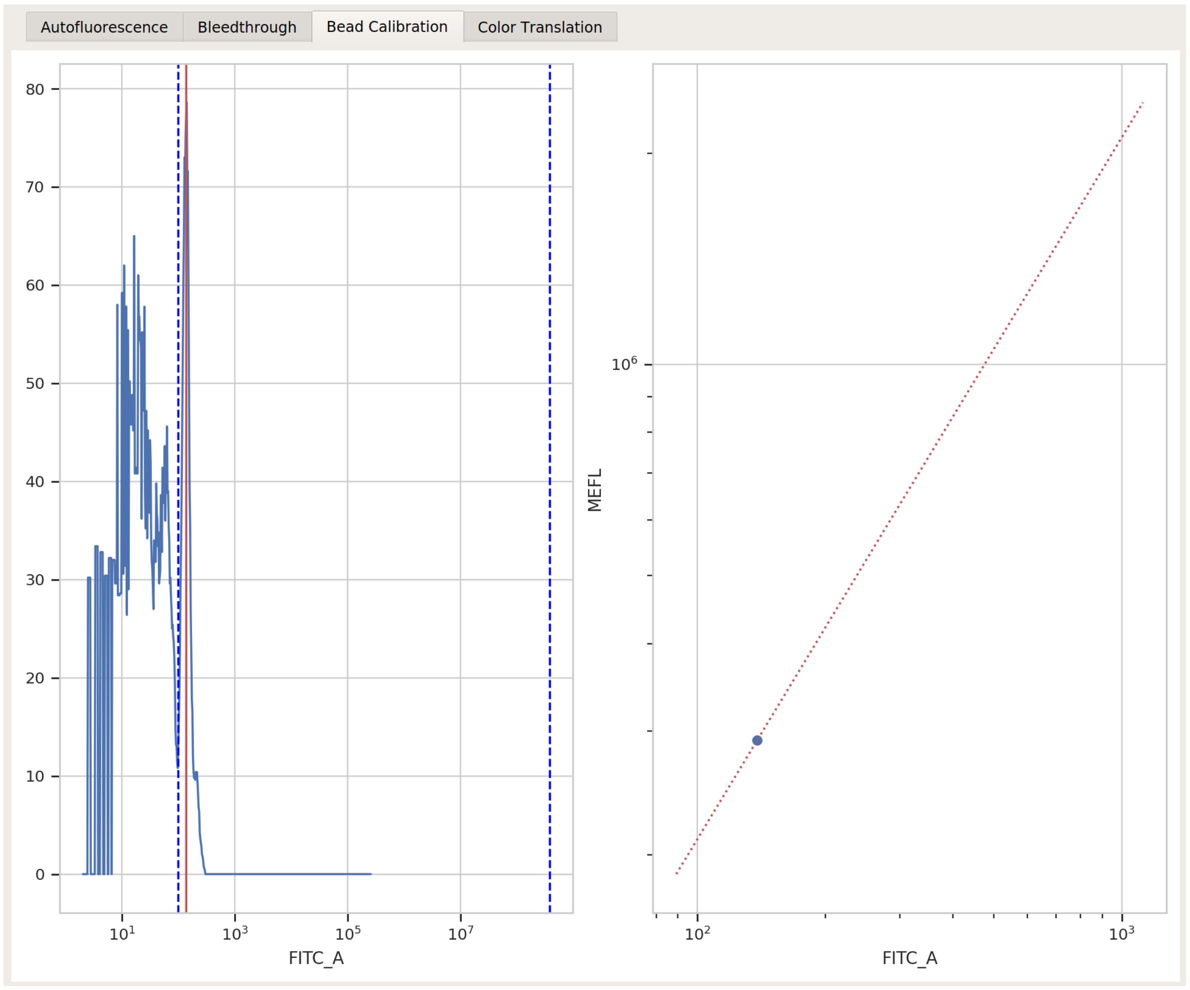
Morpho_1+ ✓ Morpho_1- ☐

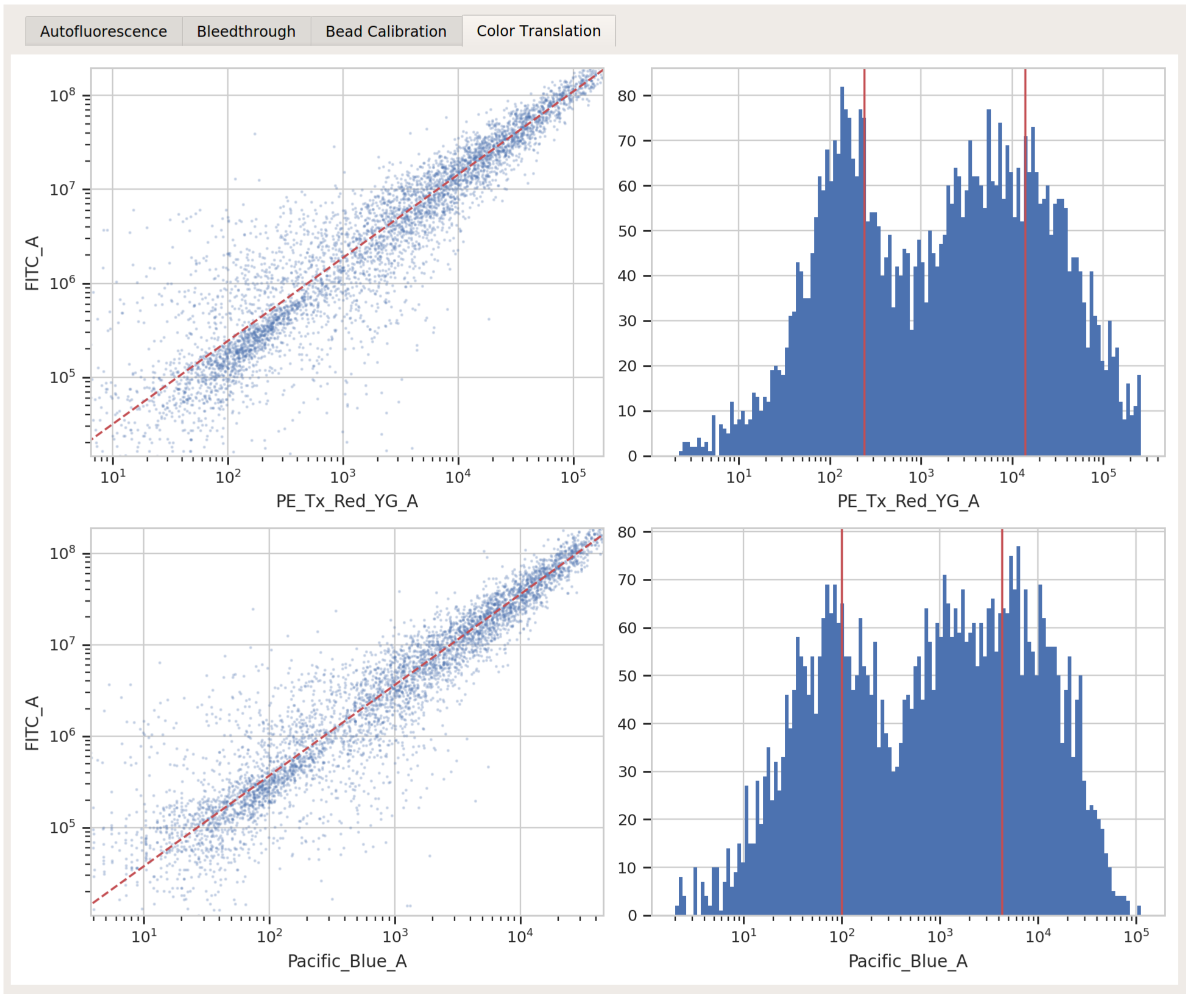
Dox: 0 2000

Estimate progress: Valid model estimated!


Autofluorescence
Bleedthrough
Bead Calibration
Color Translation
PE_Tx_Red_YG_A
FITC_A
Pacific_Blue_A


Autofluorescence
Bleedthrough
Bead Calibration
Color Translation
FITC_A
PE_Tx_Red_YG_A
Pacific_Blue_A
FITC_A
PE_Tx_Red_YG_A
Pacific_Blue_A

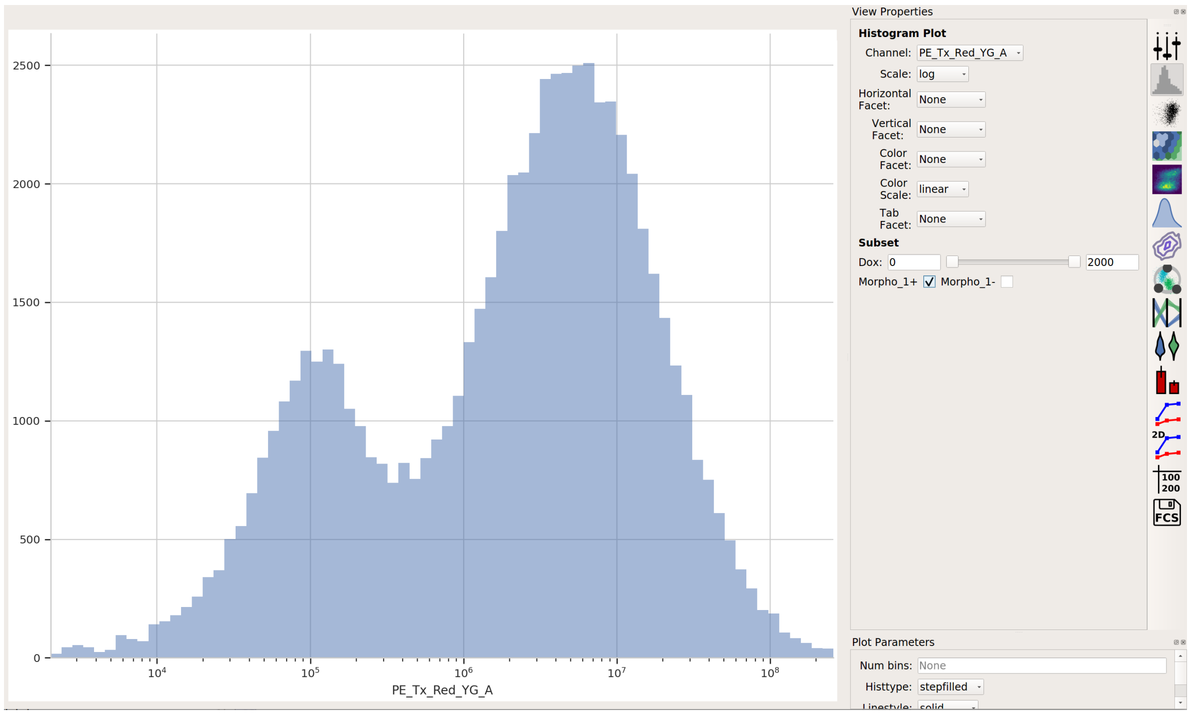

Autofluorescence
Bleedthrough
Bead Calibration
Color Translation
80
70
60
50
40
30
20
10
0
10^1
10^3
10^5
10^7
FITC_A
10^6
MEFL
10^2
10^3
FITC_A



Binned Analysis

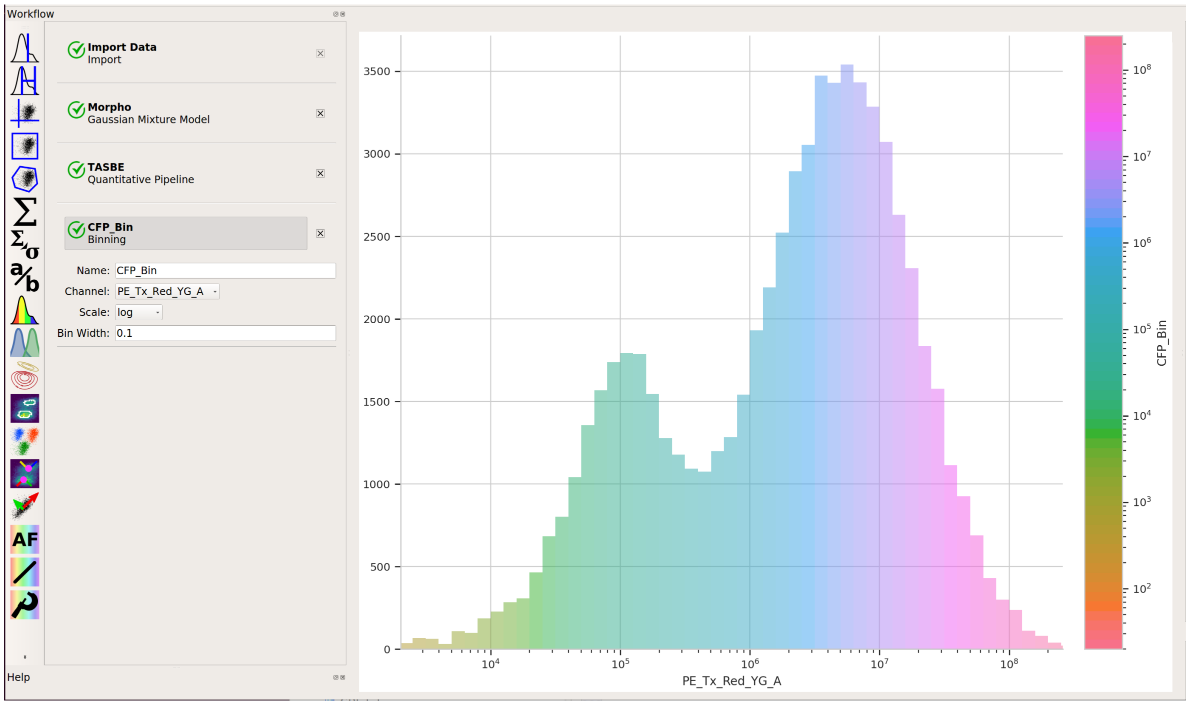
As described above, the example data in this notebook is from a transient transfection of mammalian cells in tissue culture. What this means is that there's a really broad distribution of fluorescence, corresponding to a broad distribution of transfection levels.

- Let's check the `PE_Tx_Red_YG_A` channel, where our mKate transfection marker is:

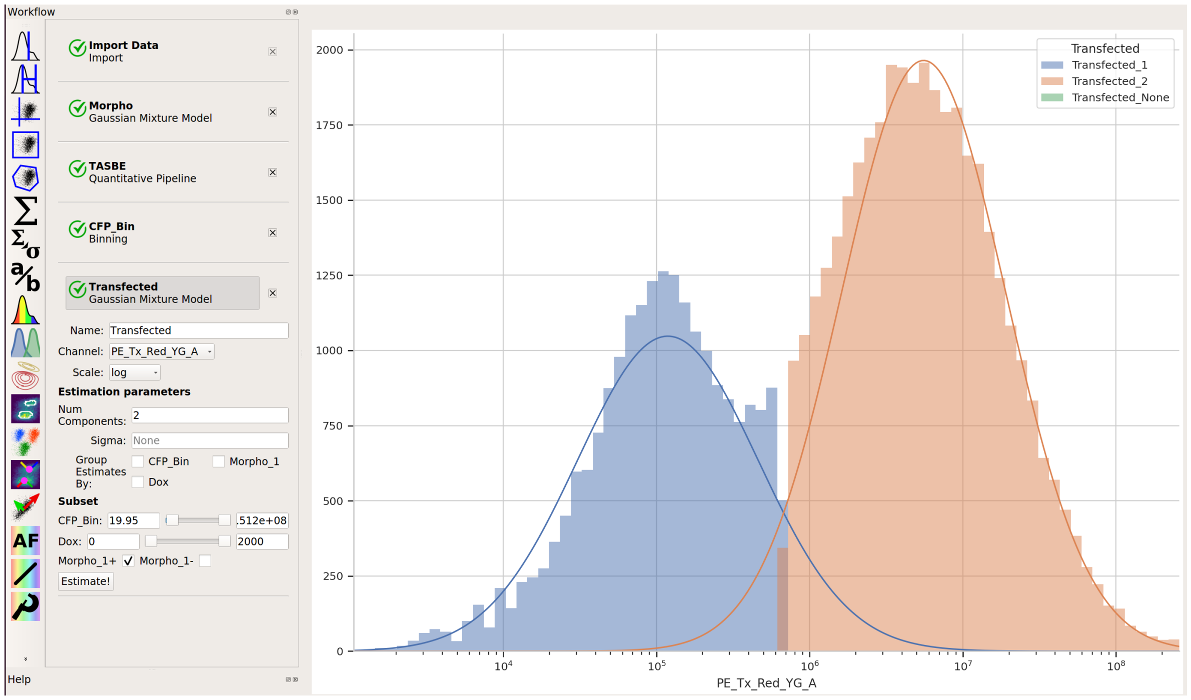


The way we handle this data is by dividing the cells into bins depending on their transfection levels. We find that cells that recieved few plasmids frequently behave differently (quantitatively speaking) than cells that received many plasmids. The **Binning** module applies evenly spaced bins; in this example, we're going to apply them on a log scale, every 0.1 log-units.

- Apply the **Binning** module with a log scale and a bin width of 0.1 log units:



- Also, we really only want to look at transfected cells – so let's use a **1D Gaussian** gate to separate the transfected population from the untransfected population:



Workflow

Import Data
Import

Morpho
Gaussian Mixture Model

TASBE
Quantitative Pipeline

CFP_Bin
Binning

Transfected
Gaussian Mixture Model

Name: Transfected
Channel: PE_Tx_Red_YG_A
Scale: log

Estimation parameters
Num Components: 2
Sigma: None
Group Estimates By: ☐ CFP_Bin ☐ Morpho_1 ☐ Dox

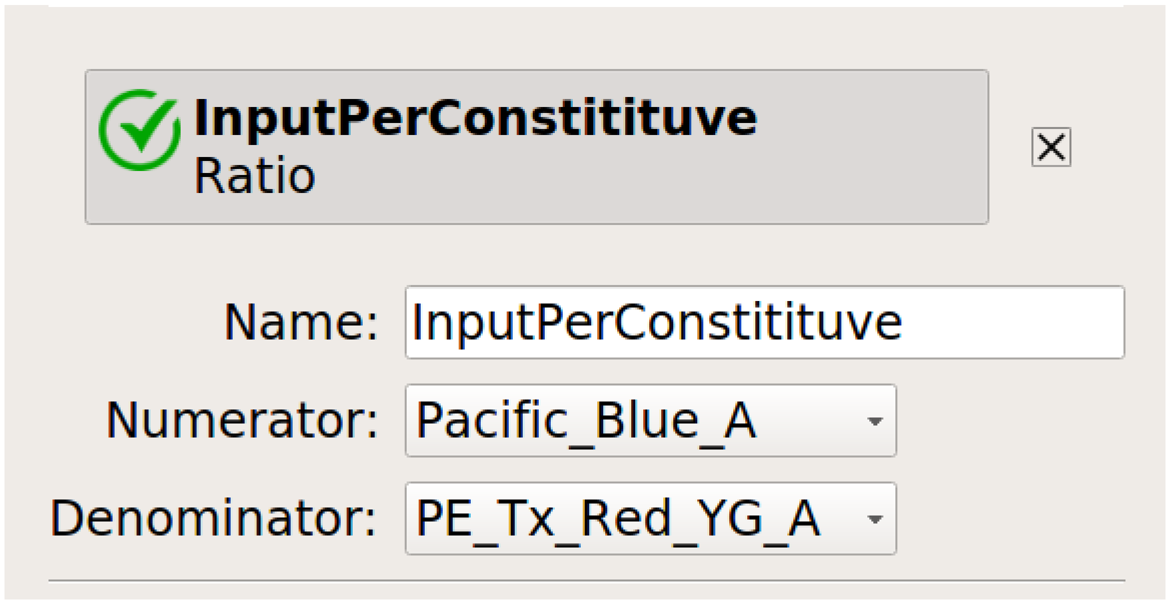
Subset
CFP_Bin: 19.95 .512e+08
Dox: 0 2000
Morpho_1+ ✓ Morpho_1- ☐

Estimate!

Help

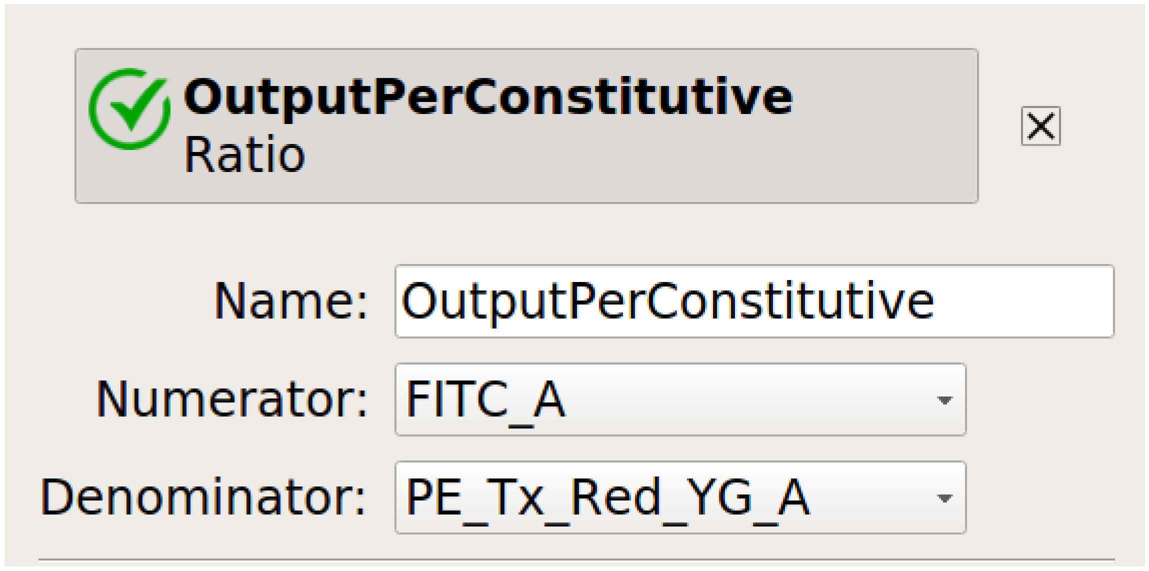
Creating Transfer Curves

- Let's normalize our input and output fluorescent proteins by our constitutive protein expression using **Ratio** operation:



InputPerConstitituve
Ratio

Name: InputPerConstitituve
Numerator: Pacific_Blue_A
Denominator: PE_Tx_Red_YG_A

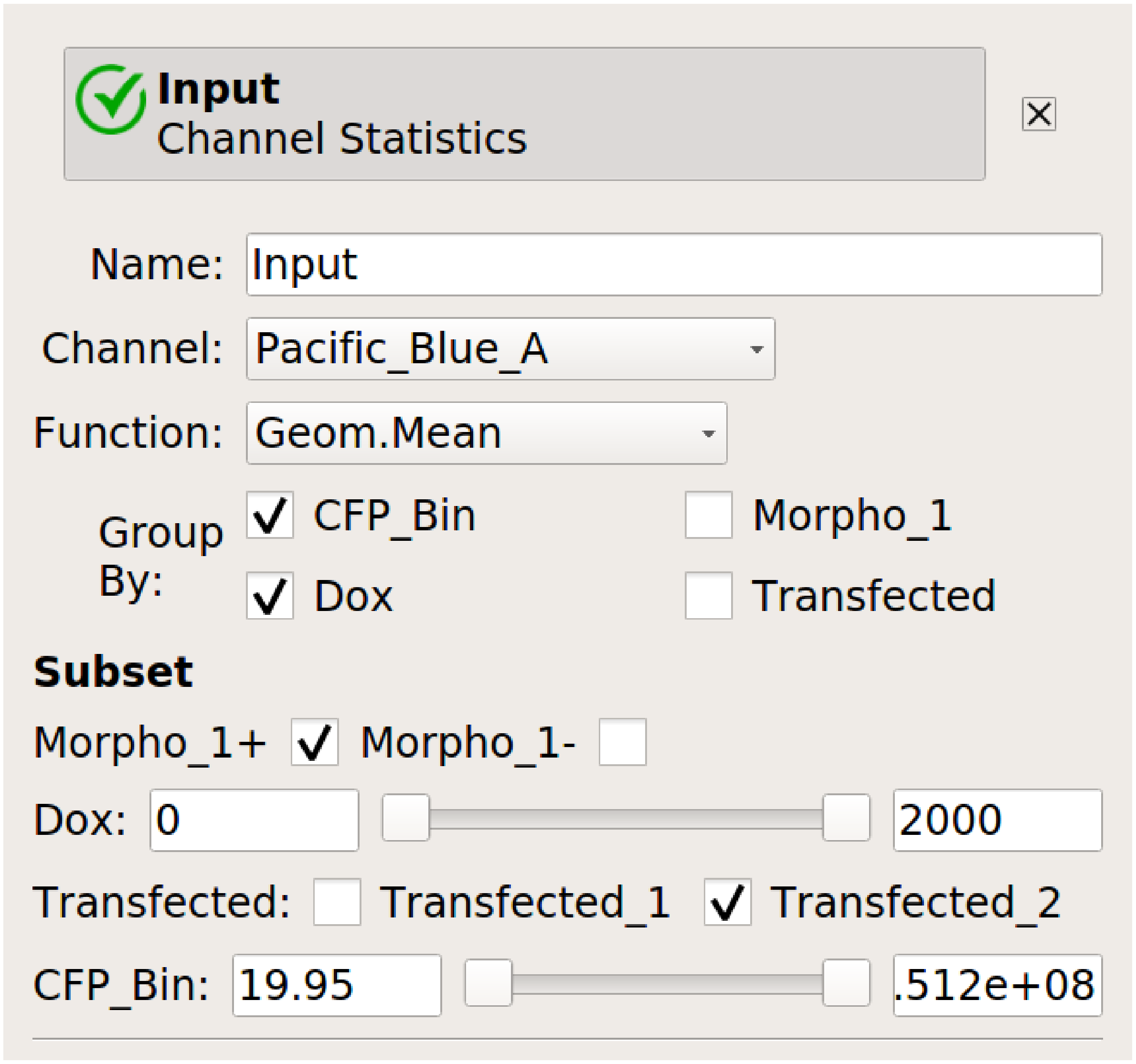

OutputPerConstitutive
Ratio

Name: OutputPerConstitutive

Numerator: FITC_A

Denominator: PE_Tx_Red_YG_A

- Now, create four statistics: the geometric mean of the input and output fluorescence measurements ,both normalized and not, for each bin, in each `Dox` condition. And make sure to only look at transfected single cells!



Input
Channel Statistics

Name: Input

Channel: Pacific_Blue_A

Function: Geom.Mean

Group By: [✓ CFP_Bin] [✓ Dox] Morpho_1 Transfected

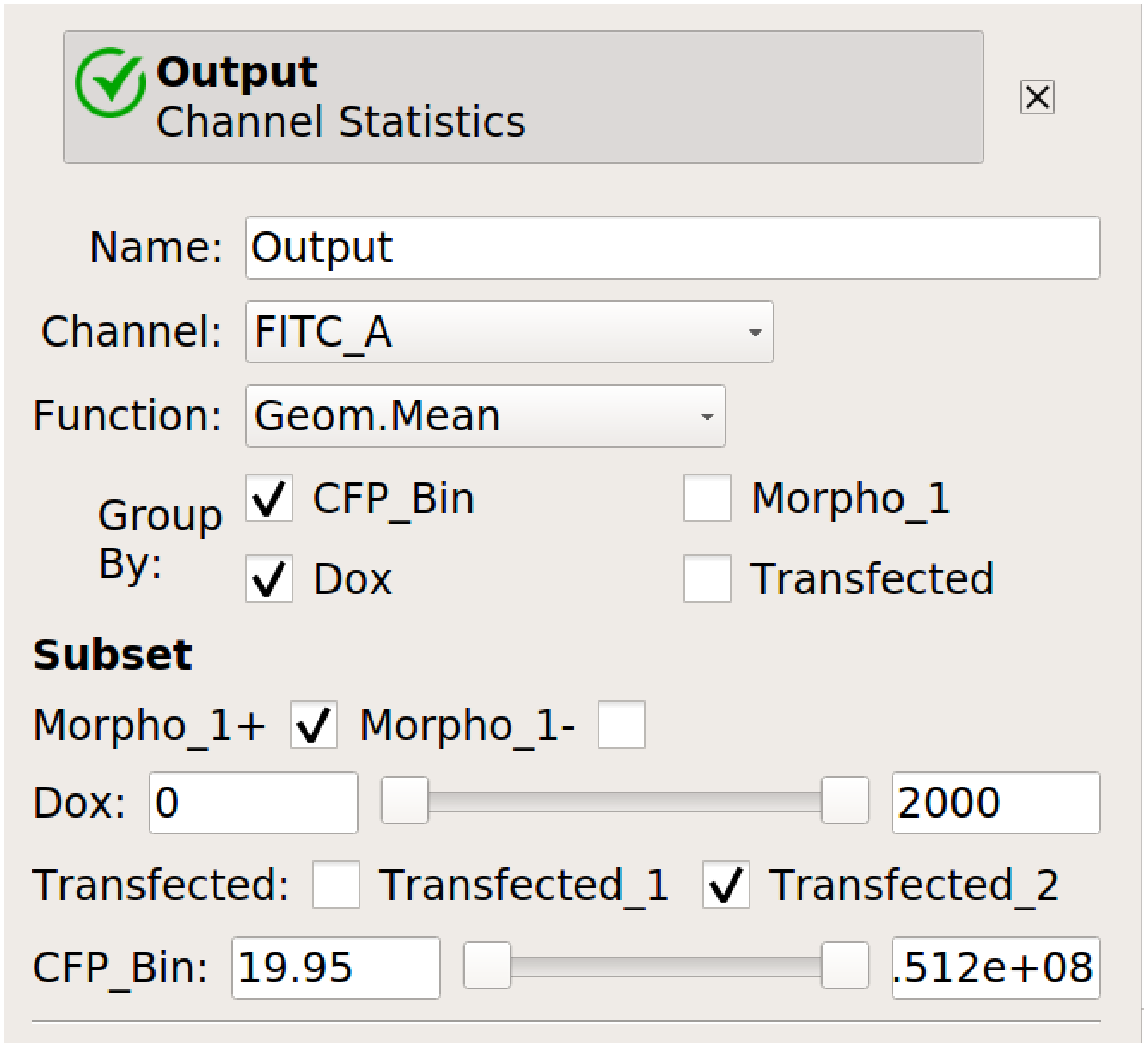
Subset

Morpho_1+ ✓ Morpho_1-

Dox: 0 2000

Transfected: Transfected_1 [✓ Transfected_2]

CFP_Bin: 19.95 .512e+08

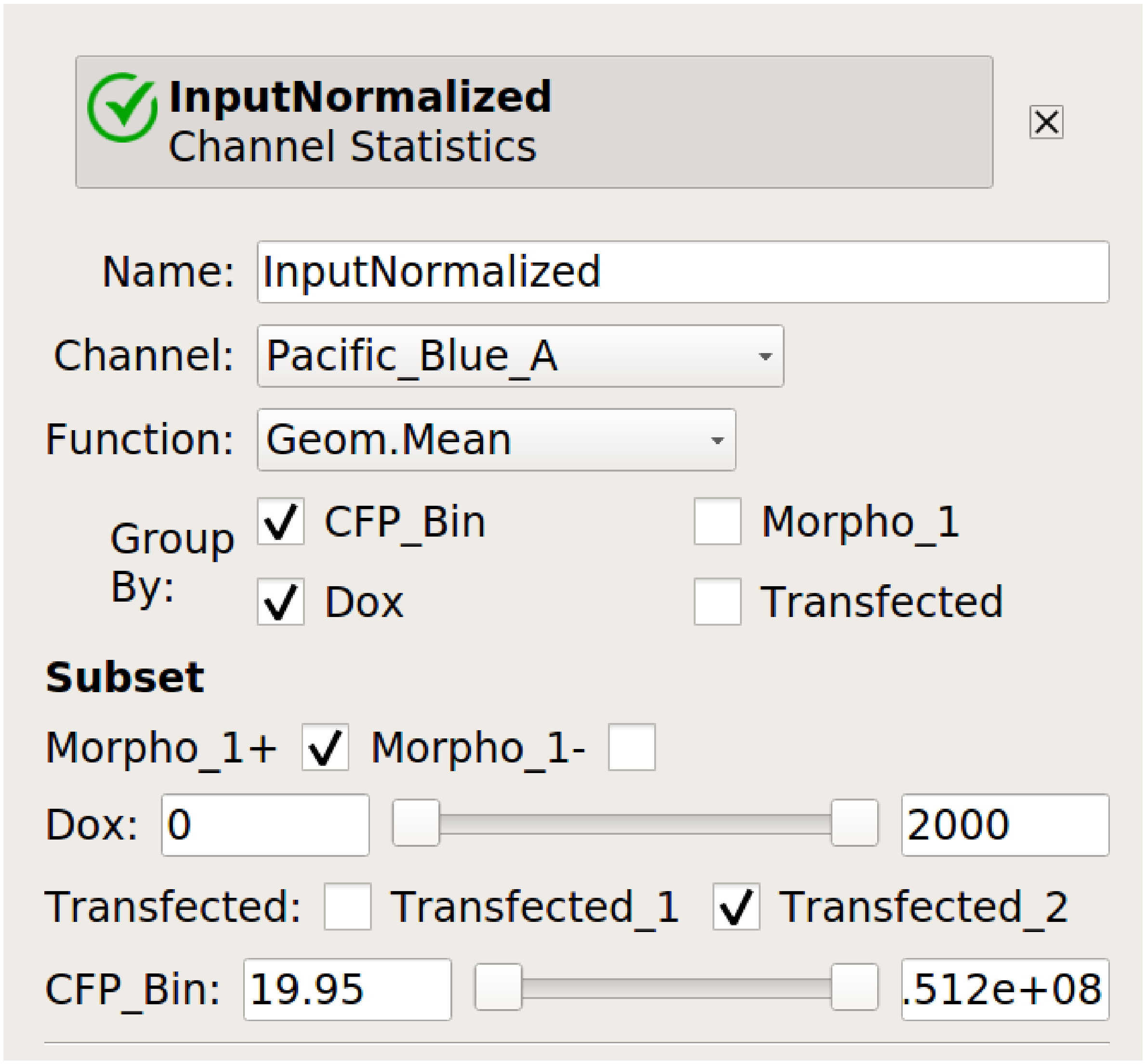


Output
Channel Statistics

Name: Output
Channel: FITC_A
Function: Geom.Mean
Group By: [✓ CFP_Bin] Morpho_1 [✓ Dox] Transfected

Subset

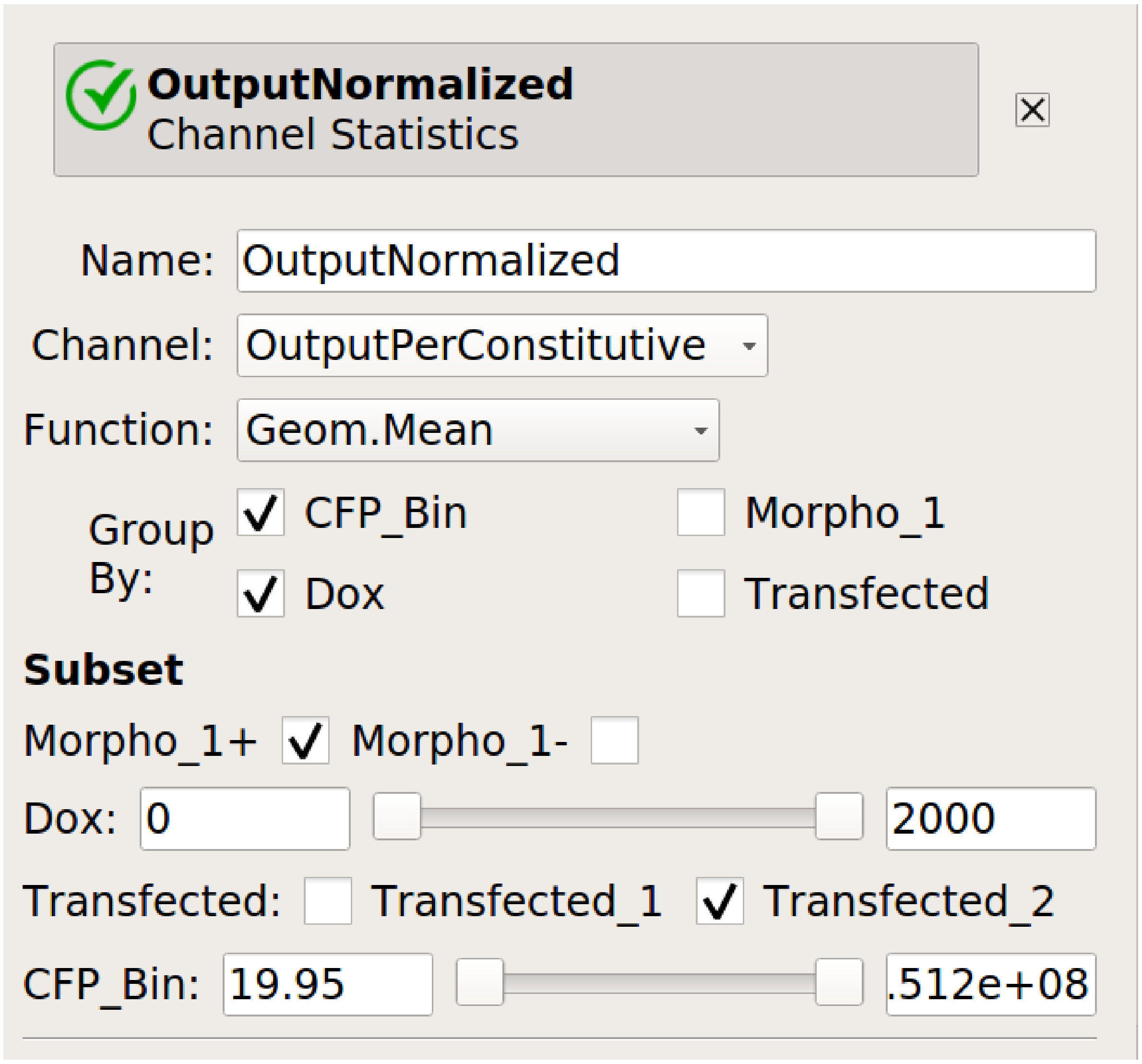
Morpho_1+ ✓ Morpho_1-
Dox: 0 2000
Transfected: Transfected_1 [✓ Transfected_2]
CFP_Bin: 19.95 .512e+08


InputNormalized
Channel Statistics

Name: InputNormalized
Channel: Pacific_Blue_A
Function: Geom.Mean
Group By: [✓ CFP_Bin] Morpho_1 [✓ Dox] Transfected

Subset

Morpho_1+ ✓ Morpho_1-
Dox: 0 2000
Transfected: Transfected_1 [✓ Transfected_2]
CFP_Bin: 19.95 .512e+08



OutputNormalized
Channel Statistics

Name: OutputNormalized

Channel: OutputPerConstitutive

Function: Geom.Mean

Group By: [✓ CFP_Bin] Morpho_1 [✓ Dox] Transfected

Subset

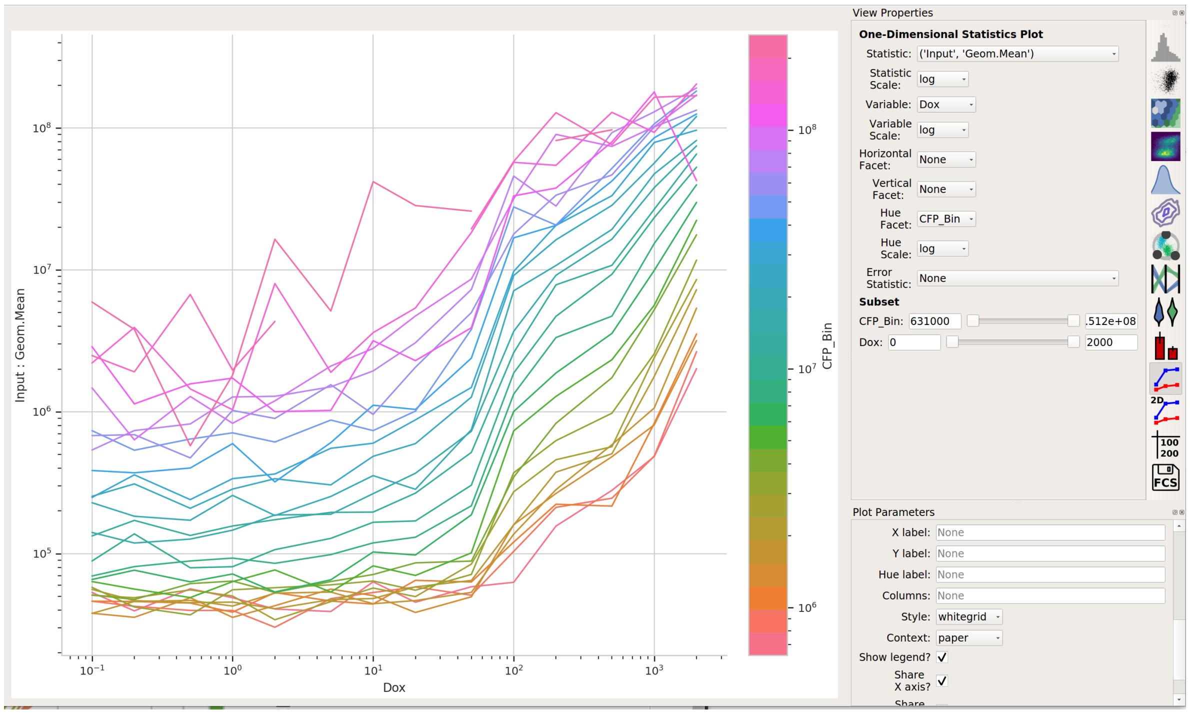
Morpho_1+ ✓ Morpho_1-

Dox: 0 2000

Transfected: Transfected_1 [✓ Transfected_2]

CFP_Bin: 19.95 .512e+08

- Now, we can start looking at transfer curves. First question: does the blue signal (our "input" signal) increase as we increase Dox concentration?



View Properties

One-Dimensional Statistics Plot

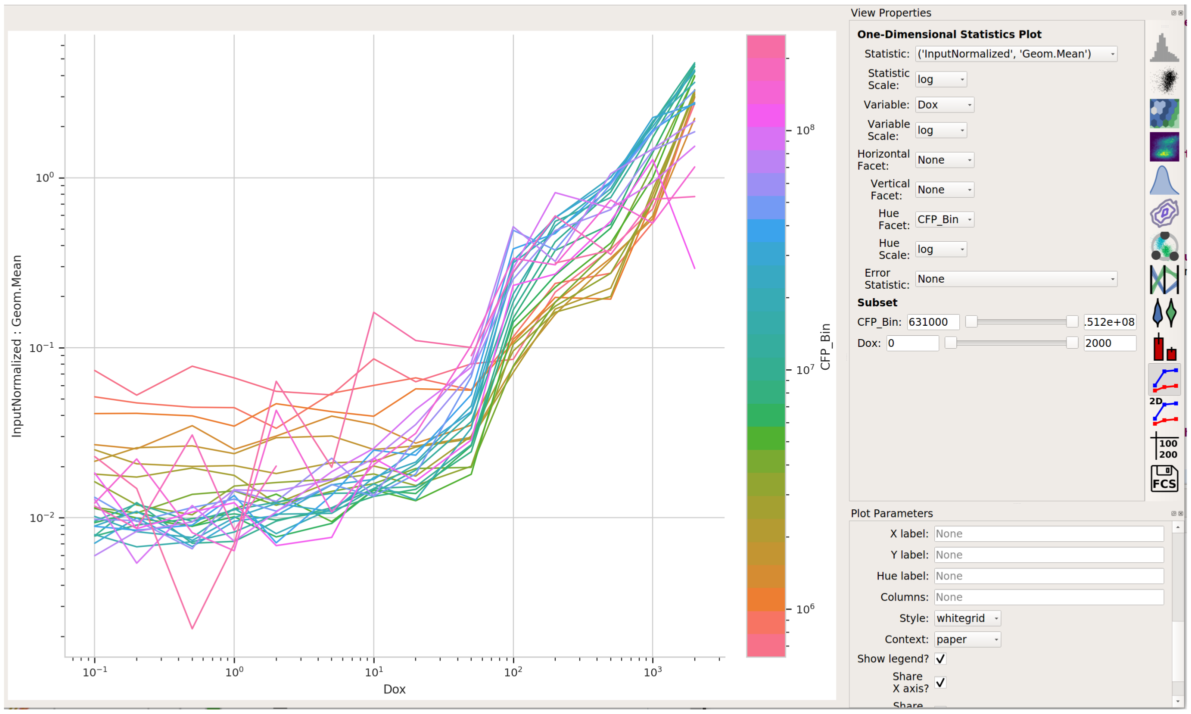
Statistic: ('Input', 'Geom.Mean')
Statistic Scale: log
Variable: Dox
Variable Scale: log
Horizontal Facet: None
Vertical Facet: None
Hue Facet: CFP_Bin
Hue Scale: log
Error Statistic: None

Subset

CFP_Bin: 631000 .512e+08
Dox: 0 2000

Plot Parameters

X label: None
Y label: None
Hue label: None
Columns: None
Style: whitegrid
Context: paper
Show legend? ✓
Share X axis? ✓
Share



View Properties

One-Dimensional Statistics Plot

Statistic: ('InputNormalized', 'Geom.Mean')
Statistic Scale: log
Variable: Dox
Variable Scale: log
Horizontal Facet: None
Vertical Facet: None
Hue Facet: CFP_Bin
Hue Scale: log
Error Statistic: None

Subset

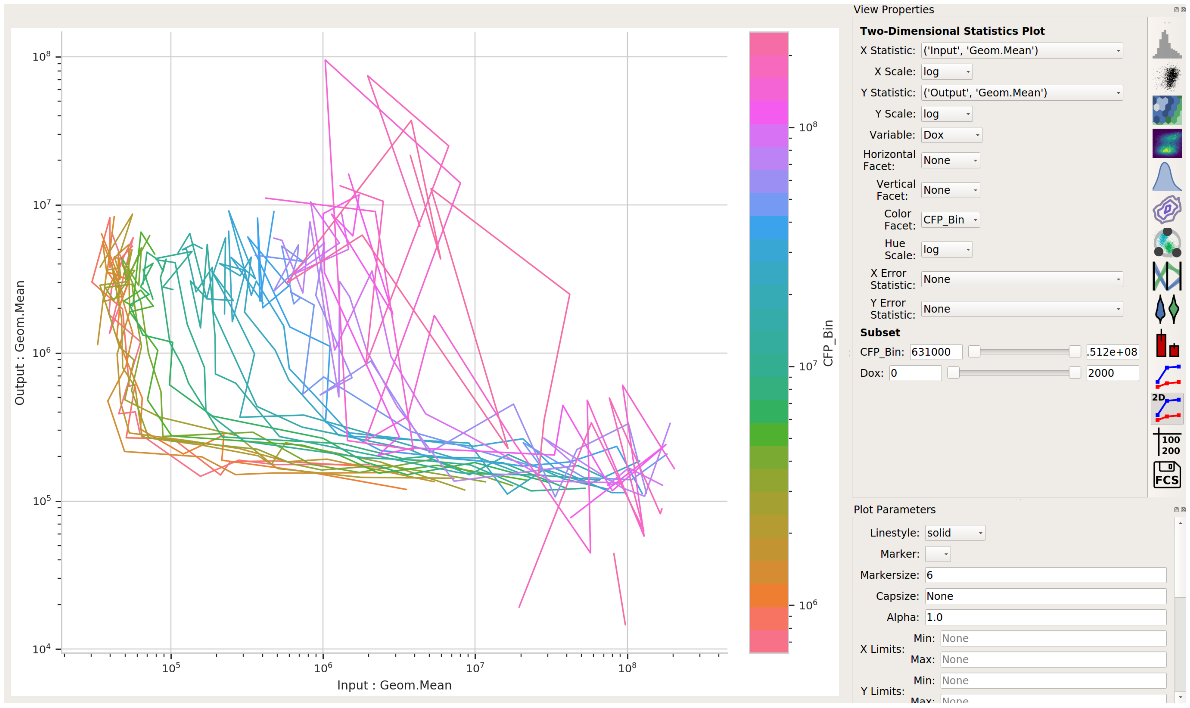
CFP_Bin: 631000 .512e+08
Dox: 0 2000

Plot Parameters

X label: None
Y label: None
Hue label: None
Columns: None
Style: whitegrid
Context: paper
Show legend? ✓
Share X axis? ✓
Share

Yes! That's a good sign that our experiment is working.

- Does the yellow signal (our "output") decrease as the blue signal (our "input") increases? Ie, is the repressor "inverting" the signal?



View Properties

Two-Dimensional Statistics Plot

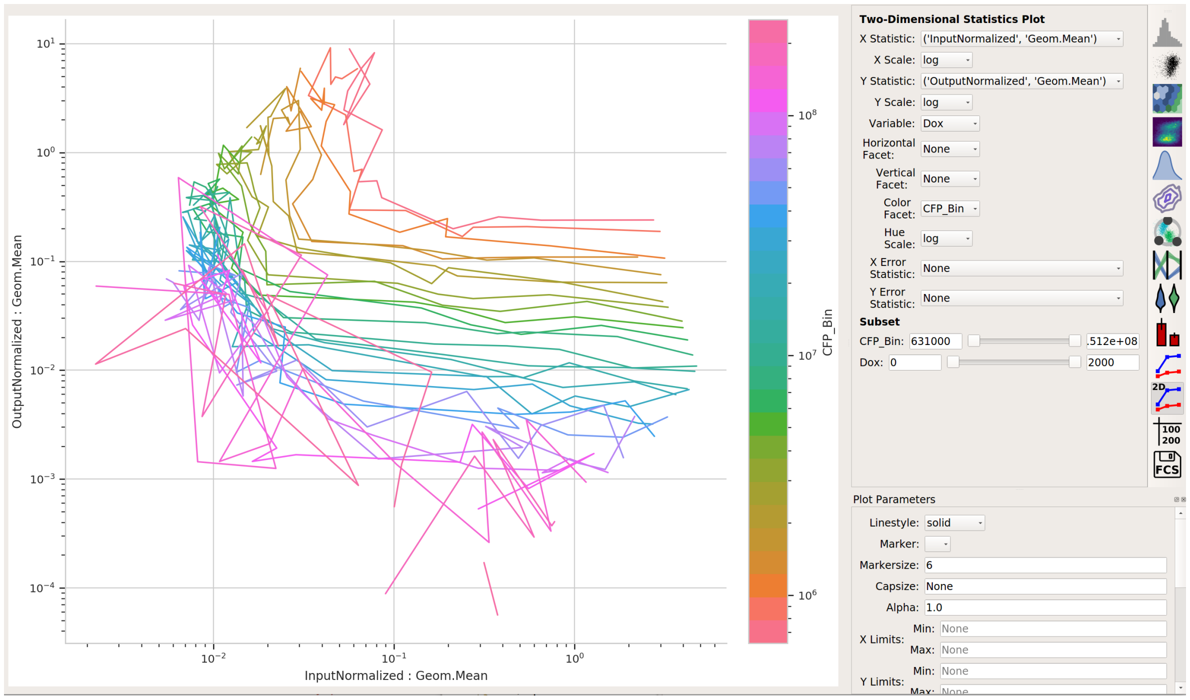
X Statistic: ('Input', 'Geom.Mean')
X Scale: log
Y Statistic: ('Output', 'Geom.Mean')
Y Scale: log
Variable: Dox
Horizontal Facet: None
Vertical Facet: None
Color Facet: CFP_Bin
Hue Scale: log
X Error Statistic: None
Y Error Statistic: None

Subset

CFP_Bin: 631000 .512e+08
Dox: 0 2000

Plot Parameters

Linestyle: solid
Marker:
Markersize: 6
Capsize: None
Alpha: 1.0
X Limits: Min: None Max: None
Y Limits: Min: None Max: None



Two-Dimensional Statistics Plot

X Statistic: ('InputNormalized', 'Geom.Mean')
X Scale: log
Y Statistic: ('OutputNormalized', 'Geom.Mean')
Y Scale: log
Variable: Dox
Horizontal Facet: None
Vertical Facet: None
Color Facet: CFP_Bin
Hue Scale: log
X Error Statistic: None
Y Error Statistic: None

Subset

CFP_Bin: 631000 .512e+08
Dox: 0 2000

Plot Parameters

Linestyle: solid
Marker:
Markersize: 6
Capsize: None
Alpha: 1.0
X Limits: Min: None Max: None
Y Limits: Min: None Max: None

The data is a little "messy" – primarily because of bins that didn't have many events in them, and thus gave quite noisy signals – but it's clear that we are seeing an inversion of the input signal to the output signal. The repressor works.

5.1.23 How-To Guides

These guides are "recipes" – step-by-step guides to accomplish a particular task. They are a little higher-level than the tutorials. If there is something that you find confusing, please feel free to submit a bug report (or even better, a patch or pull request.)

5.1.24 Users' guides

These documents help you understand some of the principles Cytoflow is based on and some subtler points about using it to analyze flow cytometry data. They can help you use Cytoflow more effectively!

5.2 Developer Manual

So you want to use *`cytoflow`* in your own Python-based analyses. Great! May I recommend you start with the *tutorials* and *examples* – they will give you a feel for the kinds of things you can use *`cytoflow`* to do. All of them are generated from Jupyter notebooks, and those notebooks and data can be found in the `Releases` tab at the project homepage (look for `cytoflow-$VERSION-examples-basic.zip` and `cytoflow-$VERSION-examples-advanced.zip`).

Then, if you decide you want to have a go, see the installation notes. Quick hint: if you have Anaconda installed, say:

```
conda config --add channels cytoflow
conda create --name cytoflow cytoflow
```

This creates a new Anaconda environment named `cytoflow` and installs the latest *`cytoflow`* package from the Anaconda Cloud.

For more details of these modules, you're likely to want to see the module documents

5.2.1 Contents:

Tutorials

These tutorials illustrate some of the capabilities of the *`cytoflow`* library. They are generated from the example notebooks and data sets in the `cytoflow-$VERSION-examples-basic.zip` file, available from the `Releases` tab at the project homepage. Feel free to fire up a Jupyter notebook and follow along!

Basic example cytometry workflow

Welcome to `cytoflow`! I'm glad you're here. The following is a heavily commented workflow for importing a few tubes of cytometry data and doing some (very) basic analysis. The goal is to give you not only a taste of using the library for interactive work, but also some insight into the rationale for the way it's designed the way it is and the way it differs from existing projects.

`cytoflow`'s goal is to enable *quantitative, reproducible* cytometry analysis. Reproducibility between cytometry experiments is poor, due in part to differences in analysis between operators; but if all your analysis is in a `Jupyter` notebook (like this one!), then sharing and reuse of workflows is much easier.

Let's look at a very basic experiment, containing only two tubes. These two tubes contain cells that are expressing fluorescent proteins as the read-out of some cell state. We'll assume that these two tubes are identical, except that one has been induced with 1 µM of the small molecule inducer Doxycycline (aka 'Dox') and the other tube has been induced with 10 µM Dox.

Start by setting up Jupyter's plotting interface, then import the `cytoflow` module.

```
# in some buggy versions of the Jupyter notebook, this needs to be in its OWN CELL.
%matplotlib inline
```

```
import cytoflow as flow

# if your figures are too big or too small, you can scale them by changing matplotlib's␣
↪DPI
import matplotlib
matplotlib.rc('figure', dpi = 160)
```

The central data structure in `cytoflow` is the `Experiment`, which is basically a `pandas.DataFrame` containing the events; its associated metadata (experimental conditions and the like); and some methods to manipulate them.

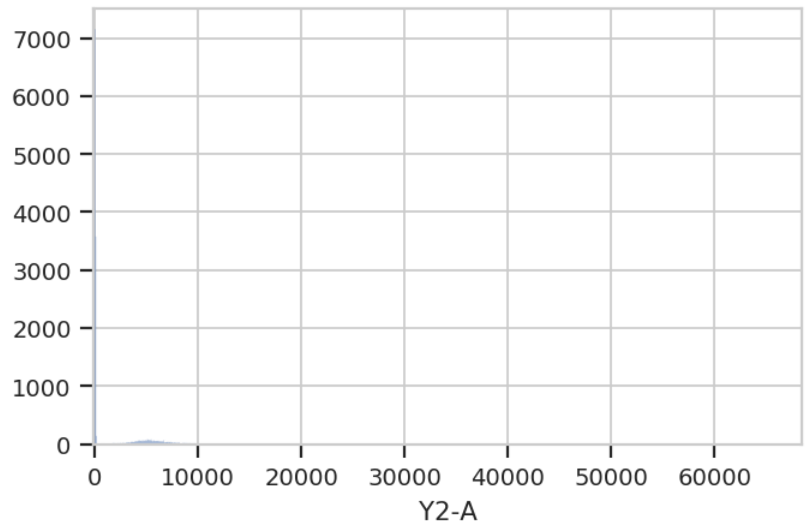
You usually create an `Experiment` using an instance of the `ImportOp` class. Start by defining two tubes, including their experimental conditions (ie, how much Dox is in each); then give those tubes, and a `dict` specifying the experimental conditions' names and types, to `ImportOp`. Call the `apply()` function to get back the `Experiment` with all the data in it.

```
tube1 = flow.Tube(file = 'data/RFP_Well_A3.fcs',
                  conditions = {'Dox' : 10.0})
tube2 = flow.Tube(file='data/CFP_Well_A4.fcs',
                  conditions = {'Dox' : 1.0})

import_op = flow.ImportOp(conditions = {'Dox' : 'float'},
                          tubes = [tube1, tube2])

ex = import_op.apply()
```

Once you have an `Experiment` instance, this is the last time you should ever think about tubes or wells. Rather, think of your experiment as a very large set of single-cell measurements, each of which has some metadata associated with it; in this case, how much Dox the cell was exposed to. `cytoflow` helps you focus your analysis on how those measurements change as the experimental conditions change, without worrying about what cells were in what tube.

Let's have a quick look at one of the fluorescence channels, `Y2-A`. Instantiate a `HistogramView` and tell it which channel we're looking at, then call `plot` and pass it the experiment containing the data. Remember, this is not a single tube, but rather all the data in the `Experiment`.

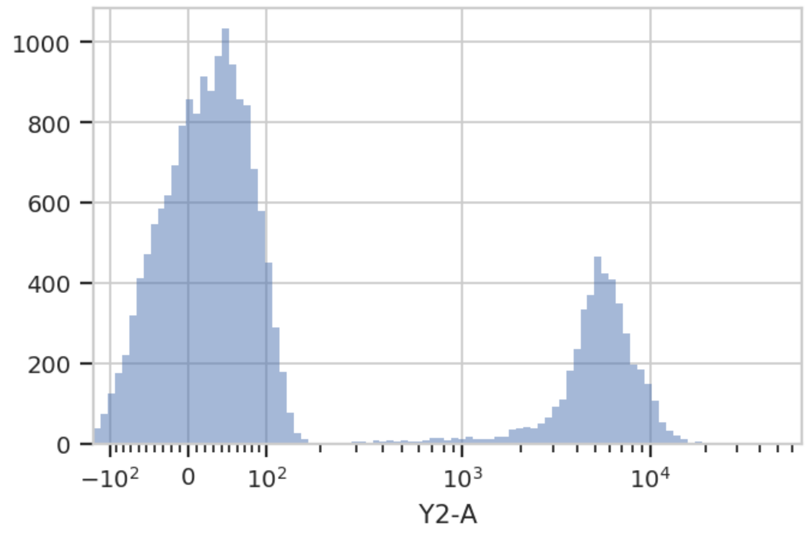
```
hist = flow.HistogramView()
hist.channel = 'Y2-A'
hist.plot(ex)
```



Hmmm. This plot is hard to interpret because most of the data is clustered around 0, in the linear range of the detector's response. Let's re-plot using a different scale. My favorite is `logicle`, which has a linear response around 0 and a log range elsewhere. We specify the plot scale by setting the `scale` attribute of `HistogramView` to `logicle`; other options are `log` and `linear`.

The cell below also demonstrates a different way to parameterize the `HistogramView` instance, by passing parameters to the constructor instead of setting the values of the instance's attributes after we make it. Either way is correct.

```
hist = flow.HistogramView(channel = 'Y2-A',
                          scale = 'logicle')
hist.plot(ex)
```



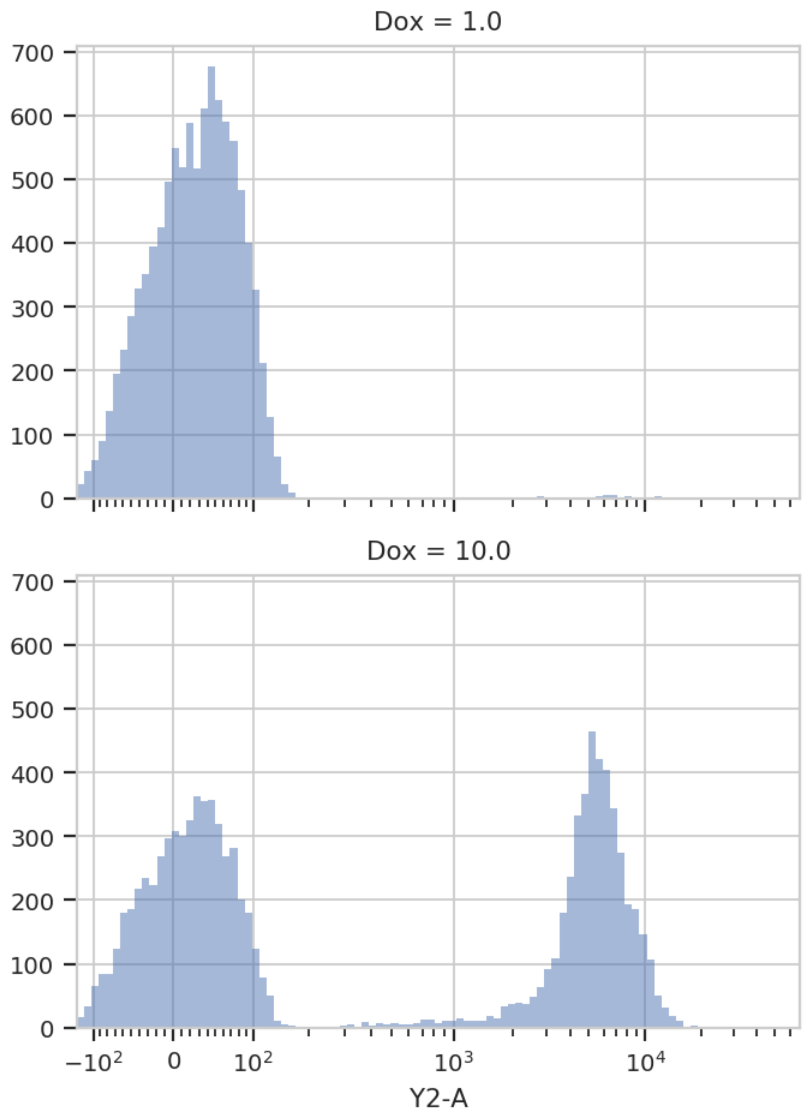
Ah, much better. There is clearly a population of cells around 0 and a population of cells around about 5000. But! This is the entire `Experiment` – is is one of the populations from the low-Dox tube and the other population from the high-Dox tube?

Let's see if the histogram is different for the two different concentrations of inducer. `CytoFlow`'s plotting takes inspiration from Trellis plots (eg the lattice package in R): to split the data into subsets, you tell the plotting module which metadata "facet" you want to plot in the X, Y and color (hue) axes.

This time, we tell `HistogramView` to make a separate plot for each different value of Dox and stack them on top of eachother by saying `yfacet = 'Dox'`; if we wanted the plots side-by-side, we would have said `xfacet = 'Dox'`.

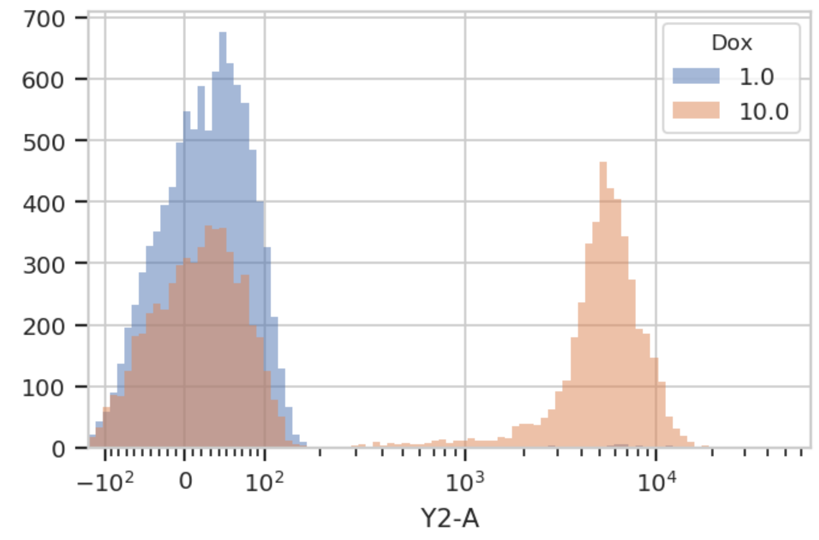
Also note that this time, we don't keep around the `HistogramView` object; if we don't need to re-use it, we can just call `plot` right after the constructor.

```
flow.HistogramView(channel = 'Y2-A',
                   scale = 'logicle',
                   yfacet = 'Dox').plot(ex)
```



Indeed, the two tubes have dramatically different histograms. We could also plot them on top of eachother with different colors, by using `huefacet` instead of `yfacet`.

```
flow.HistogramView(channel = "Y2-A",
                   scale = "logicle",
                   huefacet = 'Dox').plot(ex)
```



So, there's a clear difference between these tubes: one has a substantial population above about ~200 in the Y2-A, and the other doesn't. What's the proportion of "high" cells? Let's gate out the "high" population using a `ThresholdOp`.

```
thresh = flow.ThresholdOp(name = "T",
                          channel = "Y2-A",
                          threshold = 200)

ex2 = thresh.apply(ex)
```

Two important things to note here: first, an `Operation` (such as `ThresholdOp`) *does not operate in place*; instead, you create a `ThresholdOp` object, then call `apply()` to run it on an `Experiment`. The `apply()` function returns a new `Experiment`.

Second, a gate *does not remove events*; it simply *adds additional metadata* to the events already there. You can see this if we look at the underlying `pandas.DataFrame` for `ex` and `ex2`:

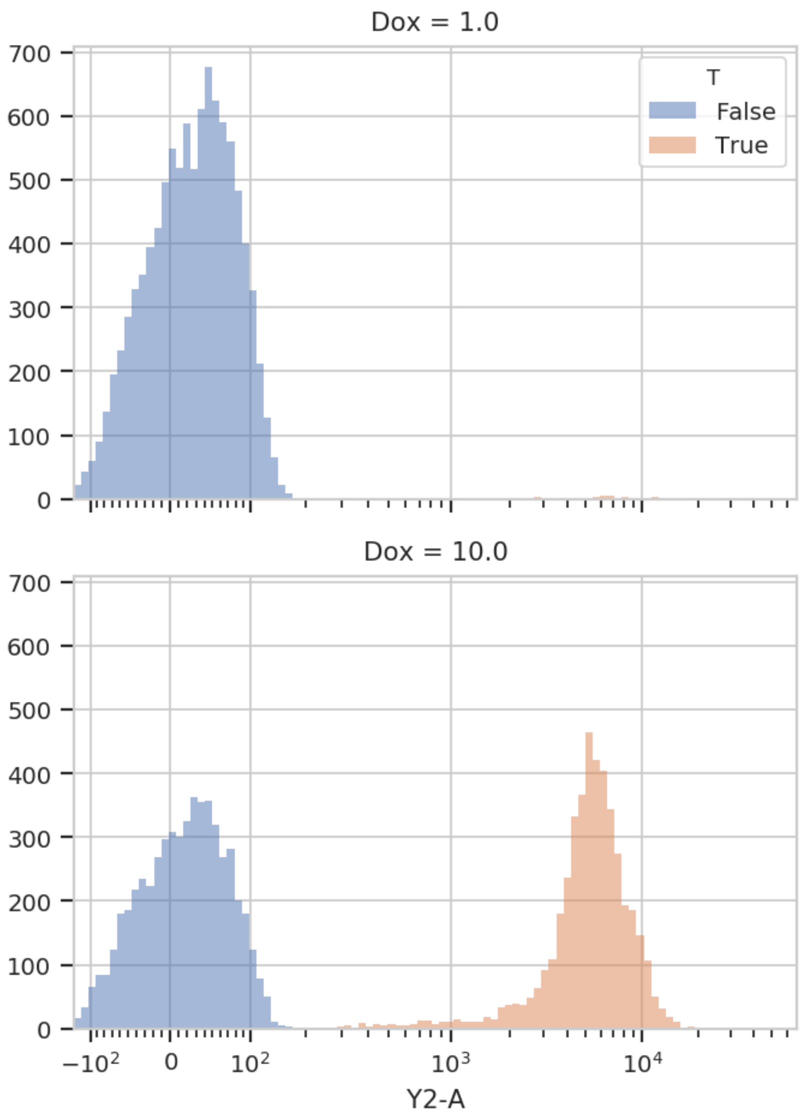
```
print(ex.data.head())
print("----")
print(ex2.data.head())
```

```
         B1-A        B1-H          B1-W   Dox        FSC-A       FSC-H  0 -127.094002 ␣
↪257.718353 -63040.300781  10.0  459.962982  437.354553
1  -70.234840  255.798340 -34034.042969  10.0 -267.174652  365.354553
2  -96.471756  313.398346 -41931.687500  10.0 -201.582336  501.354553
3   18.831570  277.669250   8514.489258  10.0  291.259888  447.029755
4  100.882095  255.756256  51291.074219  10.0 -397.168579  354.565308

           FSC-W      HDR-T         SSC-A         SSC-H          SSC-W  0  137847.578125  ␣
↪2.018511   840.091370   747.917847  147225.328125
1  -95849.679688  27.451754  3476.902344  3163.917969  144038.046875
2  -52700.828125  32.043865   480.270691   507.917877  123937.437500
3   85399.273438  79.327492  8026.275879  6741.838867  156043.484375
4 -146821.125000  79.731194  7453.750488   609.884277  262143.968750

         V2-A        V2-H           V2-W         Y2-A         Y2-H  0   41.593452  240.
↪153854   22701.017578   109.946274   153.630051
1  103.437637  336.153870   40332.058594  5554.108398  4273.629883
```

```
2 -271.375580  256.153870 -138860.828125    81.835281   121.630051
3  -26.212378  207.677841  -16543.453125   -54.120304    98.122017
4   44.559933  216.036865   27035.013672   -10.542595   127.326027

            Y2-W
0   93802.468750
1  170344.203125
2   88188.023438
3  -72294.242188
4  -10852.761719
----
         B1-A        B1-H          B1-W   Dox        FSC-A       FSC-H  0 -127.094002 ␣
↪257.718353 -63040.300781  10.0  459.962982  437.354553
1  -70.234840  255.798340 -34034.042969  10.0 -267.174652  365.354553
2  -96.471756  313.398346 -41931.687500  10.0 -201.582336  501.354553
3   18.831570  277.669250   8514.489258  10.0  291.259888  447.029755
4  100.882095  255.756256  51291.074219  10.0 -397.168579  354.565308

           FSC-W      HDR-T         SSC-A         SSC-H           SSC-W  0  137847.578125  ␣
↪2.018511   840.091370   747.917847  147225.328125
1  -95849.679688  27.451754  3476.902344  3163.917969  144038.046875
2  -52700.828125  32.043865   480.270691   507.917877  123937.437500
3   85399.273438  79.327492  8026.275879  6741.838867  156043.484375
4 -146821.125000  79.731194  7453.750488   609.884277  262143.968750

         V2-A        V2-H           V2-W         Y2-A         Y2-H  0   41.593452  240.
↪153854   22701.017578   109.946274   153.630051
1  103.437637  336.153870   40332.058594  5554.108398  4273.629883
2 -271.375580  256.153870 -138860.828125    81.835281   121.630051
3  -26.212378  207.677841  -16543.453125   -54.120304    98.122017
4   44.559933  216.036865   27035.013672   -10.542595   127.326027

            Y2-W      T
0   93802.468750  False
1  170344.203125   True
2   88188.023438  False
3  -72294.242188  False
4  -10852.761719  False
```

In ex2, the `ThresholdOp` added a new column, named `T`, which has the same name as the `ThresholdOp`'s `name` attribute. It is `True` if the `Y2-A` value is greater than 200, and `False` otherwise. The `T` column is now a piece metadata *just like the Dox concentration* that we can use to facet the plots. For example, the following plots different `Dox` concentrations in separate plots stacked on top of eachother (`yfacet = "Dox"`), and on each of these plots uses separate colors for the "low" and "high" populations (`huefacet = 'T'`):

```
flow.HistogramView(channel = "Y2-A",
                   scale = "logicle",
                   yfacet = "Dox",
                   huefacet = 'T').plot(ex2)
```



It's also important to note that we can still access the underlying `pandas.Dataframe`: either by looking at the `Experiment.data` attribute, or by referencing a column directly:

```
ex2["T"].head(10)
```

```
0    False
1     True
2    False
3    False
4    False
5    False
6     True
7    False
8     True
9    False
Name: T, dtype: bool
```

Because the data is all stored in a single `pandas.Dataframe`, we can use the `pandas` API, and indeed the rest of the SciPy stack, to ask sophisticated questions of the underlying data. For example, "how many events of each Dox concentration were above the threshold?"

```
ex2.data.groupby(['Dox', 'T']).size()
```

```
Dox   T
1.0   False    9946
      True       54
10.0  False    5561
      True     4439
dtype: int64
```

`cytoflow` can answer the same question for us using one of its statistics views. In `cytoflow`, a *statistic* is a number that summarizes a population; one of the key features of `cytoflow` is that it makes it easy to see how these summary statistics change as your experimental conditions change.

Several operations add statistics to an experiment; one of the most straightforward ones is `ChannelStatisticOp`. It groups the experiment's data by the conditions specified in `by`, then applies `function` to `channel` for each group.

In this case, we'll divide up the data into the subgroups `T == True & Dox == 1`; `T == True & Dox == 10`; `T == False & Dox == 1`; and `T == False & Dox == 10`. Then, we'll apply the function `len` to the `Y2-A` channel.

```
ex3 = flow.ChannelStatisticOp(name = "ByDox",
                              channel = "Y2-A",
                              by = ['T', 'Dox'],
                              function = len).apply(ex2)
```

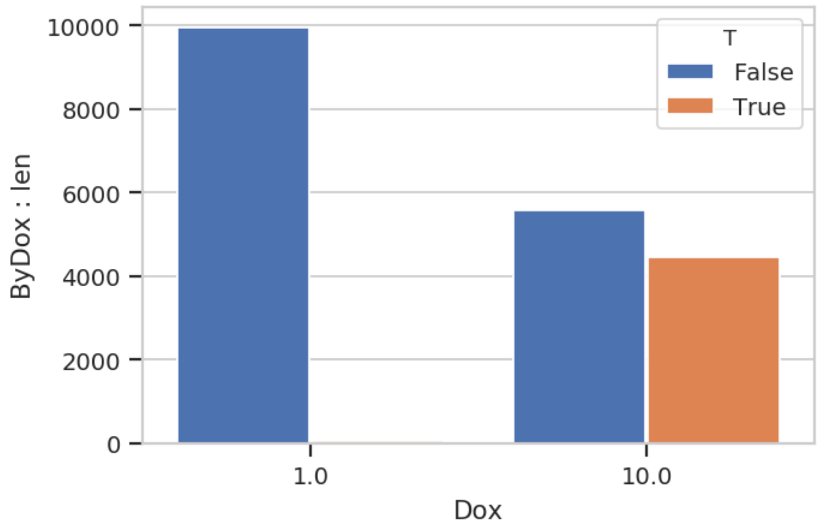
Now we can look at the new statistic: `Experiment.statistics` is a dictionary whose keys are tuples and whose values are the computed statistics (stored as `MultiIndexed pandas.Series`). The first element in the tuple is the name of the operation that added it, and the second is defined by that operation. In this case, it's `len`, the name of the function.

```
ex3.statistics[("ByDox", "len")]
```

```
T      Dox
False  1.0     9946
       10.0    5561
True   1.0       54
       10.0    4439
Name: ByDox : len, dtype: int64
```

We can also plot statistics using one of the various statistics views, such as `BarChartView`.

```
flow.BarChartView(statistic = ("ByDox", "len"),
                  variable = "Dox",
                  huefacet = 'T').plot(ex3)
```



Statistics are important enough that they get an entire notebook of examples; please see `Statistics.ipynb` for a more in-depth exploration.

I hope this makes the semantics of the `cytoflow` package clear. This was a pretty simplistic toy analysis; for more sophisticated examples, see the other accompanying Jupyter notebooks.

Interactive Plots

The `cytoflow` package is designed for both scripting and interactive use. As much as I would like the whole world to use data-driven gating and analysis methods, many workflows still require manually specifying gates, and this is most easily done by drawing the gate on a plot.

Fortunately, through the combination of `matplotlib` and the Jupyter notebook, we can have the best of both worlds: interactive REPL *and* plots that we can point-and-click with.

Clearly, this is best run interactively; but if you're looking at this notebook online, you should still get a flavor of the package's capabilities.

First, set up `Jupyter`'s `matplotlib` support, and import the `cytoflow` module. Note the `%matplotlib notebook` instead of `%matplotlib inline`.

```
# in some buggy versions of the Jupyter notebook, this needs to be in its OWN CELL.
%matplotlib notebook
```

```
import cytoflow as flow

# if your figures are too big or too small, you can scale them by changing matplotlib's␣
↪DPI
import matplotlib
matplotlib.rc('figure', dpi = 160)
```

Load a few example files, conditioning them on a float variable `Dox`.

```
tube1 = flow.Tube(file='data/RFP_Well_A3.fcs', conditions = {"Dox" : 10.0})
tube2 = flow.Tube(file='data/CFP_Well_A4.fcs', conditions = {"Dox" : 1.0})

import_op = flow.ImportOp(conditions = {"Dox" : "float"},
                          tubes = [tube1, tube2])

ex = import_op.apply()
```

Plot the Y2-A channel. We can see there's a bimodal distribution in one of the tubes.

```
flow.HistogramView(channel = "Y2-A",
                   scale = "logicle",
                   huefacet = "Dox").plot(ex)
```

```
<IPython.core.display.Javascript object>
```

Let's use a ThresholdOp to split out the top peak. You can get an interactive plot (the same HistogramView as above) by calling ThresholdOp's default_view() method. The view that gets returned is linked back to the ThresholdOp that produced it: it shows the proper channel, and when you draw a threshold on it the ThresholdOp instance's threshold trait gets updated.

One other thing to note: because the ThresholdOp's default view is derived from HistogramView, you can use all (well, most!) of the functionality in a regular HistogramView. Here, we'll use the the huefacet trait to plot the same multi-colored histogram as above.

As shown below, the steps for using an interactive view are: * Instantiate the operation * Call the operation's default_view() to get the interactive view. * Plot the view. * Set the view's interactive trait to True. This step can go before or after calling plot(). * Note that the view now shows a cursor (a vertical blue line) that follows your mouse as you move it around the view. Select the threshold you want, then click the mouse button to set it. A fixed blue line appears.

```
t = flow.ThresholdOp(name = "Threshold",
                     channel = "Y2-A")
tv = t.default_view()
tv.huefacet = "Dox"
tv.scale = "logicle"
tv.interactive = True

tv.plot(ex)
```

```
<IPython.core.display.Javascript object>
```

After you've drawn a threshold on the plot, look at the ThresholdOp instance's threshold trait and see that it matches the threshold you drew.

```
t.threshold
```

```
235.97817044140453
```

You can then apply the newly parameterized operation to the data set.

```
ex2 = t.apply(ex)
flow.HistogramView(channel = "Y2-A",
                   scale = "logicle",
```

(continues on next page)

(continued from previous page)

```
                  huefacet = "Dox",
                  yfacet = "Threshold").plot(ex2)
```

```
<IPython.core.display.Javascript object>
```

We can use a similar strategy with the RangeOp. The setup is very much the same; but instead of a single click, drag the cursor to set the range.

Also note that we can compress the invocation by passing the parameters to default_view() (as we would to a constructor.)

```
r = flow.RangeOp(name = "Range",
                 channel = "Y2-A")

r.default_view(huefacet = "Dox",
               scale = "logicle",
               interactive = True).plot(ex)
```

```
<IPython.core.display.Javascript object>
```

```
r.low, r.high
```

```
(2007.3540593264756, 11086.840684744158)
```

You can also draw ranges on 2D plots. Again, drag the cursor to draw a range.

```
r2d = flow.Range2DOp(name = "Range2D",
                     xchannel = "V2-A",
                     ychannel = "Y2-A")

r2d.default_view(huefacet = "Dox",
                 xscale = "logicle",
                 yscale = "logicle",
                 interactive = True).plot(ex)
```

```
<IPython.core.display.Javascript object>
```

```
r2d.xlow, r2d.xhigh, r2d.ylow, r2d.yhigh
```

```
(-203.01634412009793,
 104.94315196137035,
 2038.5106539405642,
 24562.81382553335)
```

You can specify a polygon this way too. Unforunately, the JavaScript link between the Jupyter notebook and the Python kernel is a little slow, so the performance here is … not ideal. Be patient.

Single click to set vertices; click the first vertex a second time to close the polygon.

```
p = flow.PolygonOp(name = "Polygon",
                   xchannel = "V2-A",
```

(continues on next page)

(continued from previous page)

```
                   ychannel = "Y2-A")

pv = p.default_view(huefacet = "Dox",
                    xscale = "logicle",
                    yscale = "logicle",
                    interactive = True)

pv.plot(ex)
```

```
<IPython.core.display.Javascript object>
```

```
p.vertices
```

```
[(-109.22381313438126, 4552.48571716678),
 (47.581169555466914, 23032.99288702212),
 (359.0848827803494, 23609.72249982525),
 (603.6648998740229, 5808.103794745581),
 (318.0855902021553, 1315.7691688032837),
 (41.025767639853456, 1513.9301298793557),
 (41.025767639853456, 1513.9301298793557),
 (-15.72537611199034, 1702.8785363694597),
 (-15.72537611199034, 1702.8785363694597)]
```

You can also specify a quadrant (or quad) gate. Move the cursor to where you want it; click to set the gate.

```
q = flow.QuadOp(name = "Quad",
                xchannel = "V2-A",
                ychannel = "Y2-A")

qv = q.default_view(huefacet = "Dox",
                    xscale = "logicle",
                    yscale = "logicle",
                    interactive = True)

qv.plot(ex)
```

```
<IPython.core.display.Javascript object>
```

```
q.xthreshold, q.ythreshold
```

```
(278.2187356624719, 5531.465729647518)
```

Machine Learning for Flow Cytometry

One of the directions `cytoflow` is going in that I'm most excited about is the application of advanced machine learning methods to flow cytometry analysis. After all, cytometry data is just a high-dimensional data set with many data points: making sense of it can take advantage of some of the sophisticated methods that have seen great success with other high-throughput biological data (such as microarrays.)

The following notebook demonstrates a heavily-commented machine learning method, Gaussian Mixture Models, applied to the demo data set we worked with in "Basic Cytometry." Then, there are briefer examples of some of the other machine learning methods that are implemented in `cytoflow`.

Set up the Jupyter notebook and import `cytoflow`.

```
# in some buggy versions of the Jupyter notebook, this needs to be in its OWN CELL.
%matplotlib inline
```

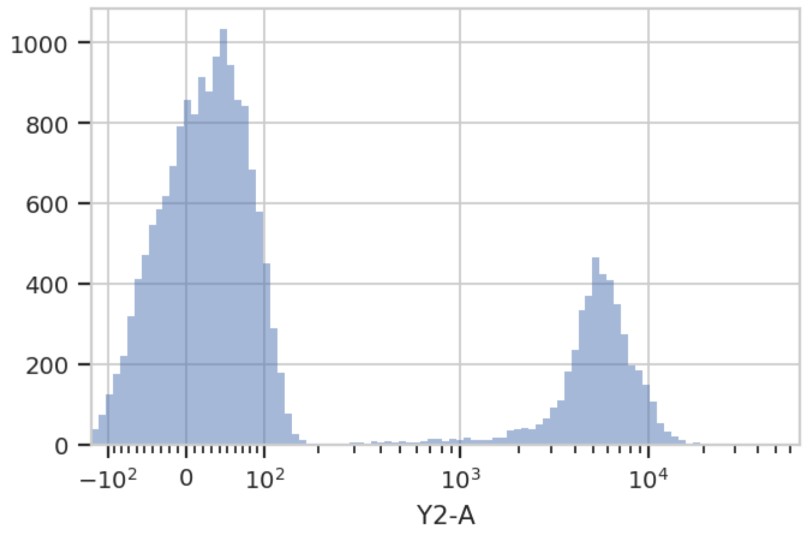
```
import cytoflow as flow

# if your figures are too big or too small, you can scale them by changing matplotlib's␣
↪DPI
import matplotlib
matplotlib.rc('figure', dpi = 160)
```

We have two `Tubes` of data that we specify were treated with two different concentrations of the inducer Doxycycline.

```
tube1 = flow.Tube(file = 'data/RFP_Well_A3.fcs',
                  conditions = {"Dox" : 10.0})
tube2 = flow.Tube(file='data/CFP_Well_A4.fcs',
                  conditions = {"Dox" : 1.0})

import_op = flow.ImportOp(conditions = {"Dox" : "float"},
                          tubes = [tube1, tube2],
                          channels = {'V2-A' : 'V2-A',
                                      'Y2-A' : 'Y2-A'})

ex = import_op.apply()
```

Let's look at the histogram of the `Y2-A` channel (on a `logicle` plot scale).

```
flow.HistogramView(scale = 'logicle',
                   channel = 'Y2-A').plot(ex)
```

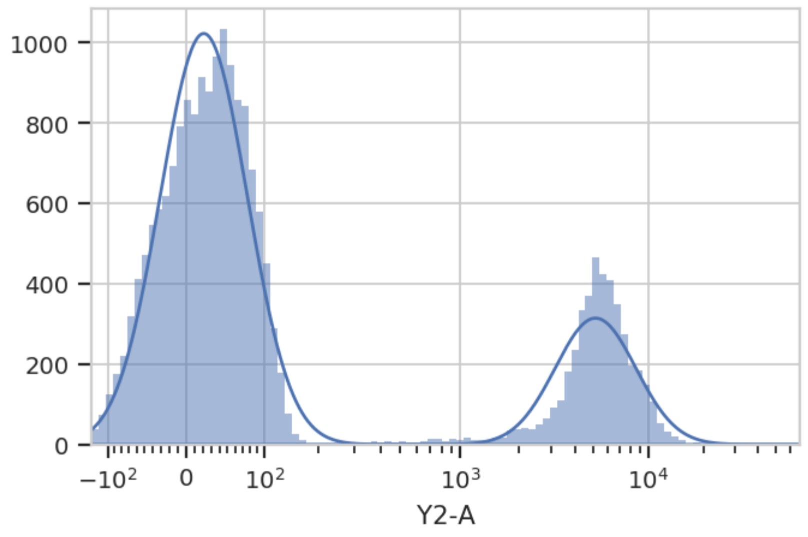


This data looks bimodal to me! Not perfect Gaussians, but close enough that using a Gaussian Mixture Model will probably let us separate them.

Let's use the `GaussianMixtureOp` to separate these two populations. In operations that are paramterized by data (either an Experiment or some auxilliary FCS file), `cytoflow` separates the estimation of module parameters from their application. Thus, after instantiating the operation, you call `estimate()` to estimate the model parameters. Those parameters stay associated with the operation instance in the same way instances of `ThresholdOp` have the gate threshold as an instance attribute.

Additionally, many modules, including `GaussianMixtureModelOp`, have a `default_view()` factory method that returns a diagnostic plot so you can check to see that the parameter estimation worked. This is particularly important for unsupervised learning methods! In this case, the `GaussianMixtureModelOp`'s `default_view()` returns a `View` that plots a histogram, colored by the component each event was assigned to, and an overlay of the Gaussian distributions on top of the histogram.

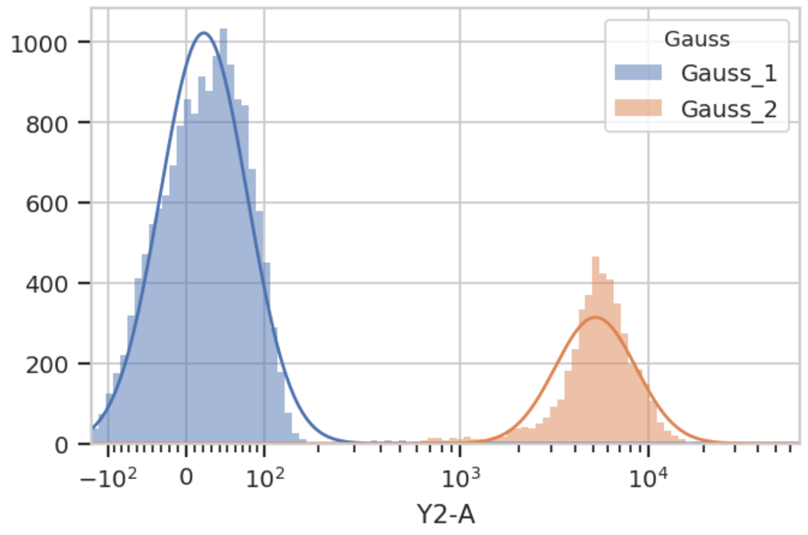
```
g = flow.GaussianMixtureOp(name = "Gauss",
                           channels = ["Y2-A"],
                           scale = {"Y2-A" : "logicle"},
                           num_components = 2)

g.estimate(ex)
g.default_view().plot(ex)
```



Excellent. It looks like the GMM found the two distributions we were looking for. Let's call `apply()`, then use the operation's default view to plot the new `Experiment`.

```
ex2 = g.apply(ex)
g.default_view().plot(ex2)
```

```
/home/brian/src/cytoflow/cytoflow/operations/base_op_views.py:341: CytoflowViewWarning:␣
↪Setting 'huefacet' to 'Gauss'
```



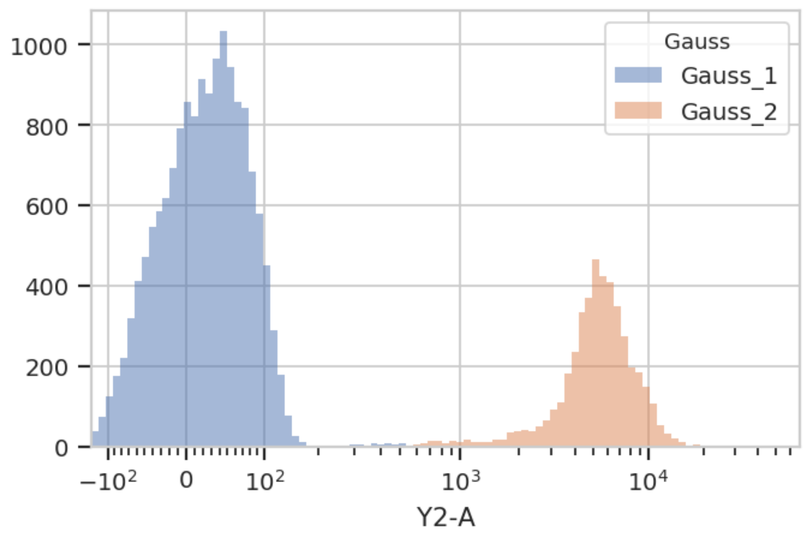
When you `apply()` the `GaussianMixtureModelOp`, it adds a new piece of metadata to each event in the data set: a condition whose name that's the same as the name of the operation (in this case, `Gauss`).

The values are `{Name}_{#}`, where `{Name}` is the name of the operation and `{#}` is which population had the highest posterior probability. So, in this example the events would be labeled `Gauss_1` or `Gauss_2`.

```
ex2.data.head()
```

We can use that new condition to plot or compute or otherwise operate on each of the populations separately:

```
flow.HistogramView(channel = "Y2-A",
                   scale = "logicle",
                   huefacet = "Gauss").plot(ex2)
```



There's an important subtlety to notice here. In the plot above, we set the data scale on the `HistogramView`, but prior to that we passed `scale = {"Y2-A" : "logicle"}` to the `GaussianMixtureOp` *operation*. We did so in order to fit the gaussian model to the *scaled data*, as opposed to the raw data.

This is an example of a broader design goal: in order to enable more quantitative analysis, `cytoflow` does not rescale the underlying data; rather, it transforms it before displaying it. Frequently it is useful to perform the same transformation before doing data-driven things, so many modules that have an `estimate()` function also allow you to specify a `scale`.

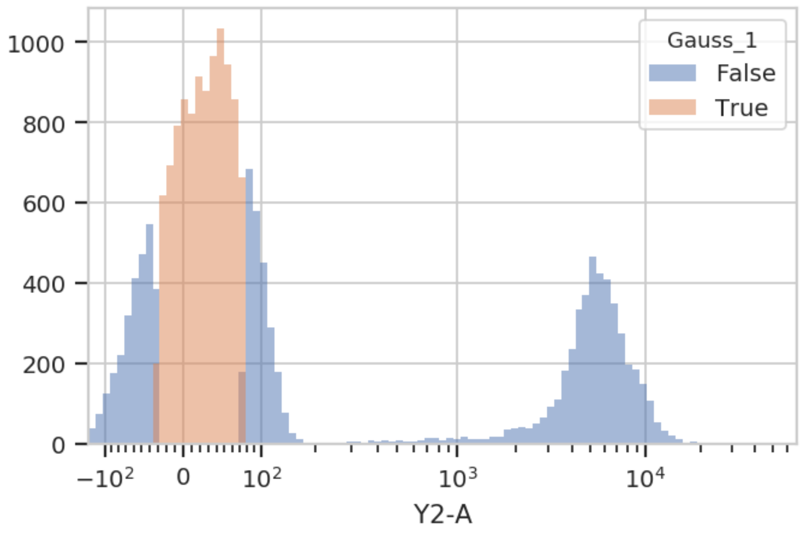
A 1-dimensional gaussian mixture model works well if the populations are well-separated. However, if they're closer together, you may only want to keep events that are "clearly" in one distribution or another. One way to accomplish this by passing a `sigma` parameter to `GaussianMixtureOp`. This doesn't change the behavior of `estimate()`, but when you `apply()` the operation it creates new conditions, one for each population. The conditions are named `{Name}_{#}`, where `{Name}` is the name of the operation and `{#}` is the index of the population. The value of the condition is `True` for an event if that event is within `sigma` standard deviations of the population mean.

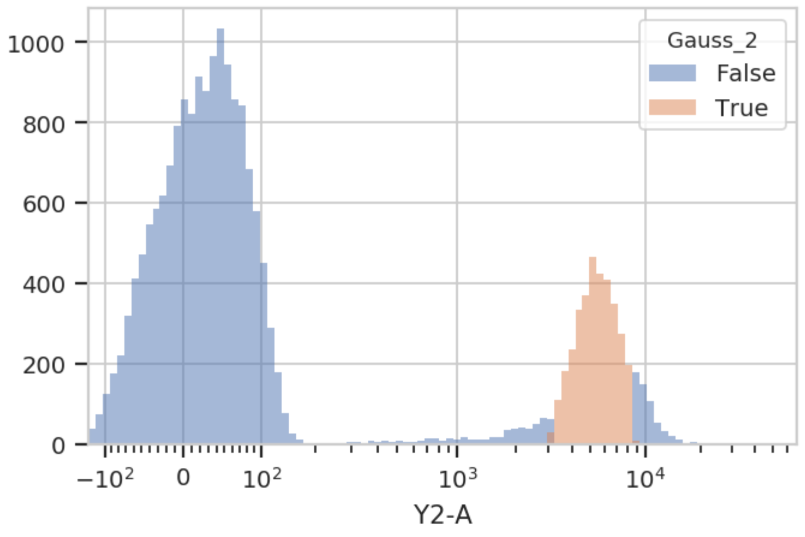
```
g = flow.GaussianMixtureOp(name = "Gauss",
                           channels = ["Y2-A"],
                           scale = {"Y2-A" : "logicle"},
                           num_components = 2,
                           sigma = 1)

g.estimate(ex)
ex2 = g.apply(ex)

flow.HistogramView(channel = "Y2-A",
```

(continues on next page)

(continued from previous page)

```
                    huefacet = "Gauss_1",
                    scale = "logicle").plot(ex2)

flow.HistogramView(channel = "Y2-A",
                    huefacet = "Gauss_2",
                    scale = "logicle").plot(ex2)
```





Sometimes, mixtures are very close and separating them is difficult. In such cases it may be better to filter the events based on the *posterior probability* that they are actually members of the components to which they were assigned. We can get this behavior by passing `posterior = True` as a parameter to `GaussianMixture1DOp`.

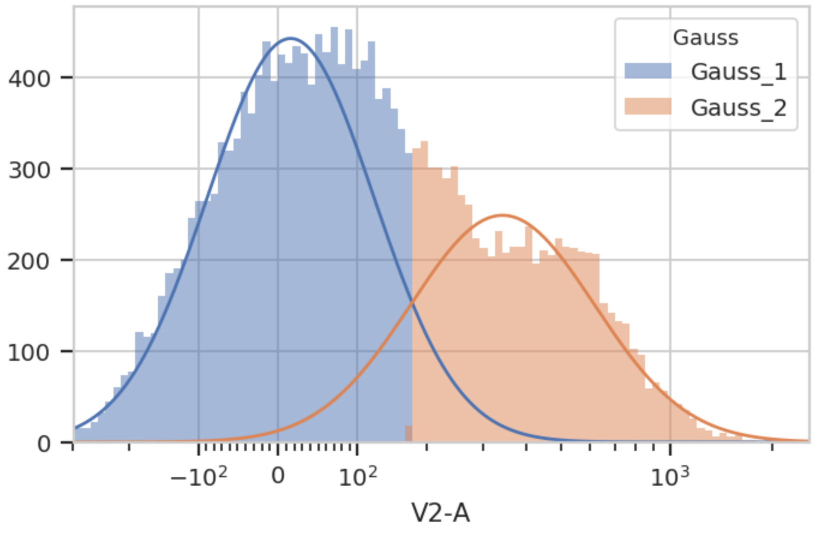
```
g = flow.GaussianMixtureOp(name = "Gauss",
                          channels = ["V2-A"],
                          scale = {"V2-A" : "logicle"},
                          num_components = 2,
                          posteriors = True)
```

(continues on next page)

(continued from previous page)

```
g.estimate(ex)
ex2 = g.apply(ex)
g.default_view().plot(ex2)
```

```
/home/brian/src/cytoflow/cytoflow/operations/base_op_views.py:341: CytoflowViewWarning:␣
↪Setting 'huefacet' to 'Gauss'
```



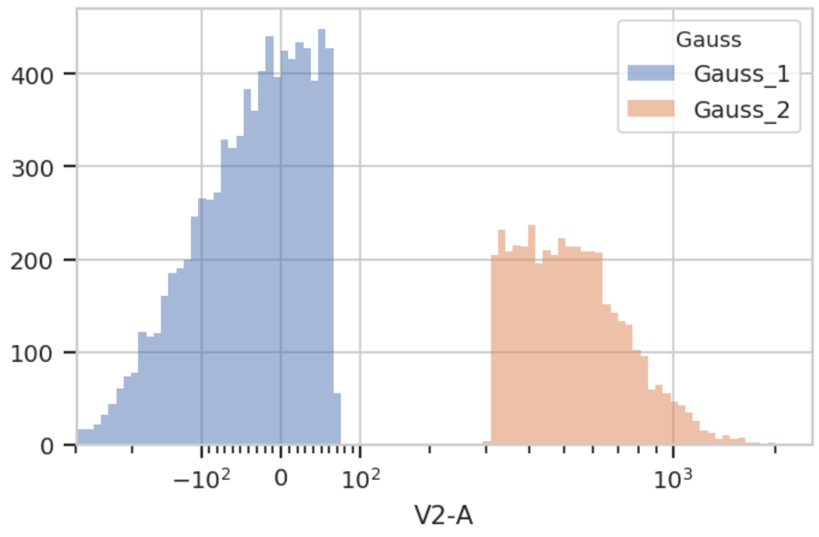
If posteriors = True, the GaussianMixtureOp.apply() adds another metadata column, {Name}_{#}_Posterior, that contains the posterior probability of each event in that component.

```
ex2.data.head()
```

We can use this second metadata column to filter out events with low posterior probabilities:

```
ex2.query("Gauss_1_posterior > 0.9 | Gauss_2_posterior > 0.9").data.head()
```

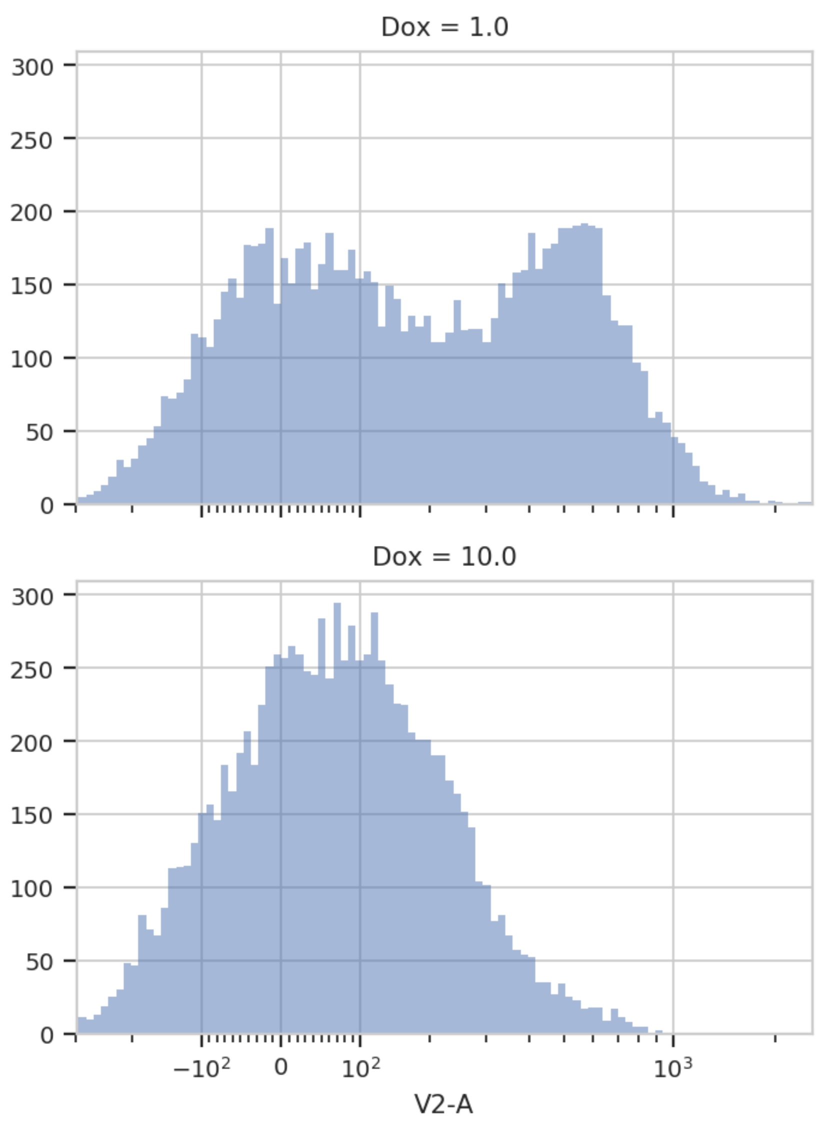
```
flow.HistogramView(channel = "V2-A",
                   huefacet = "Gauss",
                   scale = "logicle",
                   subset = "Gauss_1_posterior > 0.9 | Gauss_2_posterior > 0.9").
↪plot(ex2)
```



Finally, sometimes you don't want to use the same model parameters for your entire data set. Instead, you want to estimate different parameters for different subsets. `cytoflow`'s data-driven modules allow you to do so with a `by` parameter, which *aggregates* subsets before estimating model parameters. You pass `by` an array of metadata columns, and the module estimates a new model for each unique subset of those metadata.

For example, let's look at the `V2-A` channel faceted by `Dox`:

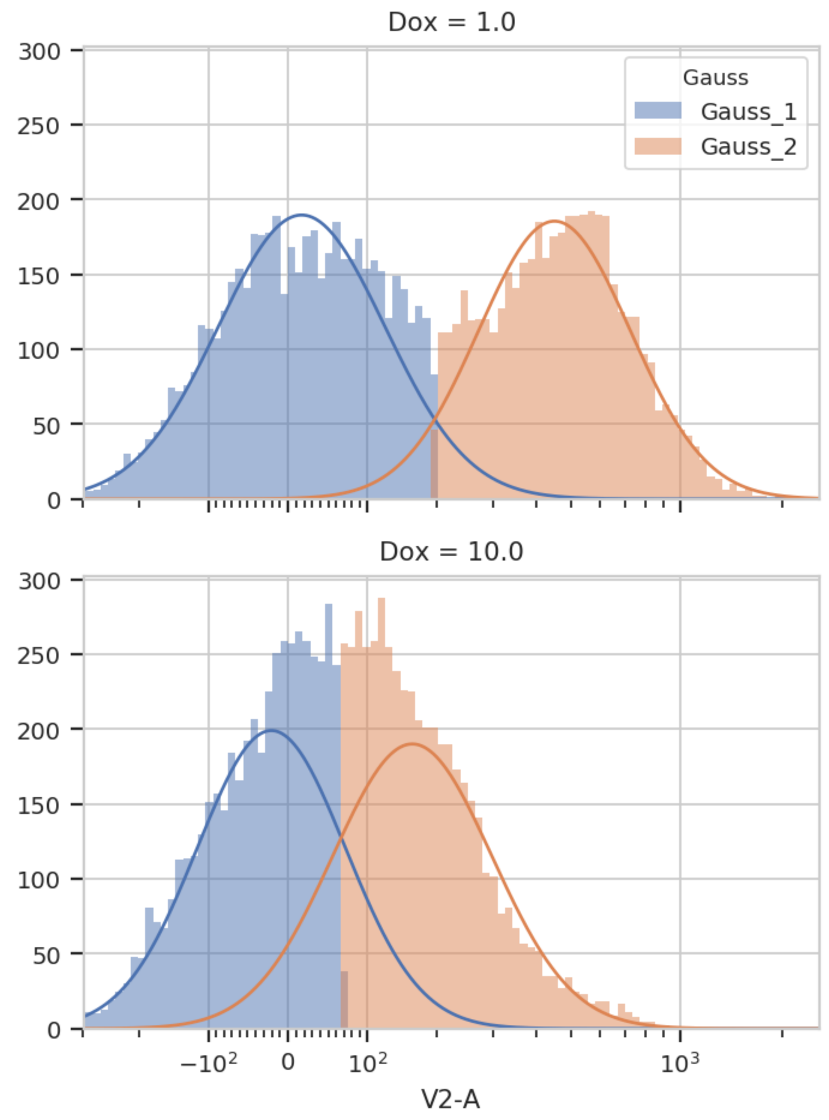
```
flow.HistogramView(channel = "V2-A",
                   scale = "logicle",
                   yfacet = "Dox").plot(ex)
```



It's pretty clear that the `Dox == 1.0` condition and the `Dox == 10.0` condition are different; a single 2-component GMM won't fit both of them. So let's fit a model to each unique value of `Dox`. Note that `by` takes a *list* of metadata columns; you must pass it a list, even if there's only one element in the list.

```
g = flow.GaussianMixtureOp(name = "Gauss",
                           channels = ["V2-A"],
                           scale = {"V2-A" : "logicle"},
                           num_components = 2,
                           by = ["Dox"])

g.estimate(ex)
ex2 = g.apply(ex)
g.default_view(yfacet = "Dox").plot(ex2)
```

```
/home/brian/src/cytoflow/cytoflow/operations/base_op_views.py:341: CytoflowViewWarning:␣
↪Setting 'huefacet' to 'Gauss'
```



You can see that the models for the two subsets are substantially different.

It is important to note that while the estimated model parameters differ between subsets, it is not currently possible to specify different *module parameters* across subsets. For example, you can't specify that the `Dox = 1.0` subset have two GMM components, but `Dox == 10.0` have only one. If we could *estimate* the number of components, on the other hand, using (say) an AIC or BIC information criterion, then different subsets could have different numbers of components. For this kind of unsupervized algorithm, see the `FlowPeaksOp` example below.

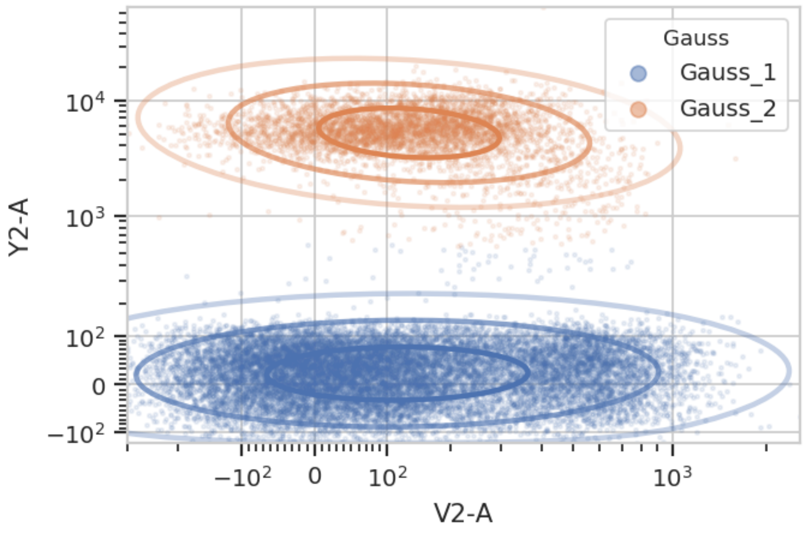
2D Gaussian Mixture Models

Did you notice how we were setting the `channels` attribute of `GaussianMixtureOp` to a one-element list? That's because `GaussianMixtureOp` will work in any number of dimensions. Here's a similar workflow in two channels instead of one:

Basic usage, assigning each event to one of the mixture components: (the isolines in the `default_view()` are 1, 2 and 3 standard deviations away from the mean.)

```
g = flow.GaussianMixtureOp(name = "Gauss",
                           channels = ["V2-A", "Y2-A"],
                           scale = {"V2-A" : "logicle",
                                    "Y2-A" : "logicle"},
                           num_components = 2)
g.estimate(ex)
ex2 = g.apply(ex)
g.default_view().plot(ex2, alpha = 0.1)
```

```
/home/brian/src/cytoflow/cytoflow/operations/base_op_views.py:341: CytoflowViewWarning:␣
↪Setting 'huefacet' to 'Gauss'
```



Subsetting based on standard deviation. Note: we use the Mahalnobis distance as a multivariate generalization of the number of standard deviations an event is from the mean of the multivariate gaussian.

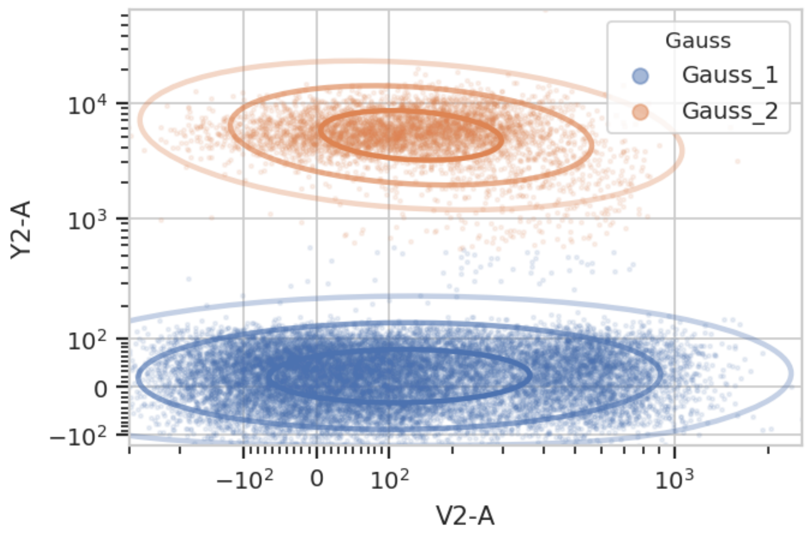
```
g = flow.GaussianMixtureOp(name = "Gauss",
                           channels = ["V2-A", "Y2-A"],
                           scale = {"V2-A" : "logicle",
                                    "Y2-A" : "logicle"},
                           num_components = 2,
                           sigma = 2)

g.estimate(ex)
ex2 = g.apply(ex)
```

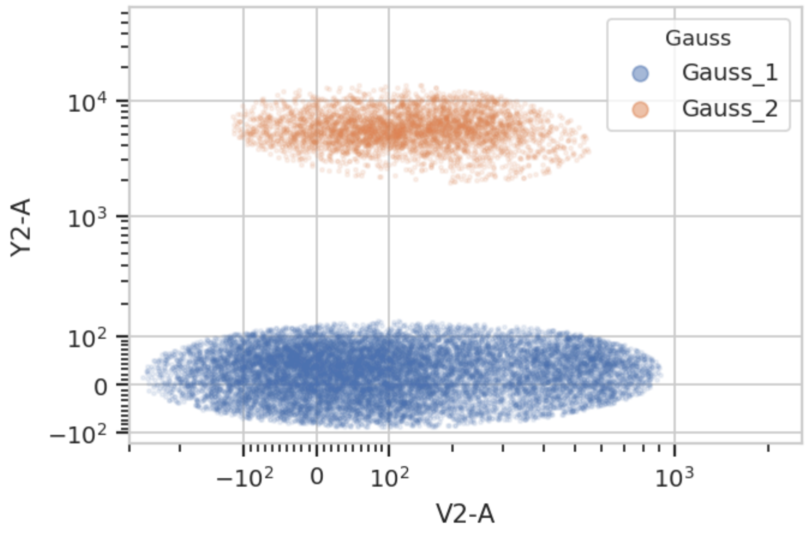
(continues on next page)

(continued from previous page)

```
g.default_view().plot(ex2, alpha = 0.1)

flow.ScatterplotView(xchannel = "V2-A",
                     ychannel = "Y2-A",
                     xscale = "logicle",
                     yscale = "logicle",
                     huefacet = "Gauss",
                     subset = "Gauss_1 == True | Gauss_2 == True").plot(ex2, alpha = 0.1)
```

```
/home/brian/src/cytoflow/cytoflow/operations/base_op_views.py:341: CytoflowViewWarning:␣
↪Setting 'huefacet' to 'Gauss'
```





Gating based on posterior probabilities:

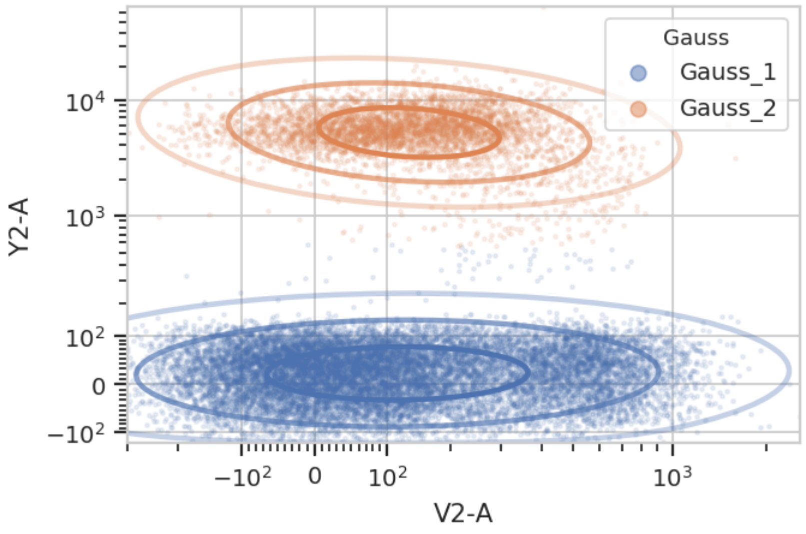
```
g = flow.GaussianMixtureOp(name = "Gauss",
```

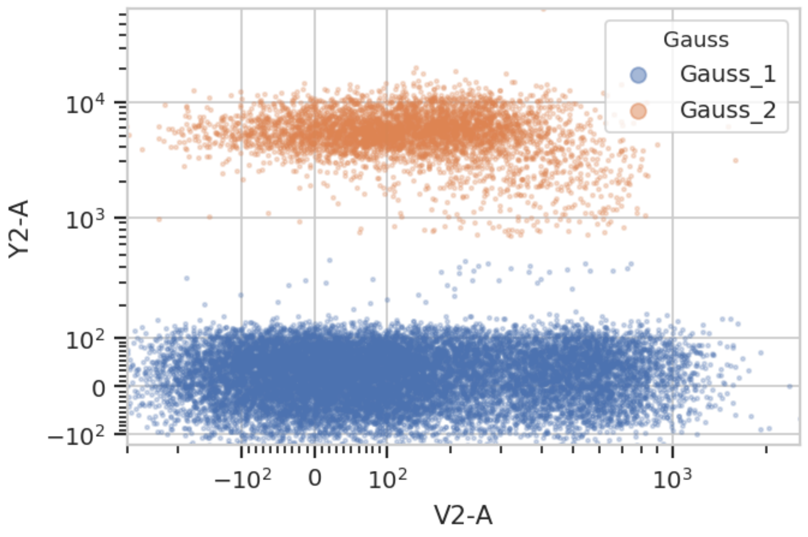
(continues on next page)

(continued from previous page)

```
                                        channels = ["V2-A", "Y2-A"],
                                        scale = {"V2-A" : "logicle",
                                                 "Y2-A" : "logicle"},
                                        num_components = 2,
                                        posteriors = True)

g.estimate(ex)
ex2 = g.apply(ex)

g.default_view().plot(ex2, alpha = 0.1)

flow.ScatterplotView(xchannel = "V2-A",
                     ychannel = "Y2-A",
                     xscale = "logicle",
                     yscale = "logicle",
                     huefacet = "Gauss",
                     subset = "Gauss_1_posterior > 0.99 | Gauss_2_posterior > 0.99").
↪plot(ex2)
```

```
/home/brian/src/cytoflow/cytoflow/operations/base_op_views.py:341: CytoflowViewWarning:␣
↪Setting 'huefacet' to 'Gauss'
```

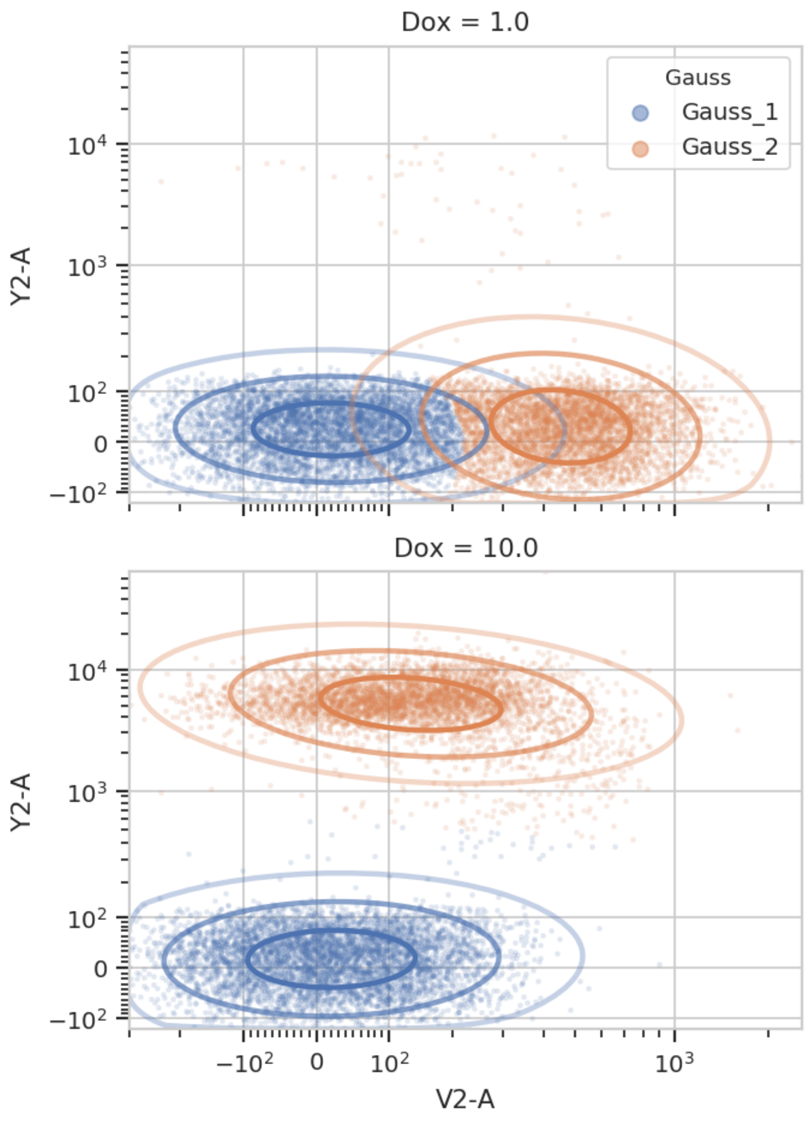




And multiple models for different subsets:

```
g = flow.GaussianMixtureOp(name = "Gauss",
                           channels = ["V2-A", "Y2-A"],
                           scale = {"V2-A" : "logicle",
                                    "Y2-A" : "logicle"},
                           num_components = 2,
                           by = ['Dox'])

g.estimate(ex)
ex2 = g.apply(ex)

g.default_view(yfacet = "Dox").plot(ex2, alpha = 0.1)
```

```
/home/brian/src/cytoflow/cytoflow/operations/base_op_views.py:341: CytoflowViewWarning:␣
↪Setting 'huefacet' to 'Gauss'
```

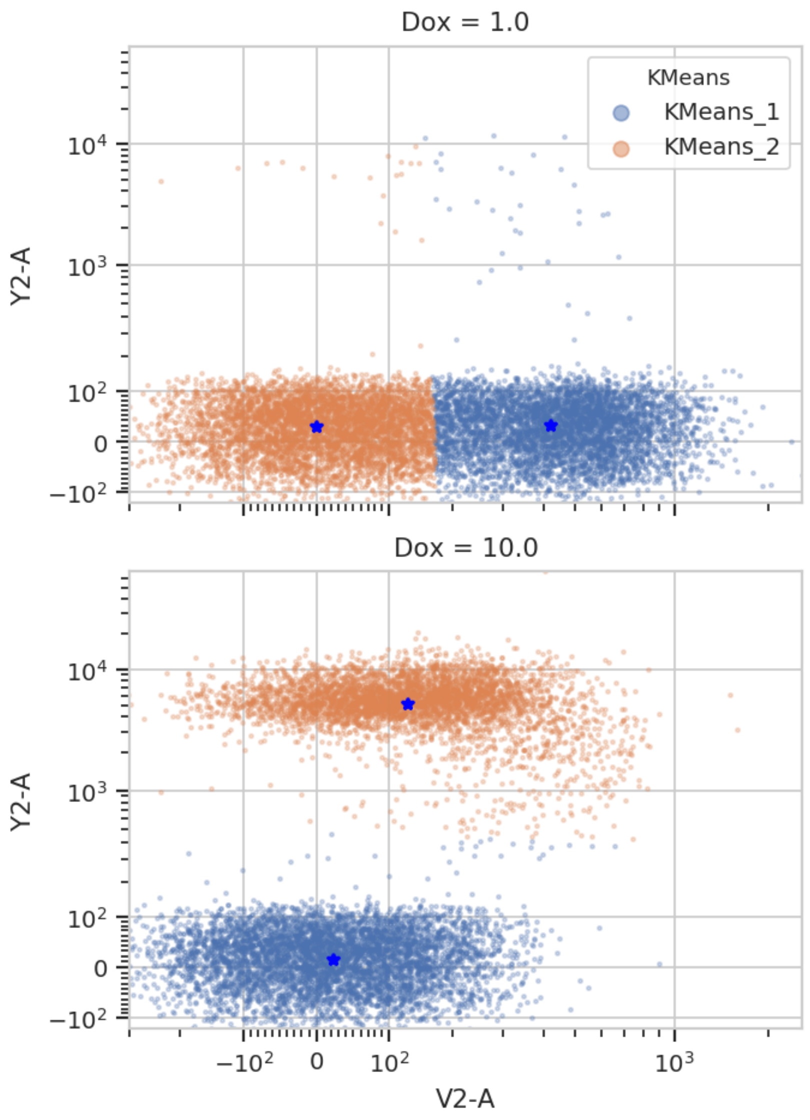


K-Means

A gaussian mixture model can find clustered data, but it works best on clusters that are shaped like gaussian distributions (duh.) K-means clustering is much more general. As with the GMM operation, `cytoflow`'s K-means operation is N-dimensional and is parameterized very similarly. Here's a demonstration in 2D.

```
k = flow.KMeansOp(name = "KMeans",
                  channels = ["V2-A", "Y2-A"],
                  scale = {"V2-A" : "logicle",
                           "Y2-A" : "logicle"},
                  num_clusters = 2,
                  by = ['Dox'])

k.estimate(ex)
ex2 = k.apply(ex)
k.default_view(yfacet = "Dox").plot(ex2)
```

```
/home/brian/src/cytoflow/cytoflow/operations/base_op_views.py:341: CytoflowViewWarning:␣
↪Setting 'huefacet' to 'KMeans'
```



Note how the centroids are marked with blue stars. The locations of the centroids are also saved as a new statistic:

```
ex2.statistics
```

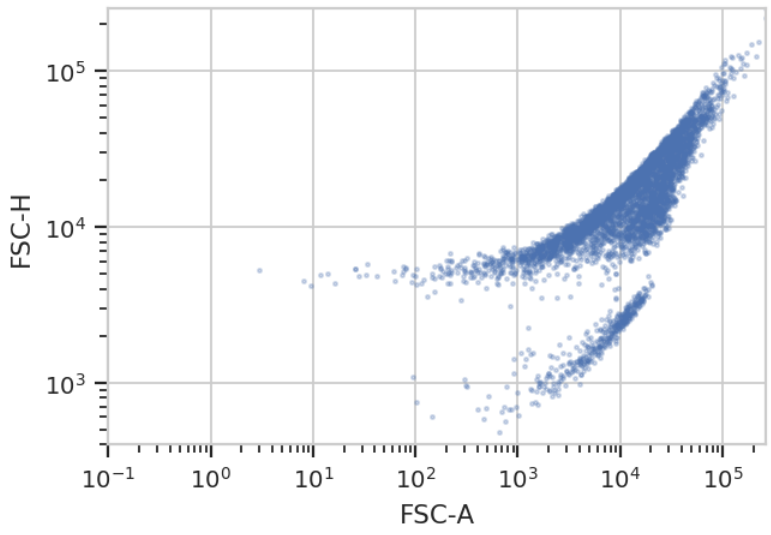
```
{('KMeans', 'centers'): Dox   Cluster  Channel
 1.0   1        V2-A        420.424332
                Y2-A         29.304036
       2        V2-A          1.276445
                Y2-A         26.576759
 10.0  1        V2-A         23.096459
                Y2-A         14.118368
       2        V2-A        127.770348
                Y2-A       5113.999433
 dtype: float64}
```

FlowPeaks

Sometimes you want to cluster a data set and K-means just doesn't work. (It generally works best when clusters are fairly compact.) In such cases, the unsupervized clustering algorith `flowPeaks` may work better for you. For example, the following FCS file (of an E. coli experiment) shows clear separation between the cells (upper population) and the particulate matter in the media (lower population.)

```
ex = flow.ImportOp(tubes = [flow.Tube(file = 'data/ecoli.fcs')]).apply()
```

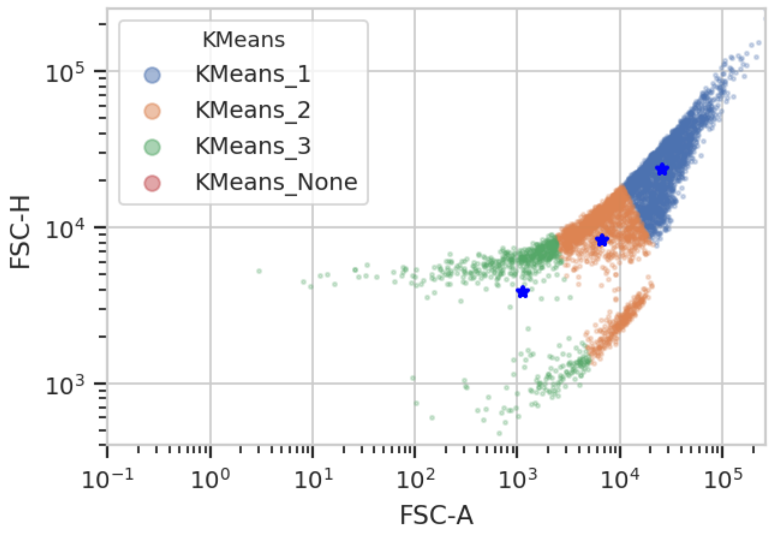
```
flow.ScatterplotView(xchannel = "FSC-A",
                     xscale = 'log',
                     ychannel = "FSC-H",
                     yscale = 'log').plot(ex)
```



Unfortunately, K-means (even with more clusters than I want) isn't finding them properly.

```
k = flow.KMeansOp(name = "KMeans",
                  channels = ["FSC-A", "FSC-H"],
                  scale = {"FSC-A" : "log",
                           "FSC-H" : "log"},
                  num_clusters = 3)

k.estimate(ex)
ex2 = k.apply(ex)
k.default_view().plot(ex2)
```

```
/home/brian/src/cytoflow/cytoflow/operations/base_op_views.py:341: CytoflowViewWarning:␣
↪Setting 'huefacet' to 'KMeans'
```

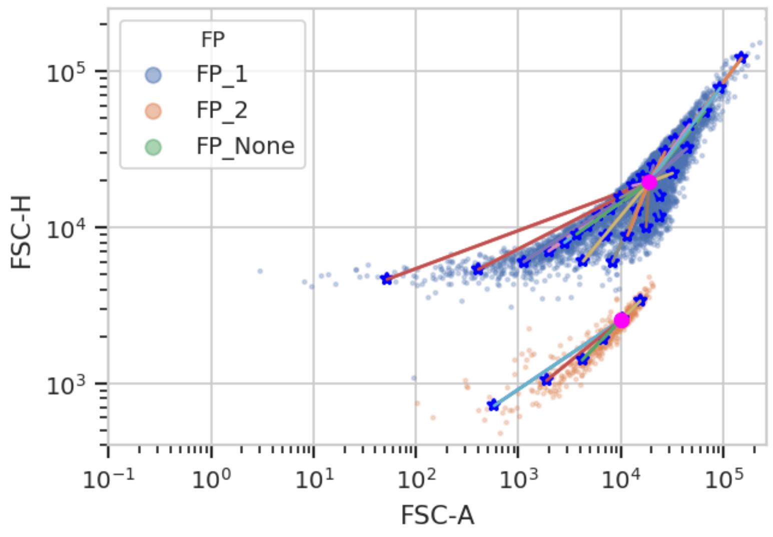


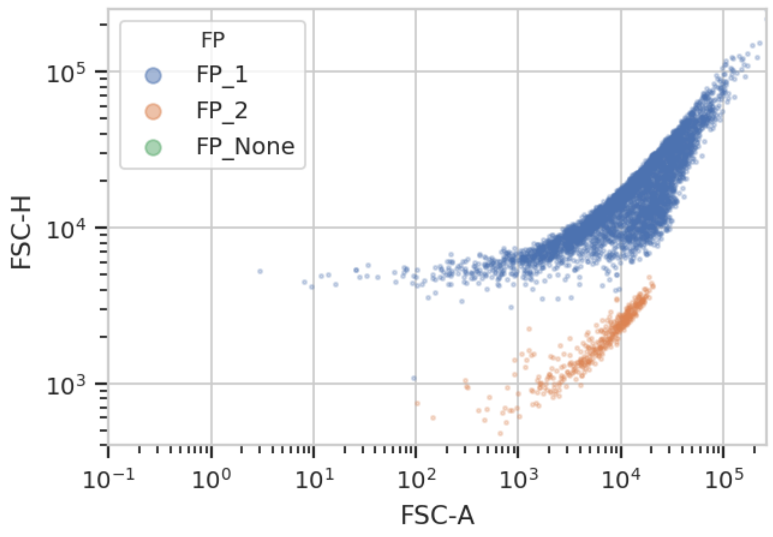
flowPeaks is an unsupervized learning strategy developed specifically for flow cytometry, and it works much better here. It can automatically determine the number of "natural" clusters in a data set. It is also much more computationally expensive – if you have a large data set, be prepared to wait for a while!

Another note – there are a number of hyperparameters for this method that can dramatically change how it operates. The defaults are good for many data sets; for the one below, I had to decrease h0 from 1.0 to 0.5. See the documentation for details.

```
fp = flow.FlowPeaksOp(name = "FP",
                      channels = ["FSC-A", "FSC-H"],
                      scale = {"FSC-A" : "log",
                               "FSC-H" : "log"},
                      h0 = 0.5)

fp.estimate(ex)
ex2 = fp.apply(ex)
fp.default_view().plot(ex2)
flow.ScatterplotView(xchannel = "FSC-A",
                     xscale = "log",
                     ychannel = "FSC-H",
                     yscale = "log",
                     huefacet = "FP").plot(ex2)
```

```
/home/brian/src/cytoflow/cytoflow/operations/base_op_views.py:341: CytoflowViewWarning:␣
↪Setting 'huefacet' to 'FP'
```





Statistics in `cytoflow`

One of the most powerful concepts in `cytoflow` is that it makes it easy to summarize your data, then track how those subsets change as your experimental variables change. This notebook demonstrates several different modules that create and plot statistics.

Set up the Jupyter `matplotlib` integration, and import the `cytoflow` module.

```
# in some buggy versions of the Jupyter notebook, this needs to be in its OWN CELL.
%matplotlib inline
```

```
import cytoflow as flow

# if your figures are too big or too small, you can scale them by changing matplotlib's␣
↪DPI
```

(continues on next page)

(continued from previous page)

```
import matplotlib
matplotlib.rc('figure', dpi = 160)
```

We use the same data set as the **Yeast Dose Response** example notebook, with one variant: we load each tube three times, grabbing only 100 events from each.

```
inputs = {
    "Yeast_B1_B01.fcs" : 5.0,
    "Yeast_B2_B02.fcs" : 3.75,
    "Yeast_B3_B03.fcs" : 2.8125,
    "Yeast_B4_B04.fcs" : 2.109,
    "Yeast_B5_B05.fcs" : 1.5820,
    "Yeast_B6_B06.fcs" : 1.1865,
    "Yeast_B7_B07.fcs" : 0.8899,
    "Yeast_B8_B08.fcs" : 0.6674,
    "Yeast_B9_B09.fcs" : 0.5,
    "Yeast_B10_B10.fcs" : 0.3754,
    "Yeast_B11_B11.fcs" : 0.2816,
    "Yeast_B12_B12.fcs" : 0.2112,
    "Yeast_C1_C01.fcs" : 0.1584,
    "Yeast_C2_C02.fcs" : 0.1188,
    "Yeast_C3_C03.fcs" : 0.0892,
    "Yeast_C4_C04.fcs" : 0.0668,
    "Yeast_C5_C05.fcs" : 0.05,
    "Yeast_C6_C06.fcs" : 0.0376,
    "Yeast_C7_C07.fcs" : 0.0282,
    "Yeast_C8_C08.fcs" : 0.0211,
    "Yeast_C9_C09.fcs" : 0.0159
}

tubes = []
for filename, ip in inputs.items():
    tubes.append(flow.Tube(file = "data/" + filename, conditions = {'IP' : ip, 'Replicate
↪' : 1}))
    tubes.append(flow.Tube(file = "data/" + filename, conditions = {'IP' : ip, 'Replicate
↪' : 2}))
    tubes.append(flow.Tube(file = "data/" + filename, conditions = {'IP' : ip, 'Replicate
↪' : 3}))


ex = flow.ImportOp(conditions = {'IP' : "float", "Replicate" : "int"},
                   tubes = tubes,
                   events = 100).apply()
```

In `cytoflow`, a *statistic* is a value that summarizes something about some data. `cytoflow` makes it easy to compute statistics for various subsets. For example, if we expect the geometric mean of `FITC-A` channel to change as the `IP` variable changes, we can compute the geometric mean of the `FITC-A` channel of each different `IP` condition with the `ChannelStatisticOp` operation:

```
op = flow.ChannelStatisticOp(name = "ByIP",
                             by = ["IP"],
                             channel = "FITC-A",
```

(continues on next page)

(continued from previous page)

```
                                 function = flow.geom_mean)
ex2 = op.apply(ex)
```

This operation splits the data set by different values of `IP`, then applies the function `flow.geom_mean` to the `FITC-A` channel in each subset. The result is stored in the `statistics` attribute of an `Experiment`. The `statistics` attribute is a dictionary. The keys are tuples, where the first element in the tuple is the name of the operation that created the statistic, and the second element is specific to the operation:

```
ex2.statistics.keys()
```

```
dict_keys([('ByIP', 'geom_mean')])
```

The `Statistics1DOp` operation sets the second element of the tuple to the function name. (You can override this by setting `Statistics1DOp.statistic_name`; this is useful if `function` is a lambda function.)

The value of each entry in `Experiment.statistics` is a `pandas.Series`. The series index is all the subsets for which the statistic was computed, and the contents are the values of the statstic itself.

```
ex2.statistics[('ByIP', 'geom_mean')]
```

```
IP
0.0159     106.760464
0.0211     145.242912
0.0282     152.152489
0.0376     194.243504
0.0500     240.788179
0.0668     413.218589
0.0892     670.798705
0.1188     971.121453
0.1584    1345.581190
0.2112    1203.016726
0.2816    1463.993731
0.3754    1733.702533
0.5000    1871.968527
0.6674    2218.190937
0.8899    2324.630935
1.1865    2486.103908
1.5820    2456.024025
2.1090    2326.584475
2.8125    2350.038991
3.7500    2676.275419
5.0000    2269.741514
Name: ByIP : geom_mean, dtype: float64
```

We can also specify multiple variables to break data set into. In the example above, `Statistics1DOp` lumps all events with the same value of `IP` together, but each amount of `IP` actually has three values of `Replicate` as well. Let's apply `geom_mean` to each unique combination of `IP` and `Replicate`:

```
op = flow.ChannelStatisticOp(name = "ByIP",
                             by = ["IP", "Replicate"],
                             channel = "FITC-A",
                             function = flow.geom_mean)
```

(continues on next page)

(continued from previous page)

```
ex2 = op.apply(ex)
ex2.statistics[("ByIP", "geom_mean")][0:12]
```

```
IP      Replicate
0.0159  1            109.294581
        2            106.956580
        3            104.099598
0.0211  1            144.564288
        2            147.712897
        3            143.484638
0.0282  1            155.558697
        2            156.508248
        3            144.679053
0.0376  1            182.294878
        2            206.147145
        3            195.023849
Name: ByIP : geom_mean, dtype: float64
```

Note that the `pandas.Series` now has a `MultiIndex`: there are values for each unique combination of `IP` and `Replicate`.

Now that we have computed a statistic, we can plot it with one of the statistics views. We can use a bar chart:

```
flow.BarChartView(statistic = ("ByIP", "geom_mean"),
                  variable = "IP").plot(ex2)
```

```
---------------------------------------------------------------------------


CytoflowViewError                         Traceback (most recent call last)

<ipython-input-8-4102c59fbf66> in <module>
      1 flow.BarChartView(statistic = ("ByIP", "geom_mean"),
----> 2                   variable = "IP").plot(ex2)



~/src/cytoflow/cytoflow/views/bar_chart.py in plot(self, experiment, plot_name, **kwargs)
    129         """
    130 
--> 131         super().plot(experiment, plot_name, **kwargs)
    132 
    133     def _grid_plot(self, experiment, grid, **kwargs):



~/src/cytoflow/cytoflow/views/base_views.py in plot(self, experiment, plot_name,␣
↪**kwargs)
    859                       plot_name = plot_name,
    860                       scale = scale,
--> 861                       **kwargs)
    862 
    863     def _make_data(self, experiment):
```

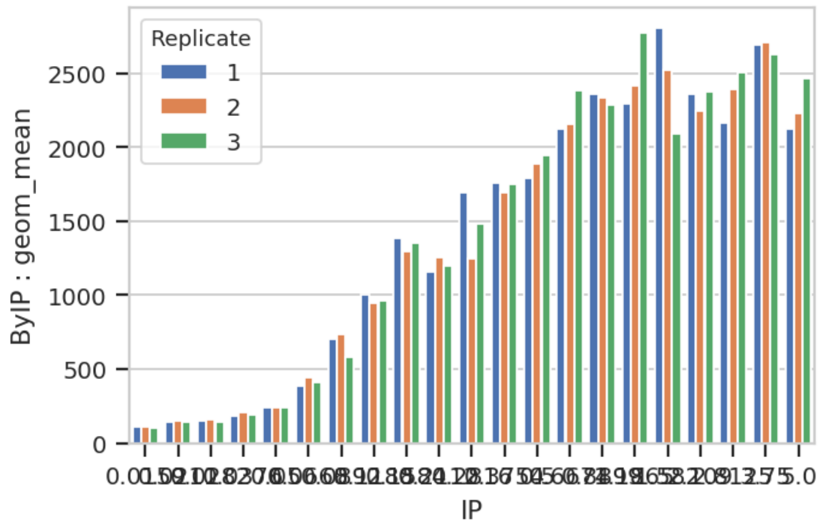
(continues on next page)

(continued from previous page)

```
~/src/cytoflow/cytoflow/views/base_views.py in plot(self, experiment, data, plot_name,␣
↪**kwargs)
    712                                             "plot variables or the plot name. "
    713                                             "Possible plot names: {}"
--> 714                                             .format(unused_names, list(groupby.
↪groups.keys())))
    715
    716             if plot_name not in set(groupby.groups.keys()):


CytoflowViewError: ('plot_name', "You must use facets ['Replicate'] in either the plot␣
↪variables or the plot name. Possible plot names: [1, 2, 3]")
```

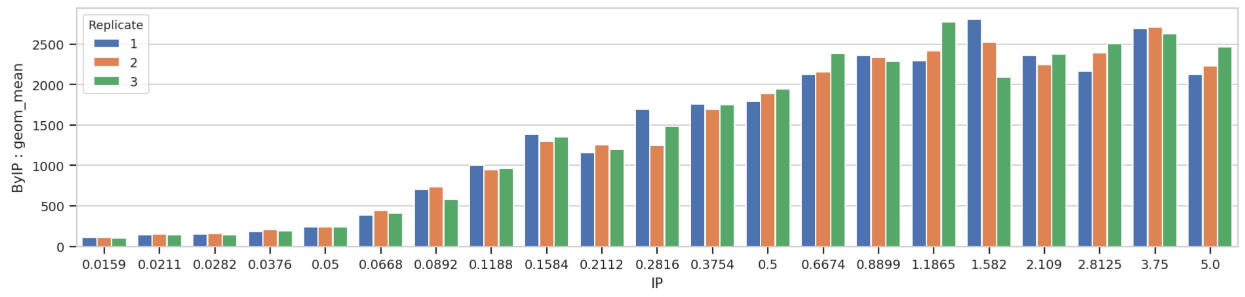
Oops! We forgot to specify how to plot the different `Replicates`. Each index of a statistic must be specified as either a variable or a facet of the plot, like so:

```
flow.BarChartView(statistic = ("ByIP", "geom_mean"),
                  variable = "IP",
                  huefacet = "Replicate").plot(ex2)
```



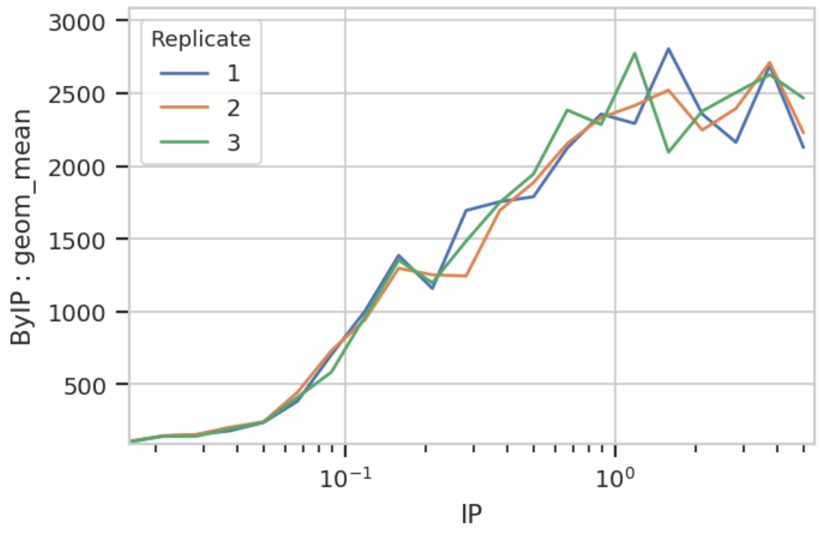
A quick aside - sometimes you get ugly axes because of overlapping labels. In this case, we want a wider plot. While we can directly specify the height of a plot, we can't directly specify its width, only its aspect ratio (the ratio of the width to the height.) `cytoflow` defaults to 1.5; in this case, let's widen it out to 4. If this results in a plot that's wider than your browser, the Jupyter notebook will scale it down for you.

```
flow.BarChartView(statistic = ("ByIP", "geom_mean"),
                  variable = "IP",
                  huefacet = "Replicate").plot(ex2, aspect = 4)
```



Bar charts are really best for categorical variables (with a modest number of categories.) Let's do a line chart instead:

```
flow.Stats1DView(statistic = ("ByIP", "geom_mean"),
                 variable = "IP",
                 variable_scale = 'log',
                 huefacet = "Replicate").plot(ex2)
```



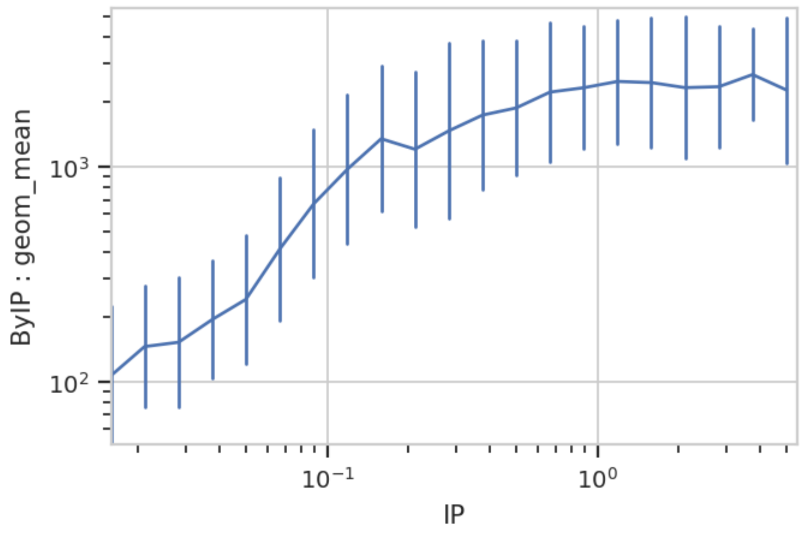
Statistics views can also plot error bars; the error bars must *also* be a statistic, and they must have the same indices as the "main" statistic. For example, let's plot the geometric mean and geometric standard deviation of each IP subset:

```
ex2 = flow.ChannelStatisticOp(name = "ByIP",
                              by = ["IP"],
                              channel = "FITC-A",
                              function = flow.geom_mean).apply(ex)

# While an arithmetic SD is usually plotted plus-or-minus the arithmetic mean,
# a *geometric* SD is usually plotted (on a log scale!) multiplied-or-divided by the
# geometric mean.  the function geom_sd_range is a convenience function that does this.

ex3 = flow.ChannelStatisticOp(name = "ByIP",
                              by = ['IP'],
                              channel = "FITC-A",
                              function = flow.geom_sd_range).apply(ex2)

flow.Stats1DView(statistic = ("ByIP", "geom_mean"),
```

(continues on next page)

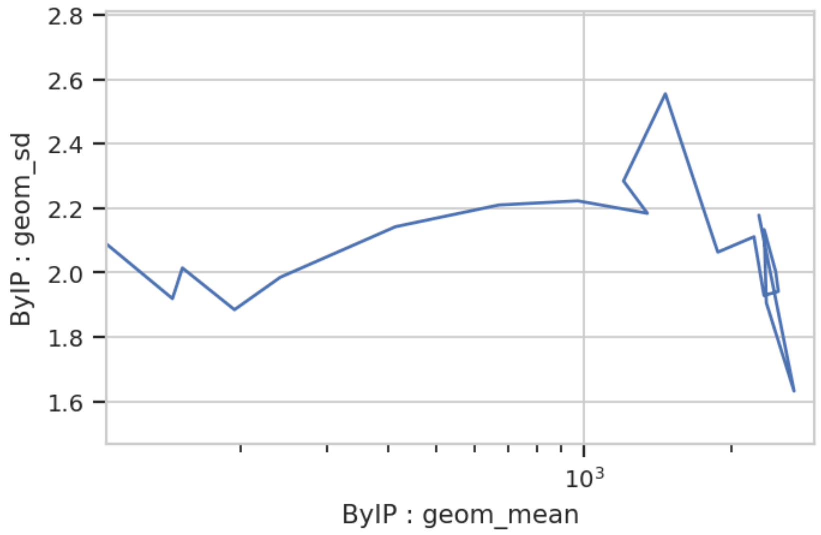
(continued from previous page)

```
                 variable = "IP",
                 variable_scale = "log",
                 scale = "log",
                 error_statistic = ("ByIP", "geom_sd_range")).plot(ex3)
```



The plot above shows how one statistic varies (on the Y axis) as a variable changes on the X axis. We can also plot two statistics against eachother. For example, we can ask if the geometric standard deviation varies as the geometric mean changes:

```
ex2 = flow.ChannelStatisticOp(name = "ByIP",
                              by = ["IP"],
                              channel = "FITC-A",
                              function = flow.geom_mean).apply(ex)

ex3 = flow.ChannelStatisticOp(name = "ByIP",
                              by = ['IP'],
                              channel = "FITC-A",
                              function = flow.geom_sd).apply(ex2)

flow.Stats2DView(variable = "IP",
                 xstatistic = ("ByIP", "geom_mean"),
                 ystatistic = ("ByIP", "geom_sd"),
                 xscale = "log").plot(ex3)
```



Nope, guess not. See the **TASBE Calibrated Flow Cytometry** notebook for more examples of 1D and 2D statistics views.

Transforming statistics

In addition to making statistics by applying summary functions to data, you can also apply functions to other statistics. For example, a common question is "What percentage of my events are in a particular gate?" We could, for instance, ask what percentage of events are above 1000 in the `FITC-A` channel, and how that varies by amount of `IP`. We start by defining a *threshold* gate with `ThresholdOp`:

```
thresh = flow.ThresholdOp(name = "Above1000",
                          channel = "FITC-A",
                          threshold = 1000)
ex2 = thresh.apply(ex)
```

Now, the `Experiment` has a condition named `Above1000` that is `True` or `False` depending on whether that event's `FITC-A` channel is greater than 1000. Next, we compute the total number of events in each subset with a unique combination of `Above1000` and `IP`:

```
ex3 = flow.ChannelStatisticOp(name = "Above1000",
                              by = ["Above1000", "IP"],
                              channel = "FITC-A",
                              function = len).apply(ex2)
ex3.statistics[("Above1000", "len")]
```

```
Above1000  IP
False      0.0159    299
           0.0211    299
           0.0282    299
           0.0376    296
           0.0500    299
           0.0668    262
```

(continues on next page)

(continued from previous page)

```
           0.0892    200
           0.1188    143
           0.1584     76
           0.2112    101
           0.2816     76
           0.3754     57
           0.5000     53
           0.6674     32
           0.8899     19
           1.1865     23
           1.5820     21
           2.1090     34
           2.8125     25
           3.7500     10
           5.0000     29
True       0.0159      1
           0.0211      1
           0.0282      1
           0.0376      4
           0.0500      1
           0.0668     38
           0.0892    100
           0.1188    157
           0.1584    224
           0.2112    199
           0.2816    224
           0.3754    243
           0.5000    247
           0.6674    268
           0.8899    281
           1.1865    277
           1.5820    279
           2.1090    266
           2.8125    275
           3.7500    290
           5.0000    271
Name: Above1000 : len, dtype: int64
```

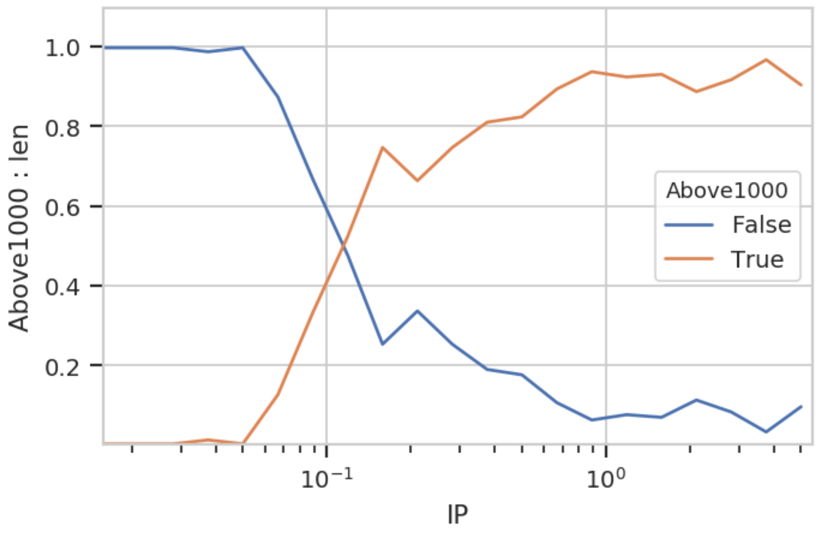
And now we compute the proportion of `Above1000 == True` for each value of `IP`. `TransformStatisticOp` applies a function to subsets of a statistic; in this case, we're applying a `lambda` function to convert each `IP` subset from length into proportion.

```
import pandas as pd

ex4 = flow.TransformStatisticOp(name = "Above1000",
                                statistic = ("Above1000", "len"),
                                by = ["IP"],
                                function = lambda a: pd.Series(a / a.sum()),
                                statistic_name = "proportion").apply(ex3)

ex4.statistics[("Above1000", "proportion")][0:8]
```

```
Above1000  IP
False      0.0159    0.996667
           0.0211    0.996667
           0.0282    0.996667
           0.0376    0.986667
           0.0500    0.996667
           0.0668    0.873333
           0.0892    0.666667
           0.1188    0.476667
Name: Above1000 : len, dtype: float64
```

Now we can plot the new statistic.

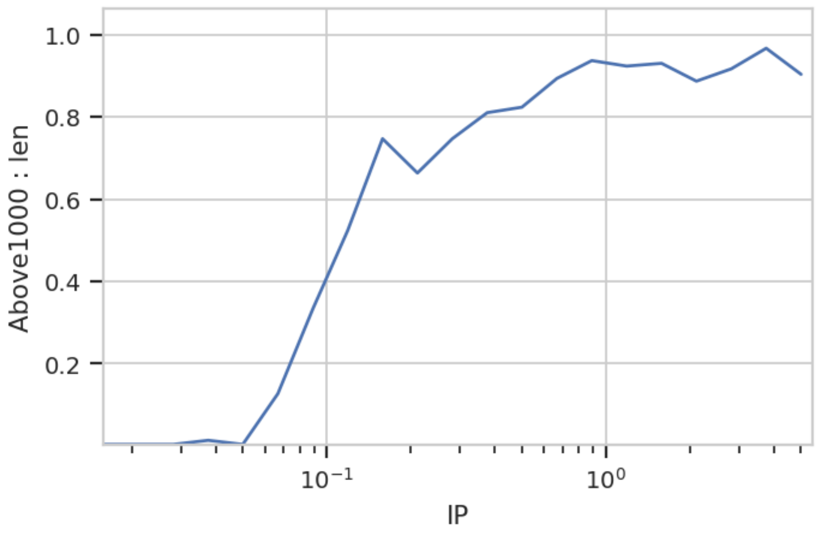
```
flow.Stats1DView(statistic = ("Above1000", "proportion"),
                 variable = "IP",
                 variable_scale = "log",
                 huefacet = "Above1000").plot(ex4)
```



Note that be default we get all the values in the statistic; in this case, it's both the proportion that's above 1000 (`True`) and the proportion that is below it (`False`). We can pass a `subset` attribute to the `Stats1DView` to look at just one or the other.

```
flow.Stats1DView(statistic = ("Above1000", "proportion"),
                 variable = "IP",
                 variable_scale = "log",
                 subset = "Above1000 == True").plot(ex4)
```

```
/home/brian/src/cytoflow/cytoflow/views/base_views.py:751: CytoflowViewWarning: Only one␣
↪value for level Above1000; dropping it.
```



Note the warning. Because the index level `Above1000` is always `True` in this subset, it gets dropped from the index. Note also that we don't have to specify it as a variable or facet anywhere.

Statistics from data-driven modules

One of the most exciting aspects of statistics in `cytoflow` is that other data-driven modules can add them to an `Experiment`, too. For example, the `GaussianMixtureOp` adds several statistics for each component of the mixture model it fits, containing the mean, standard deviation and proportion of observations in each component:

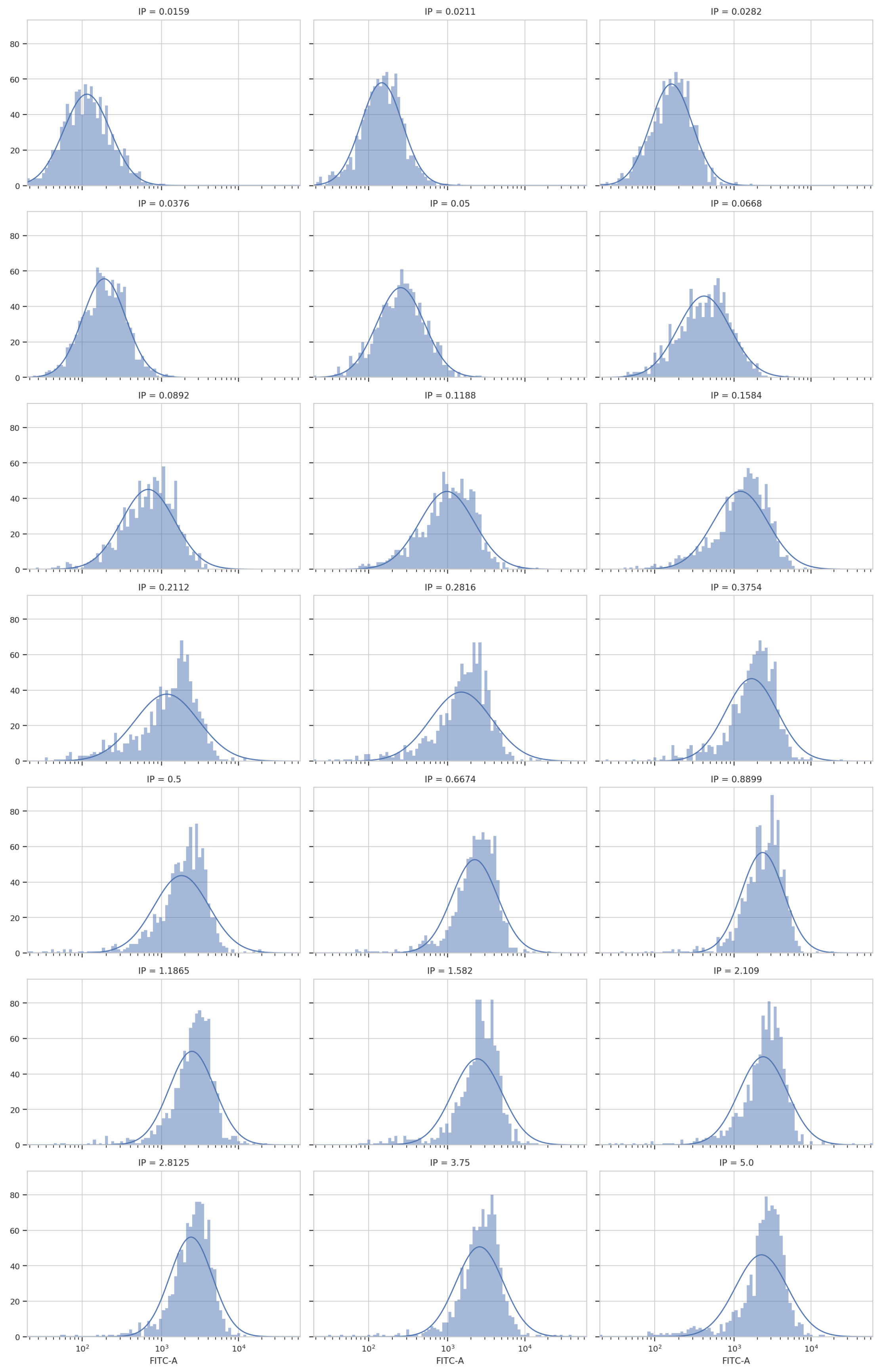
```
inputs = {
    "Yeast_B1_B01.fcs" : 5.0,
    "Yeast_B2_B02.fcs" : 3.75,
    "Yeast_B3_B03.fcs" : 2.8125,
    "Yeast_B4_B04.fcs" : 2.109,
    "Yeast_B5_B05.fcs" : 1.5820,
    "Yeast_B6_B06.fcs" : 1.1865,
    "Yeast_B7_B07.fcs" : 0.8899,
    "Yeast_B8_B08.fcs" : 0.6674,
    "Yeast_B9_B09.fcs" : 0.5,
    "Yeast_B10_B10.fcs" : 0.3754,
    "Yeast_B11_B11.fcs" : 0.2816,
    "Yeast_B12_B12.fcs" : 0.2112,
    "Yeast_C1_C01.fcs" : 0.1584,
    "Yeast_C2_C02.fcs" : 0.1188,
    "Yeast_C3_C03.fcs" : 0.0892,
    "Yeast_C4_C04.fcs" : 0.0668,
    "Yeast_C5_C05.fcs" : 0.05,
    "Yeast_C6_C06.fcs" : 0.0376,
    "Yeast_C7_C07.fcs" : 0.0282,
    "Yeast_C8_C08.fcs" : 0.0211,
    "Yeast_C9_C09.fcs" : 0.0159
}
```

(continues on next page)

(continued from previous page)

```
tubes = []
for filename, ip in inputs.items():
    tubes.append(flow.Tube(file = "data/" + filename, conditions = {'IP' : ip}))


ex = flow.ImportOp(conditions = {'IP' : "float"},
                   tubes = tubes).apply()
```

```
op = flow.GaussianMixtureOp(name = "Gauss",
                            channels = ["FITC-A"],
                            scale = {"FITC-A" : 'logicle'},
                            by = ["IP"],
                            num_components = 1)
op.estimate(ex)
ex2 = op.apply(ex)
op.default_view(xfacet = "IP").plot(ex2, col_wrap = 3)
```


IP = 0.0159
IP = 0.0211
IP = 0.0282
IP = 0.0376
IP = 0.05
IP = 0.0668
IP = 0.0892
IP = 0.1188
IP = 0.1584
IP = 0.2112
IP = 0.2816
IP = 0.3754
IP = 0.5
IP = 0.6674
IP = 0.8899
IP = 1.1865
IP = 1.582
IP = 2.109
IP = 2.8125
IP = 3.75
IP = 5.0
0
20
40
60
80
10^2
10^3
10^4
FITC-A

```
ex2.statistics.keys()
```

```
dict_keys([('Gauss', 'mean'), ('Gauss', 'sigma'), ('Gauss', 'interval')])
```

```
ex2.statistics[("Gauss", 'mean')]
```

```
IP      Component  Channel
0.0159  1          FITC-A      114.867593
0.0211  1          FITC-A      146.760865
0.0282  1          FITC-A      161.831942
0.0376  1          FITC-A      188.241811
0.0500  1          FITC-A      252.533895
0.0668  1          FITC-A      415.359435
0.0892  1          FITC-A      675.335896
0.1188  1          FITC-A      976.574632
0.1584  1          FITC-A     1212.127491
0.2112  1          FITC-A     1169.357113
0.2816  1          FITC-A     1491.938821
0.3754  1          FITC-A     1683.611714
0.5000  1          FITC-A     1812.128528
0.6674  1          FITC-A     2233.860489
0.8899  1          FITC-A     2341.076532
1.1865  1          FITC-A     2474.300525
1.5820  1          FITC-A     2401.710245
2.1090  1          FITC-A     2383.285932
2.8125  1          FITC-A     2411.122019
3.7500  1          FITC-A     2584.007182
5.0000  1          FITC-A     2265.732416
Name: Gauss : mean, dtype: float64
```

```
ex2.statistics[("Gauss", 'interval')]
```

```
IP      Component  Channel
0.0159  1          FITC-A       (56.94459866110697, 224.1783518669721)
0.0211  1          FITC-A       (80.36534657785384, 265.7842407147643)
0.0282  1          FITC-A       (88.27508462690886, 295.4218184072569)
0.0376  1          FITC-A      (101.18497543667104, 350.4182139054138)
0.0500  1          FITC-A     (127.95204222704295, 502.06564927968213)
0.0668  1          FITC-A       (195.2351392293184, 894.5712089072493)
0.0892  1          FITC-A     (310.61832832262314, 1484.6617729462246)
0.1188  1          FITC-A       (437.6150673574889, 2200.017570462604)
0.1584  1          FITC-A      (542.1024597233911, 2732.4410991530126)
0.2112  1          FITC-A      (459.2685949581729, 3011.3302773605374)
0.2816  1          FITC-A      (601.1140138003304, 3736.1069195737864)
0.3754  1          FITC-A      (784.0312198090705, 3635.8937650247603)
0.5000  1          FITC-A       (802.3248292365361, 4117.843046843494)
0.6674  1          FITC-A      (1134.2954460750368, 4414.715040960627)
0.8899  1          FITC-A      (1246.7333023687233, 4408.716470275269)
1.1865  1          FITC-A        (1256.858649306397, 4886.57964348989)
1.5820  1          FITC-A      (1151.7297619649949, 5027.751372108821)
2.1090  1          FITC-A       (1161.983752683322, 4906.469724482335)
```

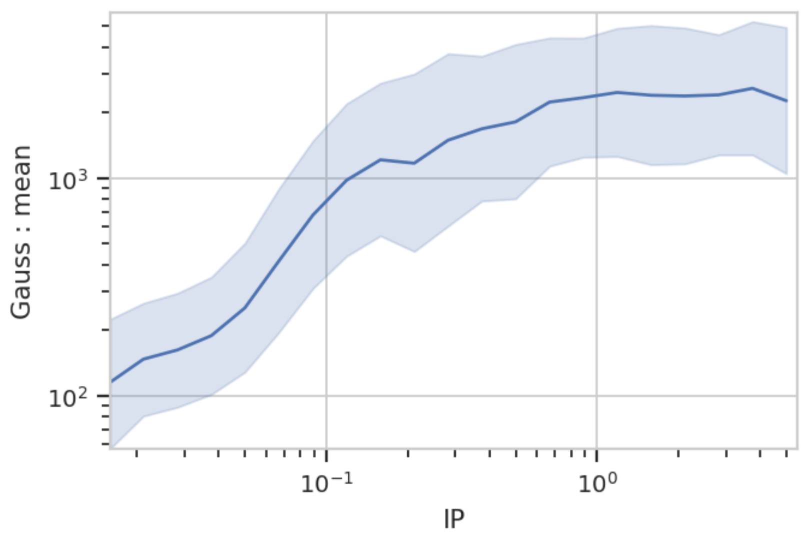
(continues on next page)

(continued from previous page)

```
2.8125  1          FITC-A     (1276.8361886540924, 4566.1348774856815)
3.7500  1          FITC-A      (1277.3600413588333, 5244.721547284582)
5.0000  1          FITC-A      (1046.8993940886585, 4925.741101423763)
Name: Gauss : interval, dtype: object
```

```
flow.Stats1DView(statistic = ("Gauss", "mean"),
                 variable = "IP",
                 variable_scale = "log",
                 scale = "log",
                 error_statistic = ("Gauss", "interval")).plot(ex2, shade_error = True)
```

```
/home/brian/src/cytoflow/cytoflow/views/base_views.py:751: CytoflowViewWarning: Only one␣
↪value for level Component; dropping it.
/home/brian/src/cytoflow/cytoflow/views/base_views.py:751: CytoflowViewWarning: Only one␣
↪value for level Channel; dropping it.
```



Yeast Dosage-Response Curve

This is data from a real experiment, measuring the dose response of an engineered yeast line to isopentyladenine, or IP. The yeast is engineered with a basic GFP reporter, and GFP fluorescence is measured after 12 hours, at which time we expect the cells to be at steady-state.

(Reference: Chen et al, Nature Biotech 2005)

Set up the Jupyter matplotlib integration, and import the cytoflow module.

```
# in some buggy versions of the Jupyter notebook, this needs to be in its OWN CELL.
%matplotlib inline
```

```
import cytoflow as flow
```

(continues on next page)

(continued from previous page)

```
# if your figures are too big or too small, you can scale them by changing matplotlib's␣
↪DPI
import matplotlib
matplotlib.rc('figure', dpi = 160)
```

First we set up the `Experiment`. This is primarily the mapping between the flow data (wells on a microtitre plate, here) and the metadata (experimental conditions, in this case the IP concentration.)

For this experiment, I did a serial dilution with three dilutions per log. Note that I specify that this metadata is of type `log`, which is represented internally as a `numpy` type `float`, but is plotted on a log scale when we go to visualize it.

Also note that, in order to keep the size of the source distribution manageable, I've only included 1000 events from each tube in the example files. The actual data set has 30,000 events from each, which gives a much prettier dose-response curve!

```
inputs = {
    "Yeast_B1_B01.fcs" : 5.0,
    "Yeast_B2_B02.fcs" : 3.75,
    "Yeast_B3_B03.fcs" : 2.8125,
    "Yeast_B4_B04.fcs" : 2.109,
    "Yeast_B5_B05.fcs" : 1.5820,
    "Yeast_B6_B06.fcs" : 1.1865,
    "Yeast_B7_B07.fcs" : 0.8899,
    "Yeast_B8_B08.fcs" : 0.6674,
    "Yeast_B9_B09.fcs" : 0.5,
    "Yeast_B10_B10.fcs" : 0.3754,
    "Yeast_B11_B11.fcs" : 0.2816,
    "Yeast_B12_B12.fcs" : 0.2112,
    "Yeast_C1_C01.fcs" : 0.1584,
    "Yeast_C2_C02.fcs" : 0.1188,
    "Yeast_C3_C03.fcs" : 0.0892,
    "Yeast_C4_C04.fcs" : 0.0668,
    "Yeast_C5_C05.fcs" : 0.05,
    "Yeast_C6_C06.fcs" : 0.0376,
    "Yeast_C7_C07.fcs" : 0.0282,
    "Yeast_C8_C08.fcs" : 0.0211,
    "Yeast_C9_C09.fcs" : 0.0159
}

tubes = []
for filename, ip in inputs.items():
    tubes.append(flow.Tube(file = "data/" + filename, conditions = {'IP' : ip}))

ex = flow.ImportOp(conditions = {'IP' : "float"},
                   tubes = tubes).apply()
```

Set `logicle` as the default plot scale:

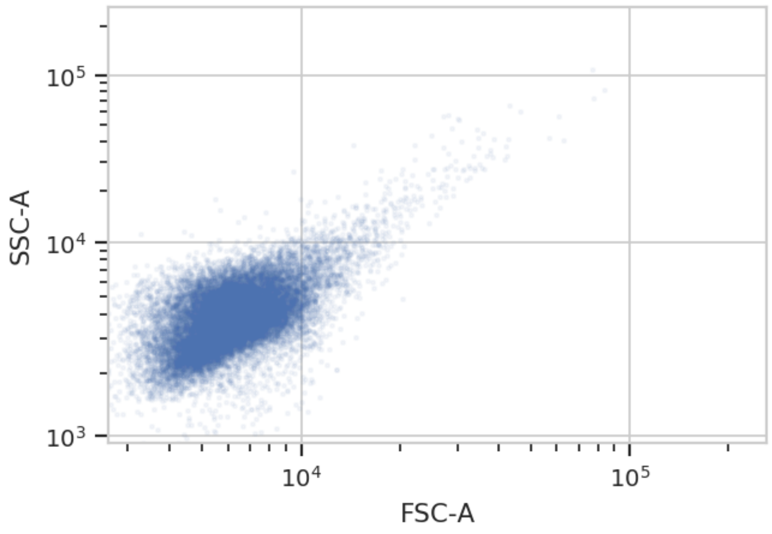
```
flow.set_default_scale("logicle")
```

The first step in most cytometry workflows is to filter out debris and aggregates based on morphological parameters (the forward- and side-scatter measurements.) So let's have a look at the FSC-A and SSC-A channels.

Remember, because we're not specifying otherwise, this is a plot of *all of the data* in the experiment, not a single tube!

```
flow.ScatterplotView(xchannel = "FSC-A",
                     ychannel = "SSC-A").plot(ex, alpha = 0.05)
```

```
/home/brian/src/cytoflow/cytoflow/utility/logicle_scale.py:307: CytoflowWarning: Channel␣
↪FSC-A doesn't have any negative data. Try a log transform instead.
```

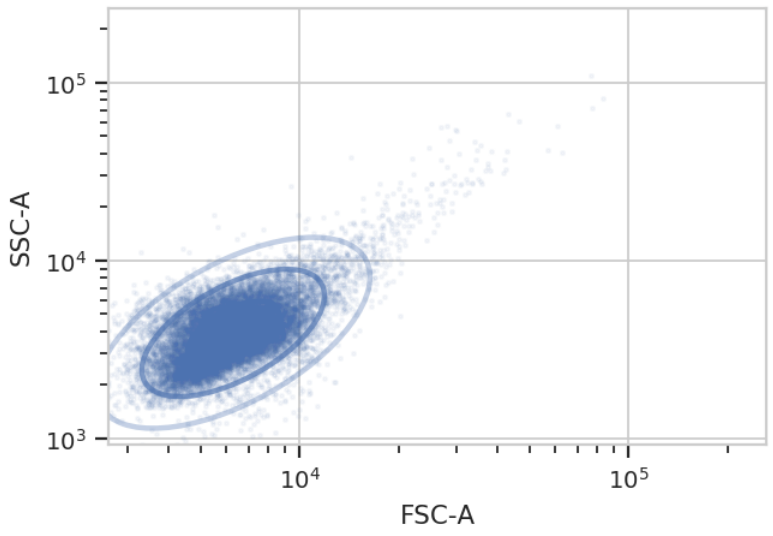


Oops! The biexponential scale is really intended for channels that have events around 0, with some negative events; let's use a log scale on both FSC-A and SSC-A in the future. (The fluorescence channels should be fine on a `logicle` scale, though.)

Because this is yeast growing in log phase on a drum roller, the distributions in FSC-A and SSC-A are pretty tight. Let's fit a 2D gaussian to it, and gate out any data that is more than three standard deviations from the mean.

```
g = flow.GaussianMixtureOp(name = "Debris_Filter",
                           channels = ["FSC-A", "SSC-A"],
                           scale = {'FSC-A' : 'log',
                                    'SSC-A' : 'log'},
                           num_components = 1,
                           sigma = 3)
g.estimate(ex)
g.default_view().plot(ex, alpha = 0.05)

ex2 = g.apply(ex)
```

```
/home/brian/src/cytoflow/cytoflow/operations/gaussian.py:528: RuntimeWarning: invalid␣
↪value encountered in less_equal
```

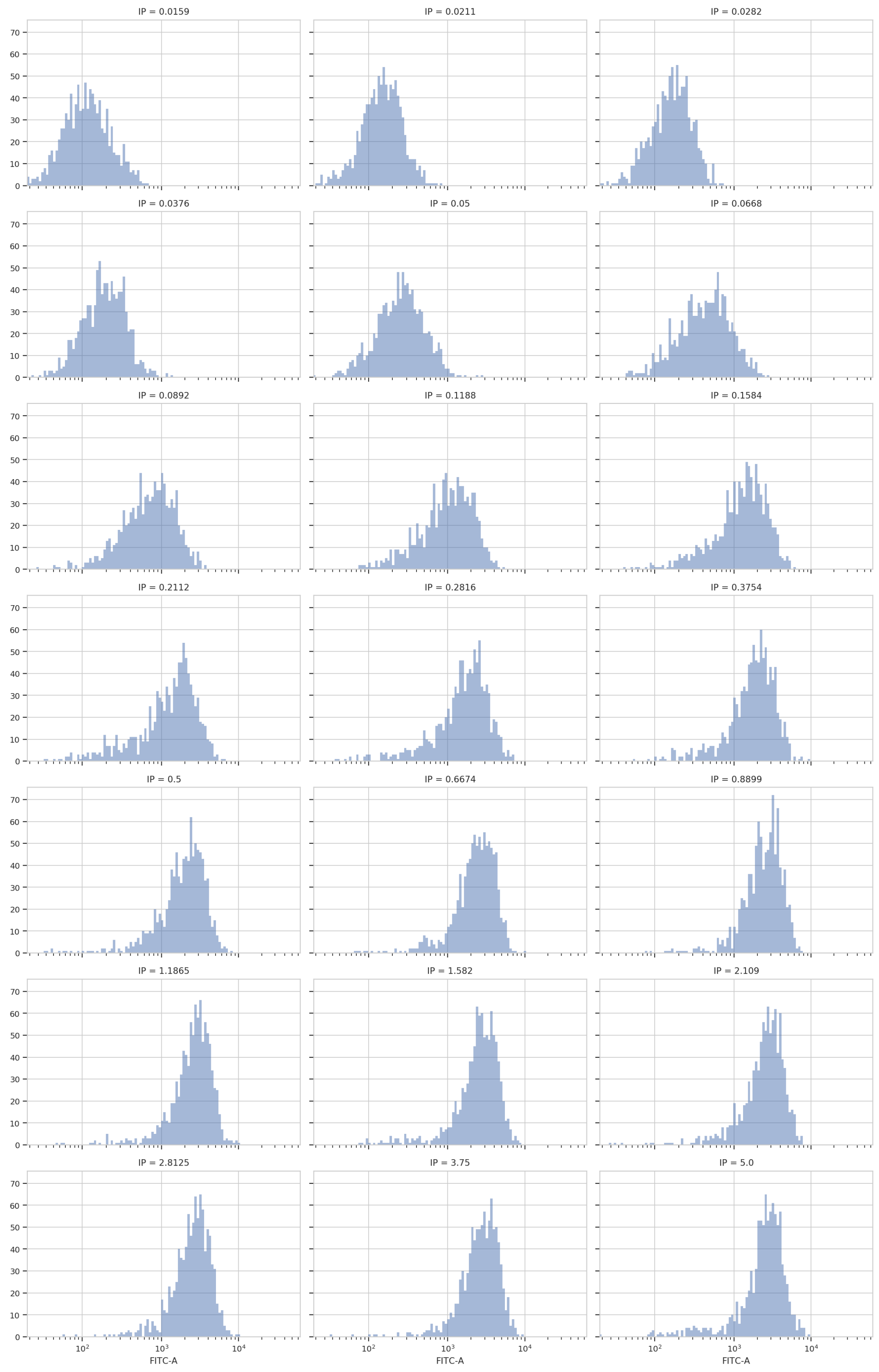


One thing to note: if you've read the `Machine Learning` notebook, you know that a `GaussianMixtureOp` adds categorical metadata as well. In this case, we have a new boolean column named `Debris_Filter_1`, which is `True` if the event is less than `sigma` from the centroid and `False` otherwise.

```
ex2.data.head()
```

Now, let's look at the FITC-A channel, which is the one we're interested in. First of all, just check histograms of the FITC-A distribution in each tube. Make sure to set the `subset` trait to only include the events that aren't debris!

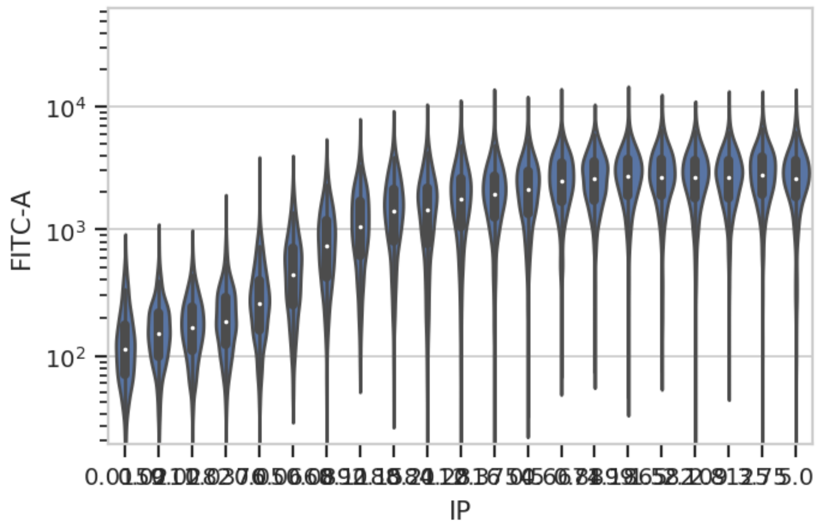
```
flow.HistogramView(channel = "FITC-A",
                   xfacet = "IP",
                   subset = "Debris_Filter_1 == True").plot(ex2, col_wrap = 3)
```



IP = 0.0159
IP = 0.0211
IP = 0.0282
IP = 0.0376
IP = 0.05
IP = 0.0668
IP = 0.0892
IP = 0.1188
IP = 0.1584
IP = 0.2112
IP = 0.2816
IP = 0.3754
IP = 0.5
IP = 0.6674
IP = 0.8899
IP = 1.1865
IP = 1.582
IP = 2.109
IP = 2.8125
IP = 3.75
IP = 5.0
0
10
20
30
40
50
60
70
10^2
10^3
10^4
FITC-A

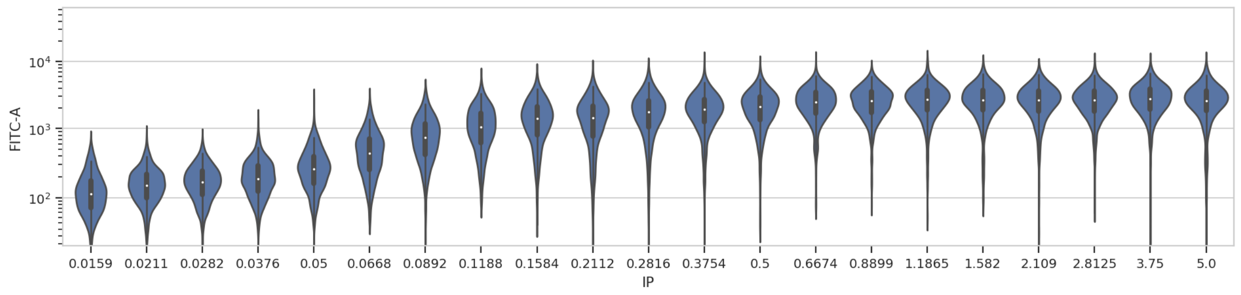
Sure enough, looking at the plots, it's clear that the histogram peak moves to the right as the IP concentration increases. We can get a more compact representation of the same data with one of `cytoflow`'s fancier plots, a violin plot:

```
flow.ViolinPlotView(variable = "IP",
                    channel = "FITC-A",
                    scale = "logicle",
                    subset = "Debris_Filter_1 == True").plot(ex2)
```



A quick aside - sometimes you get ugly axes because of overlapping labels. In this case, we want a wider plot. While we can directly specify the height of a plot, we can't directly specify its width, only its aspect ratio (the ratio of the width to the height.) `cytoflow` defaults to 1.5; in this case, let's widen it out to 4. If this results in a plot that's wider than your browser, the Jupyter notebook will scale it down for you.

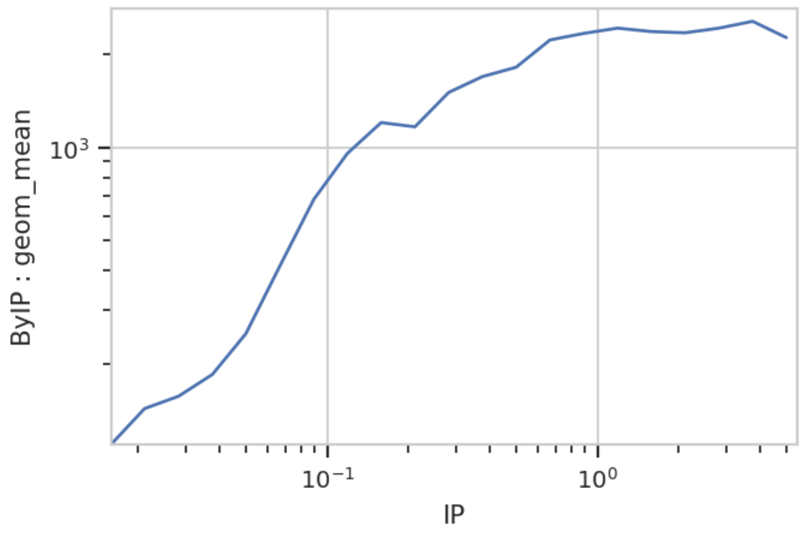
```
flow.ViolinPlotView(variable = "IP",
                    channel = "FITC-A",
                    scale = "logicle",
                    subset = "Debris_Filter_1 == True").plot(ex2, aspect = 4)
```



At least on my screen that's better, but a violin plot is really best for categorical data (with a modest number of categories.) Instead, let's make a true does-response curve. The `ChannelStatisticsOp` divides the data into subsets by different values of the `IP` variable and applies `flow.geom_mean` to each subset. Then, the `Stats1DView` plots that statistic.

(Confused? Have a look at the `Statistics` example notebook for a more in-depth discussion.)

```
ex3 = flow.ChannelStatisticOp(name = "ByIP",
                              channel = "FITC-A",
                              by = ["IP"],
                              function = flow.geom_mean,
                              subset = "Debris_Filter_1 == True").apply(ex2)

flow.Stats1DView(statistic = ("ByIP", "geom_mean"),
                 scale = "log",
                 variable = "IP",
                 variable_scale = "log").plot(ex3)
```



And of course, because the underlying data is just a `pandas.DataFrame`, we can manipulate it with the rest of the Scientific Python ecosystem. Here, we get the compute the geometric means using `pandas.DataFrame.groupby`.

```
(ex2.data.query("Debris_Filter_1 == True")[['IP', 'FITC-A']]
         .groupby('IP')
         .agg(flow.geom_mean))
```

Examples

These examples illustrate some of the advanced capabilities of the *cytoflow* library. They are generated from the example notebooks and data sets in the `cytoflow-$VERSION-examples-advanced.zip` file, available from the `Releases` tab at the project homepage. Feel free to fire up a Jupyter notebook and follow along!

Yeast induction timecourse

This notebook demonstrates a real-world multidimensional analysis example. The yeast strain responds to the small molecule isopentyladenine (IP) by expressing green fluorescent protein (GFP), which we measure using a flow cytometer in the FITC-A channel.

This experiment was designed to determine the dynamics of the IP –> GFP response. So, we induced several yeast cultures with different amounts of IP, then took readings on the cytometer every 30 minutes for 8 hours. The outline of the experimental setup is below.

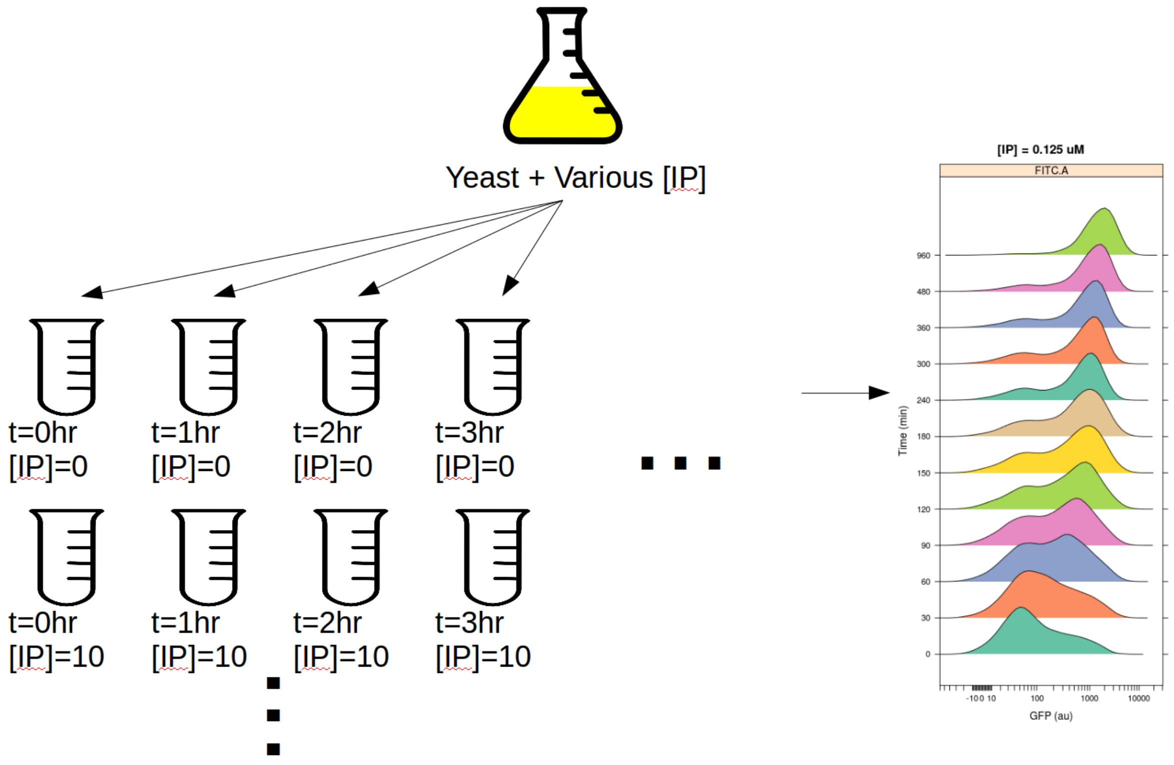


Fig. 3: Induction experiment

Setup the notebook's matplotlib integration, and import `cytoflow`.

```
# in some buggy versions of the Jupyter notebook, this needs to be in its OWN CELL.
%matplotlib inline
```

```
import cytoflow as flow

# if your figures are too big or too small, you can scale them by changing matplotlib's␣
↪DPI
import matplotlib
matplotlib.rc('figure', dpi = 160)
```

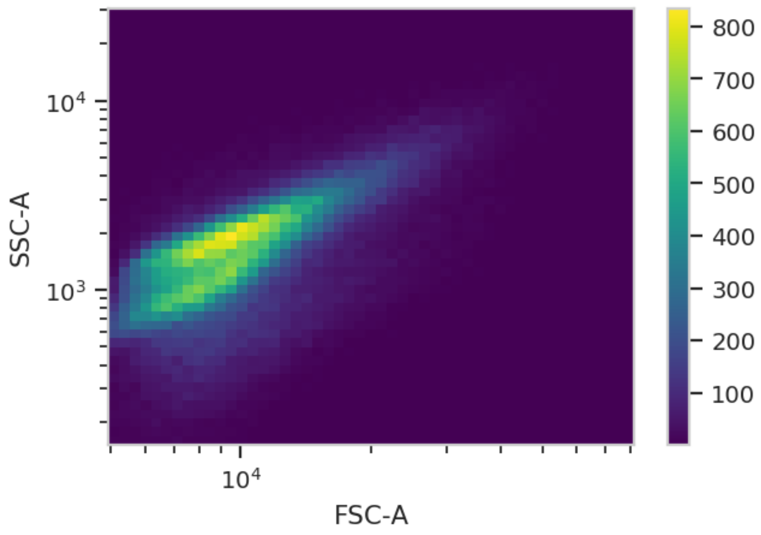
In this instance, the amount of IP and the time is actually encoded in the FCS filename. So, use `glob` to iterate through all the FCS files in the directory and `re` extract the IP concentration and timepoint from the filename.

NB. Many FCS files already have a channel named “Time” which encodes how long since the acquisition start an event was collected. So it is inadvisable to use “Time” as a condition name.

```
# In this instance, I have encoded the experimental conditions in the filenames.
# So, use glob to get the files and parse the conditions back out.

import glob, re
tubes = []

for f in glob.glob("*.fcs"):
    r = re.search("IP_(.*?)_Minutes_(.*?)\.fcs", f)
    ip = r.group(1)
    minutes = r.group(2)

    tube = flow.Tube(file = f, conditions = {"IP" : float(ip), "Minutes" : int(minutes)})
    tubes.append(tube)

ex = flow.ImportOp(tubes = tubes,
                   conditions = {"IP" : "float",
                                 "Minutes" : "int"},
                   events = 1000).apply()
```

Take a quick look at the morphological parameters.

```
flow.DensityView(xchannel = "FSC-A",
                 xscale = "log",
                 ychannel = "SSC-A",
                 yscale = "log").plot(ex)
```



There does seem to be a little bit of structure here, but in general the distribution is quite tight. So, we’ll use a 2D gaussian mixture model to get single cells, keeping the events that are within two standard deviations of the distribution mean.

```
gm = flow.GaussianMixtureOp(name = "GM",
                            channels = ['FSC-A', 'SSC-A'],
                            scale = {'FSC-A' : 'log',
```

(continues on next page)

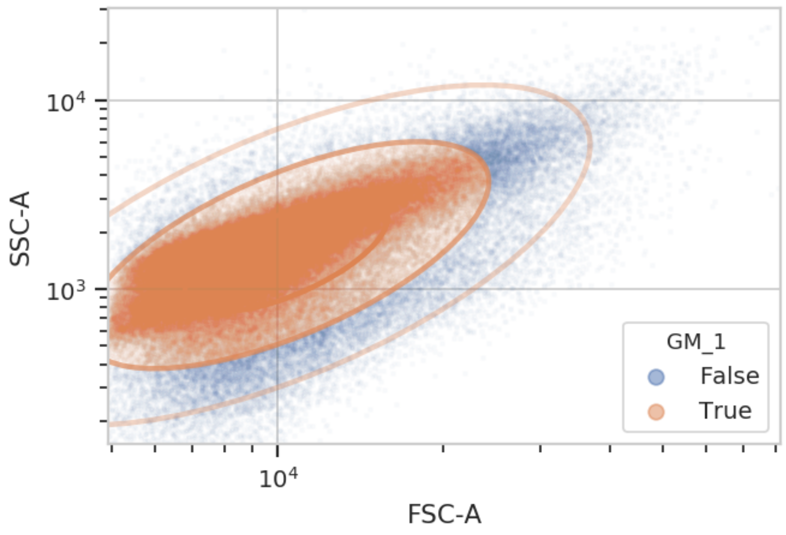
(continued from previous page)

```
                                             'SSC-A' : 'log'},
                                  num_components = 1,
                                  sigma = 2)
gm.estimate(ex)
ex_gm = gm.apply(ex)
```

A diagnostic plot of the GMM.

```
gm.default_view().plot(ex_gm, alpha = 0.02)
```

```
/home/brian/src/cytoflow/cytoflow/operations/base_op_views.py:341: CytoflowViewWarning:␣
↪Setting 'huefacet' to 'GM_1'
```

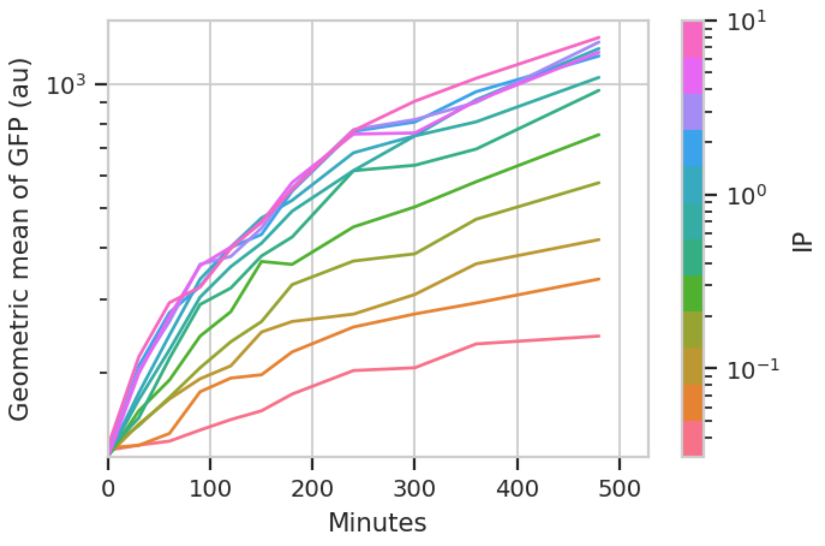


Yep, that looks fine. Now compute the geometric mean in the FITC-A channel to see how GFP expression varies with IP concentration and time since induction.

```
ex_stat = flow.ChannelStatisticOp(name = 'GFP',
                                  channel = 'FITC-A',
                                  function = flow.geom_mean,
                                  by = ['IP', 'Minutes'],
                                  subset = 'GM_1 == True').apply(ex_gm)
```

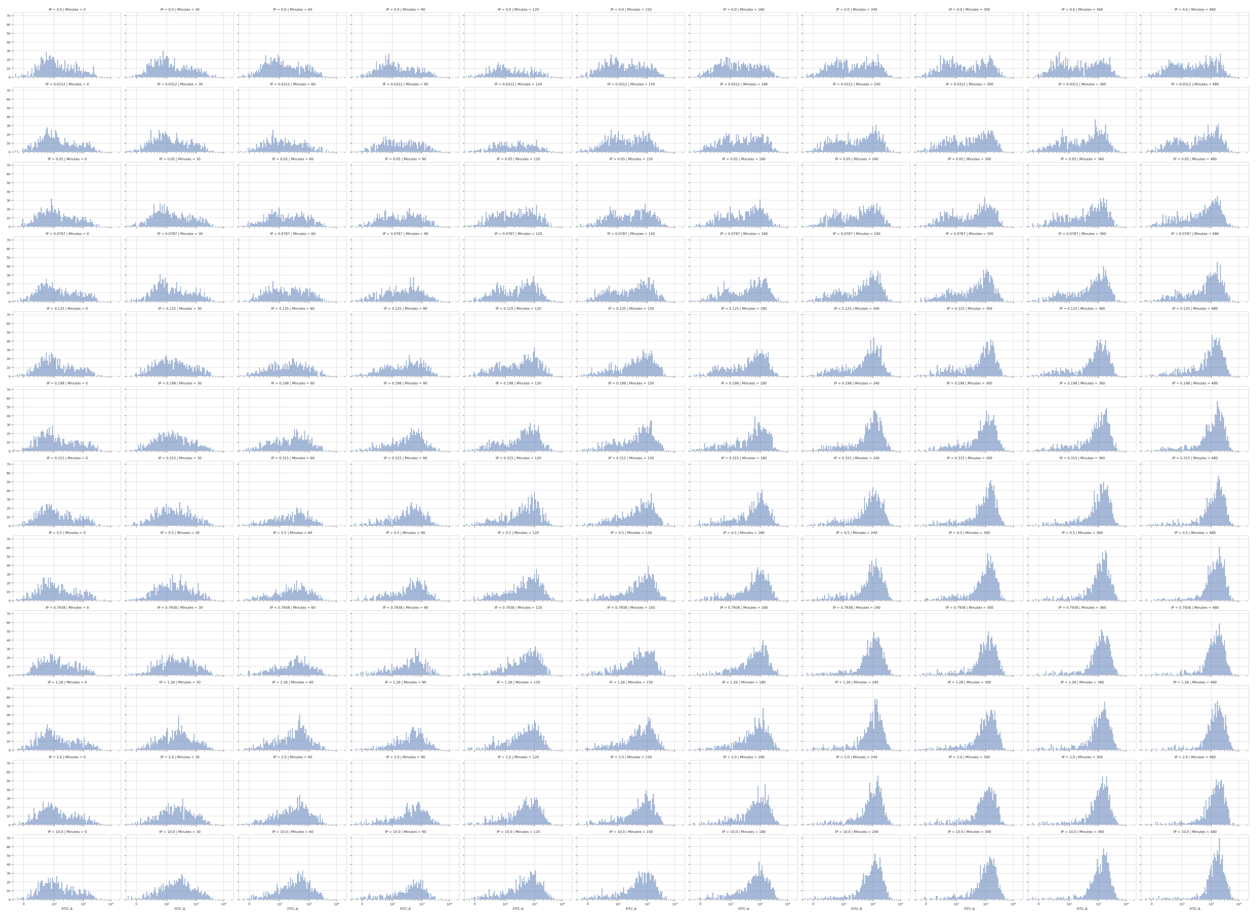
And plot it. Could you use pandas and seaborn to do this instead? Absolutely.

```
flow.Stats1DView(statistic = ('GFP', 'geom_mean'),
                 variable = 'Minutes',
                 scale = 'log',
                 huefacet = 'IP',
                 huescale = 'log').plot(ex_stat,
                                        ylabel = 'Geometric mean of GFP (au)')
```



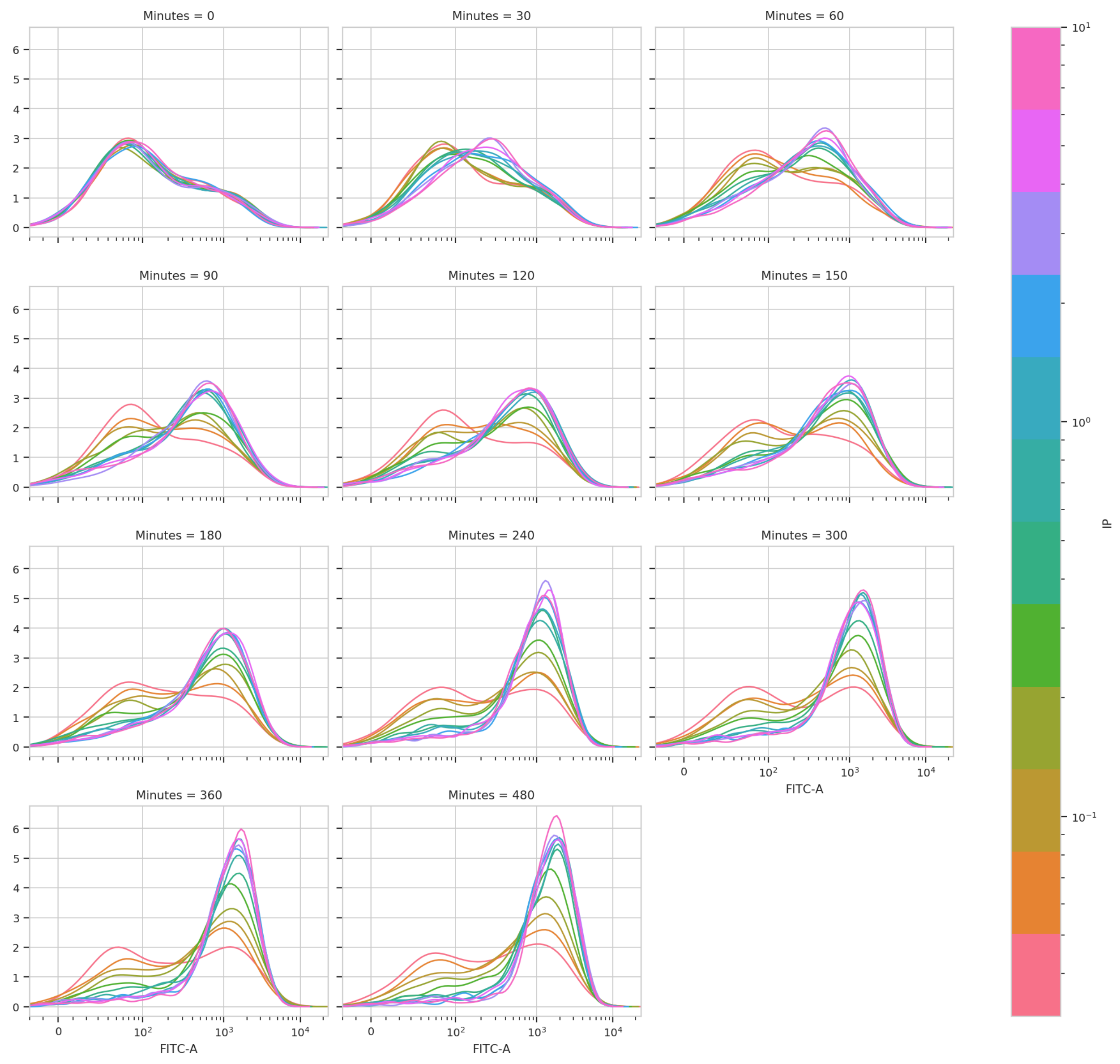
A geometric mean is only an appropriate summary statistic if the unimodal in log space. Is this actually true? Let's look at the histogram of each [IP]/time combination to find out.

```
flow.HistogramView(channel = 'FITC-A',
                   xfacet = 'Minutes',
                   yfacet = 'IP',
                   scale = 'logicle',
                   subset = 'GM_1 == True').plot(ex_gm)
```



Wow! That's a lot of plots, and reading the axes is impossible. I maybe could use `matplotlib.pyplot` to change the plot parameters and get something useful, but instead let's put [IP] on the hue facet instead of the Y facet. Then, we can wrap the X facet into three columns and actually see each plot.

```
flow.Kde1DView(channel = 'FITC-A',
               xfacet = 'Minutes',
               huefacet = 'IP',
               scale = 'logicle',
               huescale = 'log',
               subset = 'GM_1 == True').plot(ex_gm, col_wrap = 3, shade = False)
```



This is very, very interesting! There appears to be significant structure to this data. It's almost as if there are two populations, one that is "off" and one that is "on" – and that higher [IP] influences the rate at which cells switch from the off population to the on population.

We can model this mixture of gaussians using a class we've already seen, `GaussianMixtureOp`. We'll estimate two components, and we won't specify `sigma`. We'll also say `by = ['IP', 'Minutes']` to fit a different model to each unique combination of [IP] and time.

```
gm_fitc = flow.GaussianMixtureOp(name = "GM_FITC",
                                 channels = ['FITC-A'],
                                 scale = {'FITC-A' : 'log'},
                                 by = ['IP', 'Minutes'],
                                 num_components = 2)
gm_fitc.estimate(ex_gm, subset = 'GM_1 == True')
```

```
ex_stat_2 = gm_fitc.apply(ex_gm)
```

Most data-driven operations add summary statistics to the experiment as well. Let's have a look at which statistics are defined for this experiment.

```
ex_stat_2.statistics.keys()
```

```
dict_keys([('GM', 'mean'), ('GM', 'sigma'), ('GM', 'interval'), ('GM', 'correlation'), (
↪'GM_FITC', 'mean'), ('GM_FITC', 'sigma'), ('GM_FITC', 'interval'), ('GM_FITC',
↪'proportion')])
```

The statistic `('GM_FITC', 'proportion')` looks promising.

```
ex_stat_2.statistics[('GM_FITC', 'proportion')]
```

```
IP       Minutes  Component
0.0000   0        1             0.689475
                  2             0.310525
         30       1             0.689704
                  2             0.310296
         60       1             0.667829
                  2             0.332171
         90       1             0.677261
                  2             0.322739
         120      1             0.651504
                  2             0.348496
         150      1             0.573523
                  2             0.426477
         180      1             0.585170
                  2             0.414830
         240      1             0.576592
                  2             0.423408
         300      1             0.589542
                  2             0.410458
         360      1             0.531817
                  2             0.468183
         480      1             0.527385
                  2             0.472615
0.0312   0        1             0.763097
                  2             0.236903
         30       1             0.648558
                  2             0.351442
         60       1             0.625895
                  2             0.374105
         90       1             0.585238
                  2             0.414762
                                  ...
2.0000   240      1             0.189016
                  2             0.810984
         300      1             0.155909
                  2             0.844091
         360      1             0.152130
                  2             0.847870
```

(continues on next page)

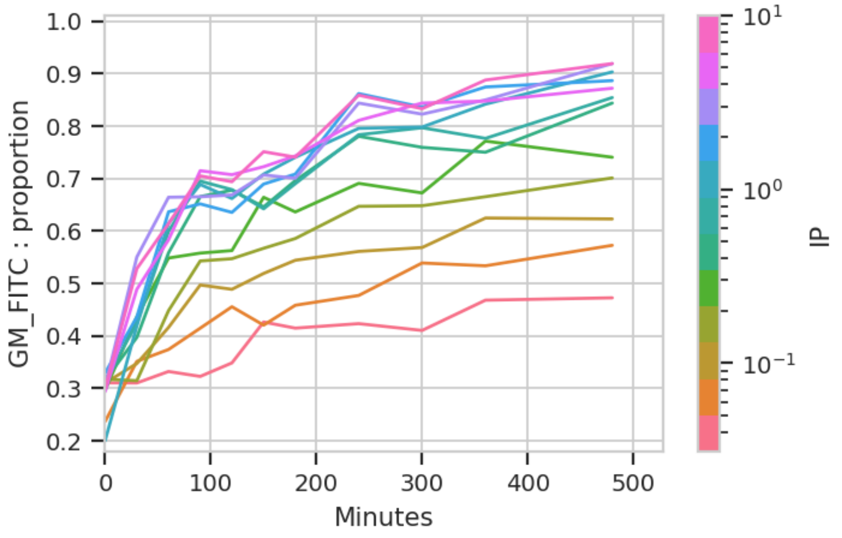
(continued from previous page)

```
         480      1             0.127653
                  2             0.872347
10.0000  0        1             0.699984
                  2             0.300016
         30       1             0.471701
                  2             0.528299
         60       1             0.385929
                  2             0.614071
         90       1             0.294833
                  2             0.705167
         120      1             0.305929
                  2             0.694071
         150      1             0.248707
                  2             0.751293
         180      1             0.259084
                  2             0.740916
         240      1             0.140860
                  2             0.859140
         300      1             0.166703
                  2             0.833297
         360      1             0.112013
                  2             0.887987
         480      1             0.080634
                  2             0.919366
Name: GM_FITC : proportion, Length: 264, dtype: float64
```

Great, we've got the proportions of component 1 and 2, at each IP concentration and timepoint. Plot the proportion in component 2 with "Minutes" on the X axis and "IP" on the hue facet.

```
flow.Stats1DView(statistic = ("GM_FITC", "proportion"),
                 variable = "Minutes",
                 huefacet = "IP",
                 huescale = "log",
                 subset = "Component == 2").plot(ex_stat_2)
```

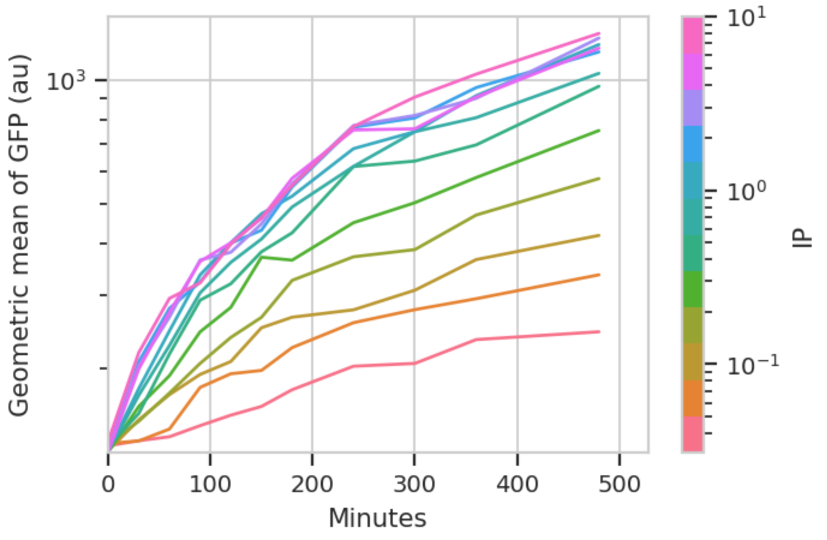
```
/home/brian/src/cytoflow/cytoflow/views/base_views.py:751: CytoflowViewWarning: Only one␣
↪value for level Component; dropping it.
```



Ignore the "jagged" nature of the plot. The original data set is hundreds of megabytes big, and in that data set the curves are much smoother (-;

The important question is: is this any different than the geometric mean? Let's re-plot the geometric mean plot so we can look at both.

```
flow.Stats1DView(statistic = ('GFP', 'geom_mean'),
                 variable = 'Minutes',
                 scale = 'log',
                 huefacet = 'IP',
                 huescale = 'log').plot(ex_stat,
                                        ylabel = 'Geometric mean of GFP (au)')
```



I think those dynamics look significantly different. For one thing, the mixture model "saturates" much more quickly – both in time and in [IP]. The geometric mean model indicates saturation at about 5 uM, while the mixture model seems to saturate one or two steps earlier. Things also stop changing quite as dramatically by about 240 minutes, whereas the geometric mean hasn't reached anything like a steady state by 480 minutes (the end of the experiment.)

I hope this has demonstrated a non-trivial insight into the dynamics of this biological system that are gained by looking at it through a quantitative lens, with some machine learning thrown in there as well.

Kiani et al, Nature Methods 2014

This example reproduces Figure 2, part (a), from Kiani et al, Nature Methods 11: 723 (2014).

In order to keep download sizes reasonable, the files only have about 10% of the original events.

Experimental Layout

This experiment uses a dCas9 fusion to repress the output of a yellow fluorescent reporter. The dCas9 is directed to the repressible promoter by a guide RNA under the control of rtTA3, a transcriptional activator controlled with the small molecule inducer doxycycline (Dox).

The plasmids that were co-transfected are shown below (reproduced from the above publication's Supplementary Figure 6a.)

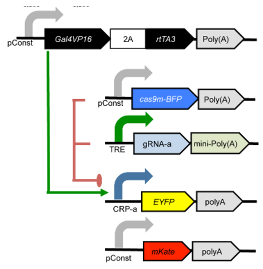


Fig. 4: Genetic circuit

The experiment compares three different conditions: - No dCas9 / no gRNA - No gRNA - The full gene network

Each condition was induced with 0, 100, 500, and 4000 uM Dox.

```
# change "inline" --> "notebook" if you want to interact with the plots.
# in some buggy versions of the Jupyter notebook, this needs to be in its OWN CELL.

%matplotlib inline
```

```
import cytoflow as flow

# if your figures are too big or too small, you can scale them by changing matplotlib's␣
↪DPI
import matplotlib
matplotlib.rc('figure', dpi = 160)
```

As is usual with `cytoflow`, we start by mapping the files to the experimental conditions. We have three different conditions; four different Dox concentrations; and three replicates.

```
# (Condition, [Dox]) --> filename
inputs = {
    ("Full_Circuit", 0.0) : "Specimen_002_A1_A01.fcs",
    ("Full_Circuit", 100.0) : "Specimen_002_A2_A02.fcs",
```

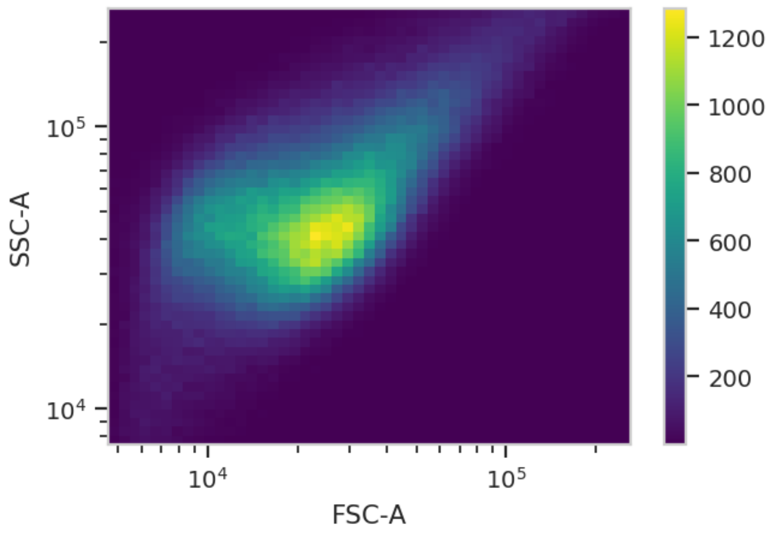
(continues on next page)

(continued from previous page)

```
    ("Full_Circuit", 500.0) : "Specimen_002_A3_A03.fcs",
    ("Full_Circuit", 4000.0) : "Specimen_002_A4_A04.fcs",
    ("No_gRNA", 0.0) : "Specimen_002_C5_C05.fcs",
    ("No_gRNA", 100.0) : "Specimen_002_C6_C06.fcs",
    ("No_gRNA", 500.0) : "Specimen_002_C7_C07.fcs",
    ("No_gRNA", 4000.0) : "Specimen_002_C8_C08.fcs",
    ("No_Cas9", 0.0) : "Specimen_002_D1_D01.fcs",
    ("No_Cas9", 100.0) : "Specimen_002_D2_D02.fcs",
    ("No_Cas9", 500.0) : "Specimen_002_D3_D03.fcs",
    ("No_Cas9", 4000.0) : "Specimen_002_D4_D04.fcs"}

tubes = []

for repl, path in {1 : "repl1", 2 : "repl2", 3 : "repl3"}.items():
    for (condition, dox), filename in inputs.items():
        tube = flow.Tube(file = path + "/" + filename,
                         conditions = {'Condition' : condition,
                                       'Dox' : dox,
                                       'Replicate' : repl})
        tubes.append(tube)

import_op = flow.ImportOp(conditions = {'Condition' : "category",
                                        'Dox' : "float",
                                        'Replicate' : "int"},
                          tubes = tubes)

ex = import_op.apply()
```

Have a look at the morphological parameters.

```
flow.DensityView(xchannel = "FSC-A",
                 ychannel = "SSC-A",
                 xscale = 'log',
                 yscale = 'log').plot(ex, min_quantile = 0.005)
```



This looks like it's been pre-gated (ie, there's not a mixture of populations.) It's also pushed up against the top axes in both SSC-A and FSC-A, which is a little concerning.

Let's see how many events we're getting piled up.

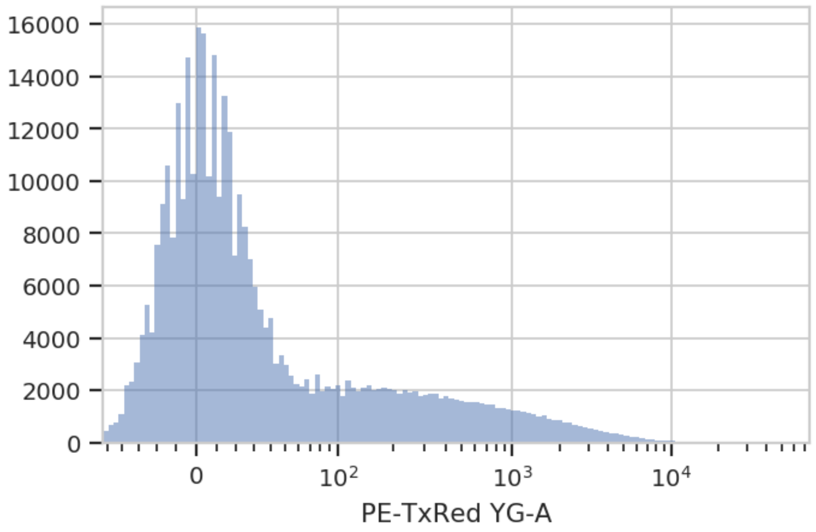
```
(ex['SSC-A'] == ex['SSC-A'].max()).sum()
```

```
5434
```

Alright. Some events (~5,000 out of 360,000) but not enough to be concerning.

The next thing we usually do is select for positively transfected cells. mKate is the transfection marker, so look at the red channel.

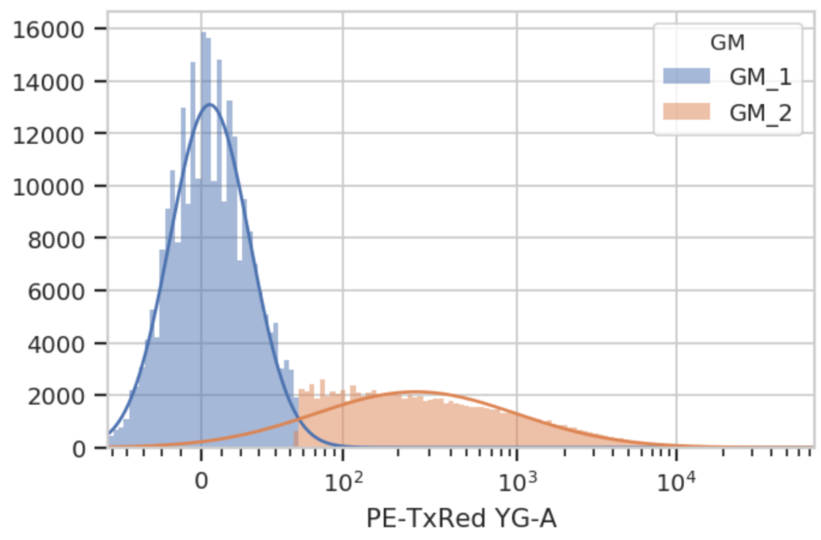
```
flow.HistogramView(channel = "PE-TxRed YG-A",
                   scale = 'logicle').plot(ex)
```



Let's fit a mixture-of-gaussians, for a nice principled way of separating the transfected population from the untransfected population.

```
gm = flow.GaussianMixtureOp(name = "GM",
                            channels = ["PE-TxRed YG-A"],
                            scale = {'PE-TxRed YG-A' : 'logicle'},
                            num_components = 2)
gm.estimate(ex)
ex_gm = gm.apply(ex)
gm.default_view().plot(ex_gm)
```

```
/home/brian/src/cytoflow/cytoflow/operations/base_op_views.py:341: CytoflowViewWarning:␣
↪Setting 'huefacet' to 'GM'
```



Looks good: the events with `GM == 'GM_2'` are the cells in the transfected population. Let's see how many events were in each population.

```
ex_gm.data.groupby(['GM']).size()
```

```
GM
GM_1    262010
GM_2     97990
dtype: int64
```

Let's make a new condition, `Transfected`, to be a boolean value with whether the cell was in `GM_2` or not. **Note: ``add_condition`` acts on the ``Experiment`` in-place.**

```
ex_gm.add_condition(name = 'Transfected',
                    dtype = 'bool',
                    data = (ex_gm['GM'] == 'GM_2'))
```

Now, we can reproduce the bar chart in the publication by taking the output (EYFP, in the `FITC-A` channel) geometric mean of the positively transfected cells, split out by condition and [Dox]. Don't forget to look at just the *transfected* cells (using `subset`). We'll compute the geometric mean across circuit and [Dox], and then split it out by replicate so we can compute an SEM.

Please note: This is a **terrible** place to use error bars. See:

https://www.nature.com/nature/journal/v492/n7428/full/492180a.html

and

http://jcb.rupress.org/content/177/1/7

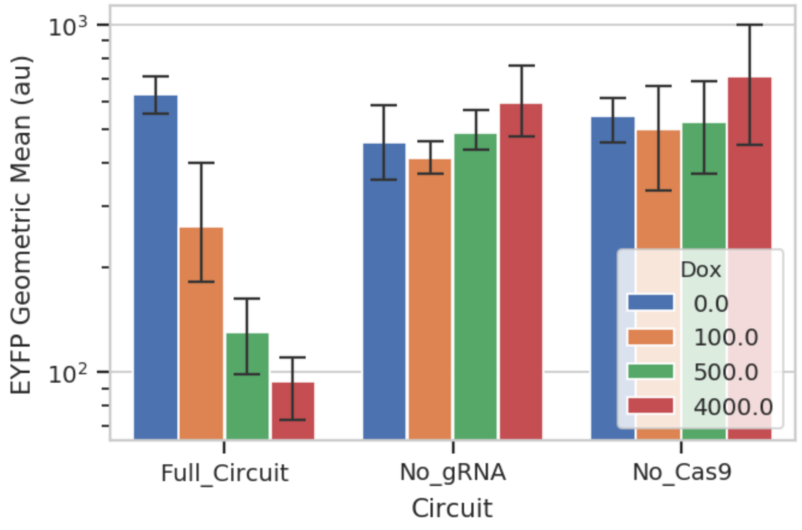
for the reason why. I'm using them here to demonstrate the capability, rather than argue that you should perform your analysis this way.

```
ex_stat = flow.ChannelStatisticOp(name = "FITC_mean",
                                  channel = "FITC-A",
                                  by = ["Condition", "Dox"],
                                  function = flow.geom_mean,
                                  subset = "Transfected == True").apply(ex_gm)
```

(continues on next page)

(continued from previous page)

```
ex_stat = flow.ChannelStatisticOp(name = "FITC_by_replicate",
                                  channel = "FITC-A",
                                  by = ["Condition", "Dox", "Replicate"],
                                  function = flow.geom_mean,
                                  subset = "Transfected == True").apply(ex_stat)

ex_stat = flow.TransformStatisticOp(name = "FITC_by_replicate",
                                    statistic = ("FITC_by_replicate", "geom_mean"),
                                    by = ["Condition", "Dox"],
                                    function = flow.geom_sd_range).apply(ex_stat)

flow.BarChartView(statistic = ("FITC_mean", "geom_mean"),
                  error_statistic = ("FITC_by_replicate", "geom_sd_range"),
                  variable = "Condition",
                  scale = "log",
                  huefacet = "Dox").plot(ex_stat,
                                         xlabel = 'Circuit',
                                         ylabel = 'EYFP Geometric Mean (au)',
                                         capsize = 5,
                                         errwidth = 1)
```



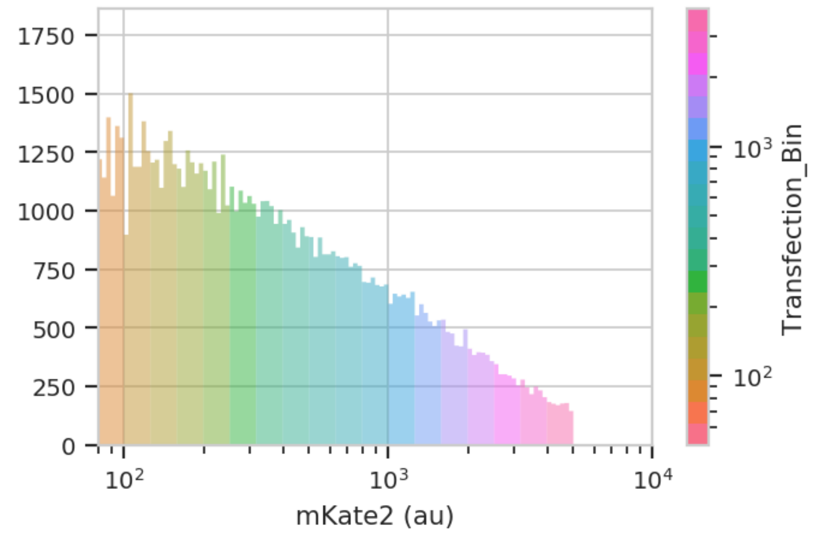
So that's useful, but maybe there's more in this data. We've noticed in our lab that gene circuit behavior frequently changes as copy number changes. Is this the case here? We can bin the data by transfection level, and see if the behavior changes as the bin number increases. We can also ask that the number of events per bin be included as another piece of metadata, so we can exclude bins with a small number of events.

```
bin_op = flow.BinningOp(name = "Transfection_Bin",
                        channel = "PE-TxRed YG-A",
                        bin_width = 0.1,
                        scale = "log",
                        bin_count_name = "Transfection_Bin_Count")

ex_bin = bin_op.apply(ex_gm)
```

(continues on next page)

(continued from previous page)

```
flow.HistogramView(channel =  "PE-TxRed YG-A",
                   huefacet = "Transfection_Bin",
                   scale = 'log',
                   huescale = 'log',
                   subset = "Transfected == True & "
                            "Transfection_Bin_Count > 1000").plot(ex_bin,
                                                                  lim = (80, 10000),
                                                                  xlabel = 'mKate2 (au)')
```



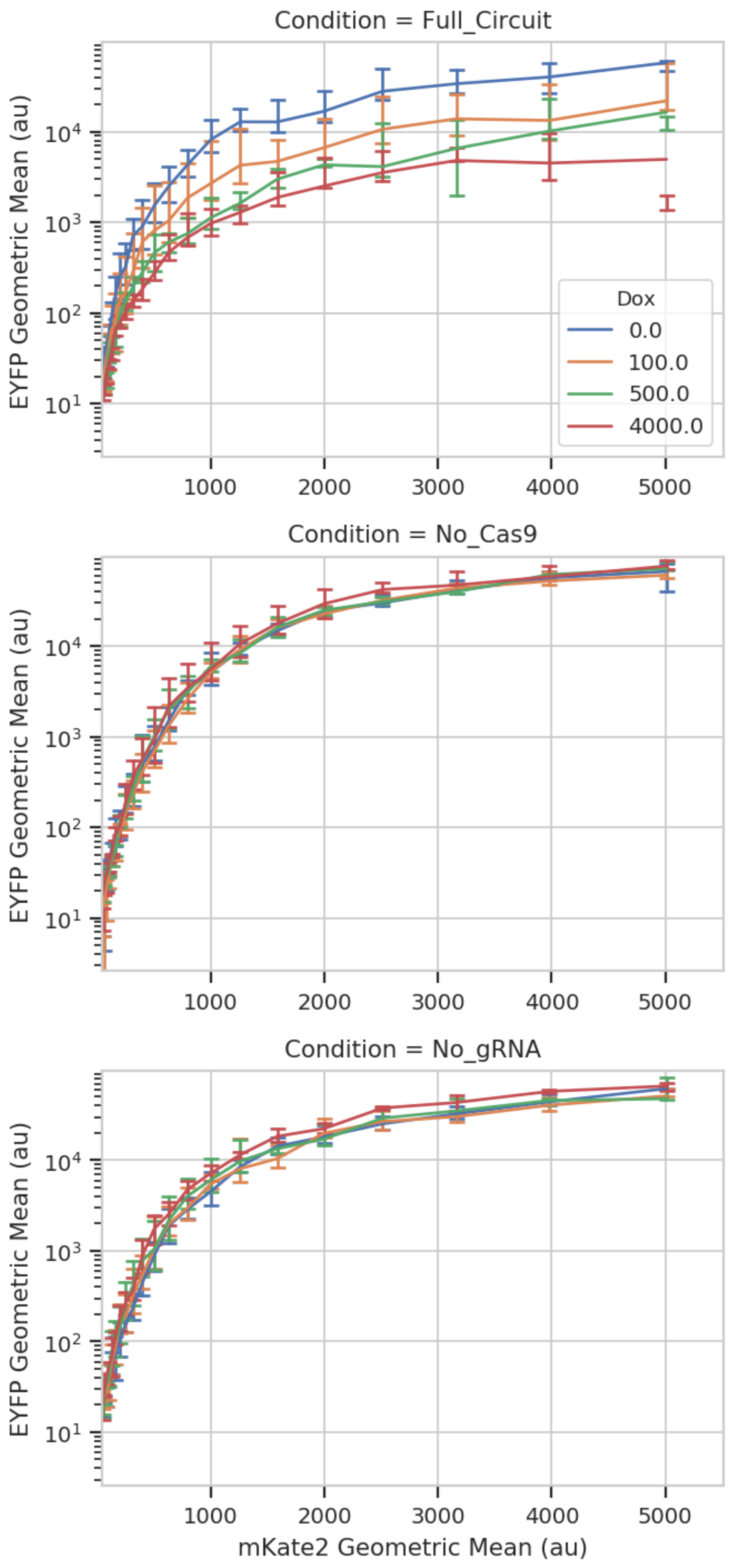
Now plot a line plot with transfection bin on the X-axis and mean FITC output on the Y-axis.

```
ex_stat = flow.ChannelStatisticOp(name = "FITC_mean",
                                  channel = "FITC-A",
                                  by = ["Condition", "Dox", "Transfection_Bin"],
                                  function = flow.geom_mean,
                                  subset = "Transfected == True & "
                                           "Transfection_Bin_Count > 500").apply(ex_bin)

ex_stat = flow.ChannelStatisticOp(name = "FITC_by_repl",
                                  channel = "FITC-A",
                                  by = ["Condition", "Dox", "Transfection_Bin",
↪"Replicate"],
                                  function = flow.geom_mean,
                                  subset = "Transfected == True & "
                                           "Transfection_Bin_Count > 500").apply(ex_stat)

ex_stat = flow.TransformStatisticOp(name = "FITC_by_repl",
                                    statistic = ("FITC_by_repl", "geom_mean"),
                                    by = ["Condition", "Dox", "Transfection_Bin"],
                                    function = flow.geom_sd_range).apply(ex_stat)

flow.Stats1DView(statistic = ("FITC_mean", "geom_mean"),
                 error_statistic = ("FITC_by_repl", "geom_sd_range"),
                 variable = "Transfection_Bin",
```

(continues on next page)

(continued from previous page)

```
                 huefacet = "Dox",
                 variable_scale = 'linear',
                 scale = 'log',
                 yfacet = "Condition").plot(ex_stat,
                                            xlabel = 'mKate2 Geometric Mean (au)',
                                            ylabel = 'EYFP Geometric Mean (au)',
                                            sharex = False,
                                            capsize = 3)
```


Condition = Full_Circuit
Condition = No_Cas9
Condition = No_gRNA
EYFP Geometric Mean (au)
mKate2 Geometric Mean (au)
Dox
0.0
100.0
500.0
4000.0
10^1
10^2
10^3
10^4
1000
2000
3000
4000
5000

TASBE Workflow Example

This notebook demonstrates using `cytoflow` for doing calibrated flow cytometry, converting the arbitrary units from the flow cytometer to Molecules Equivalent Fluorescein (MEFLs).

Our implementation closely follows that described in Beal et al and its application in Davidsohn et al.. It consists of four steps:

- Autofluorescence correction.
- Spectral bleedthrough correction.
- Calibration to physical units (MEFLs, etc.)
- Mapping to the same logical units (MEFLs in the FITC channel).

Each step requires a particular set of controls. They'll be described in more detail below, but they are (in short): * Blank (unstained, untransfected, untransformed) cells (to do autofluorescence removal) * One-color controls for each channel (to do spectral bleedthrough correction) * Calibration beads (to do physical unit calibration) * Two-color controls (or perhaps 3- or 4-color controls) to do logical unit mapping.

Experimental Layout

The experiment whose data we'll be analyzing characterizes a TALE transcriptional repressor (TAL14, from Li et al). The experiment is a multi-plasmid transient transfection in mammalian cells, depicted below:

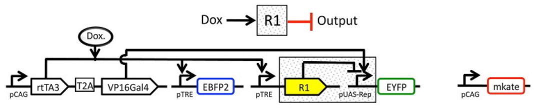


Fig. 5: Genetic circuit

The small molecule doxycycline ("Dox") drives the transcriptional activator rtTA3 to activate the transcriptional repressor ("R1" in the diagram), which then represses output of the yellow fluorescent protein EYFP. rtTA3 also drives expression of a blue fluorescent protein, eBFP, which serves as a proxy for the amount of repressor. Finally, since we're doing transient transfection, there's a huge amount of variability in the level of transfection; we measure transfection level with a constitutively expressed red fluorescent protein, mKate.

Setup

Connect `matplotlib` to the IPython notebook, and load some modules.

```
# change "inline" to "notebook" if you want ot interact with the plots.
# this command needs to happen in its OWN CELL.
%matplotlib inline
```

```
# load the cytoflow library
import cytoflow as flow
```

```
# if your figures are too big or too small, you can scale them by changing matplotlib's␣
↪DPI
import matplotlib
matplotlib.rc('figure', dpi = 160)
```

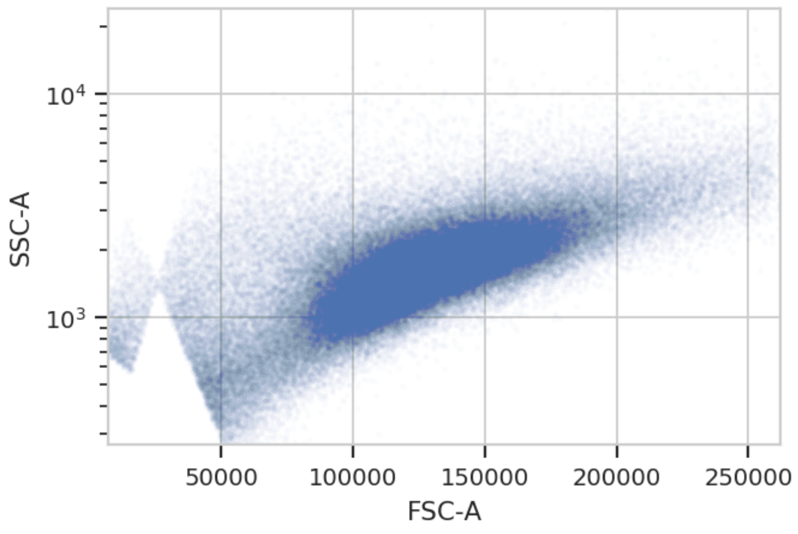
As is usual with `cytoflow`, we start by mapping the files to experimental conditions. Here, we only vary the amount of Doxycycline, the small molecule inducer of the repressor.

```
# [Dox] --> filename
inputs = {
    0.0 : 'TAL14_1.fcs',
    0.1 : 'TAL14_2.fcs',
    0.2 : 'TAL14_3.fcs',
    0.5 : 'TAL14_4.fcs',
    1.0 : 'TAL14_5.fcs',
    2.0 : 'TAL14_6.fcs',
    5.0 : 'TAL14_7.fcs',
   10.0 : 'TAL14_8.fcs',
   20.0 : 'TAL14_9.fcs',
   50.0 : 'TAL14_10.fcs',
  100.0 : 'TAL14_11.fcs',
  200.0 : 'TAL14_12.fcs',
  500.0 : 'TAL14_13.fcs',
 1000.0 : 'TAL14_14.fcs',
 2000.0 : 'TAL14_15.fcs'}

tubes = []

for dox, filename in inputs.items():
    tube = flow.Tube(file = filename,
                     conditions = {'Dox' : dox})
    tubes.append(tube)

import_op = flow.ImportOp(conditions = {'Dox' : "float"},
                          tubes = tubes)

ex = import_op.apply()
```

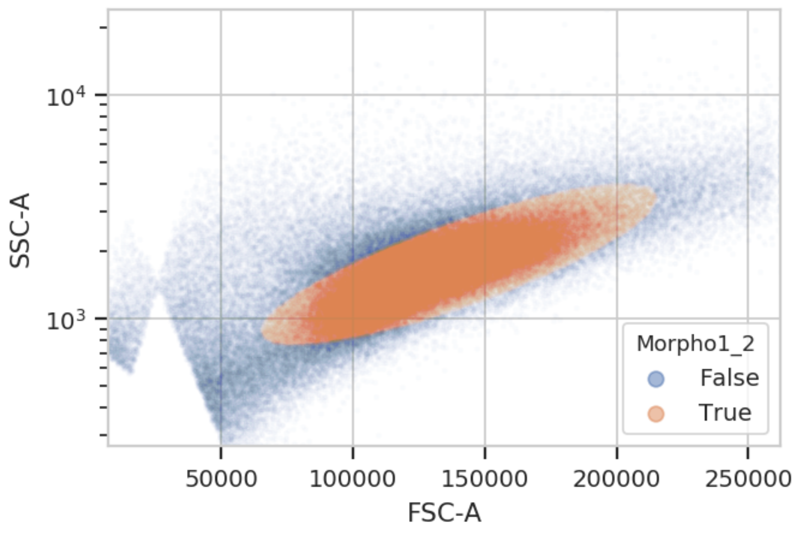
Morphological gate

Start by gating out the cells that we want. We can apply this gate to the controls, too.

```
flow.ScatterplotView(xchannel = "FSC-A",
                     ychannel = "SSC-A",
                     yscale = "log").plot(ex, alpha = 0.01)
```



```
gm_1 = flow.GaussianMixtureOp(name = "Morpho1",
                              channels = ["FSC-A", "SSC-A"],
                              scale = {"SSC-A" : "log"},
                              num_components = 2,
                              sigma = 2)
gm_1.estimate(ex)
ex_morpho = gm_1.apply(ex)

flow.ScatterplotView(xchannel = "FSC-A",
                     ychannel = "SSC-A",
                     yscale = "log",
                     huefacet = "Morpho1_2").plot(ex_morpho, alpha = 0.01)
```

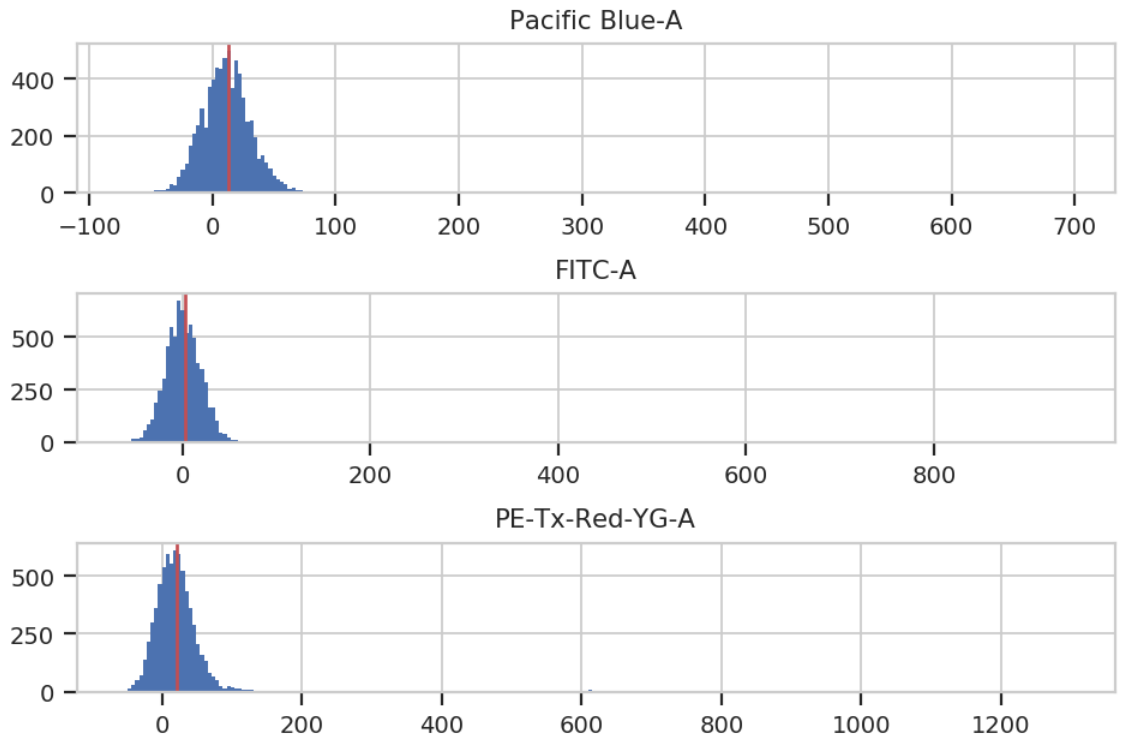


Autofluorescence correction

To account for autofluorescence, we measure a tube of *blank cells* (unstained, untransfected, untransformed – not fluorescing.) The autofluorescence operation finds the (arithmetic) median of the blank cells' distributions in the fluorescence channels and subtracts it from all the observations in the experimental data.

The diagnostic plot just shows the fluorescence histograms and the medians. Make sure that they're unimodal and the median is at the peak.

```
af_op = flow.AutofluorescenceOp()
af_op.blank_file = "controls/Blank-1_H12_H12_P3.fcs"
af_op.channels = ["Pacific Blue-A", "FITC-A", "PE-Tx-Red-YG-A"]

af_op.estimate(ex_morpho, subset = "Morpho1_2 == True")
af_op.default_view().plot(ex_morpho)
```



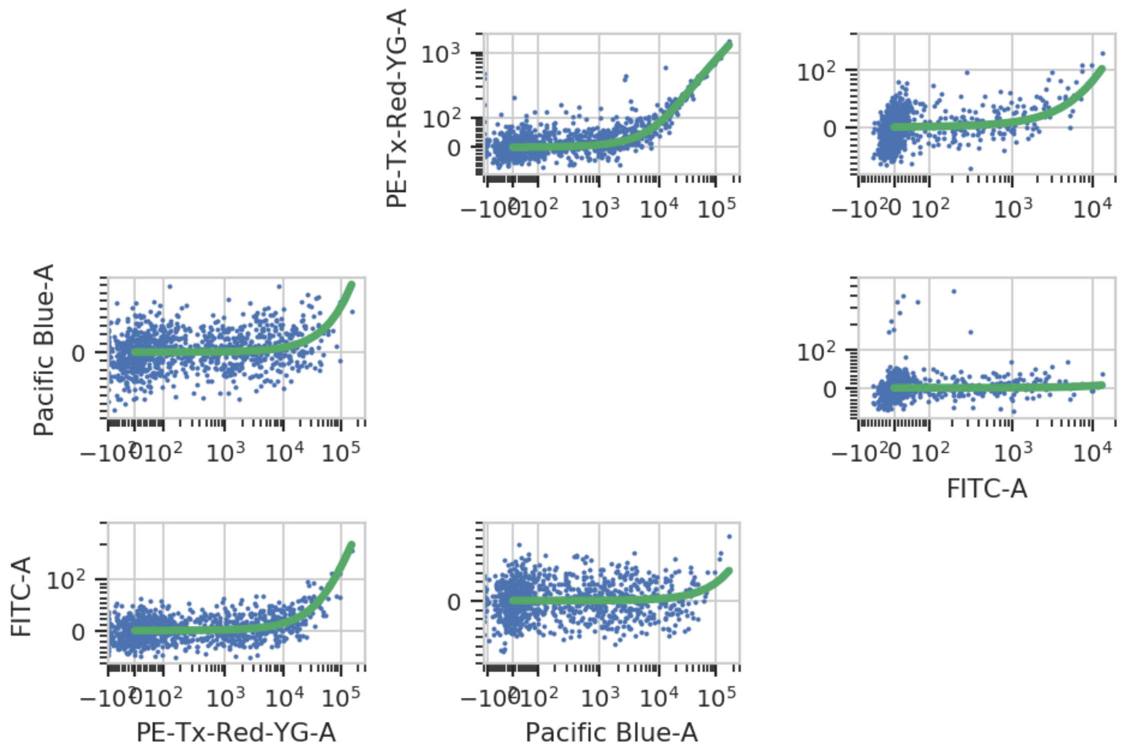
```
ex_af = af_op.apply(ex_morpho)
```

Spectral bleedthrough correction

This operation characterizes how much a fluorophore's signal shows up in channels other than the one you are using to detect it. For example, EYFP is primarily measured in the (yellow) FITC channel, but some signal also shows up in the (red) PE-Texas Red channel.

The controls for this operation are single-fluorescent controls – either single fluorescent proteins or singly-stained cells. They should fluoresce brightly, so as to best-characterize bleedthrough.

```
bl_op = flow.BleedthroughLinearOp()
bl_op.controls = {'Pacific Blue-A' : 'controls/EBFP2-1_H9_H09_P3.fcs',
                  'FITC-A' : 'controls/EYFP-1_H10_H10_P3.fcs',
                  'PE-Tx-Red-YG-A' : 'controls/mkate-1_H8_H08_P3.fcs'}

bl_op.estimate(ex_af, subset = "Morpho1_2 == True")
bl_op.default_view().plot(ex_af)
```



```
ex_bl = bl_op.apply(ex_af)
```

Bead Calibration

Unfortunately, the raw measurements from a flow cytometer are sensitive to many factors. These range from the precise optical configuration, to the laser power, to the PMT voltage, to the last time the instrument was cleaned and calibrated. Thus, the measurements taken on one instrument are not directly compatible with those taken on another. (Sometimes, even day-to-day variation on the same instrument is enough to ruin comparisons.)

One way around this is to calibrate your measurements against a stable calibrant. Our favorite is a set of stable fluorescent beads, such as the Spherotech RCP-30-5As. This module calibrates measurements to molecules of equivalent fluorophores, which can make calibrations (more) comparable. Read the beads' documentation for more details.

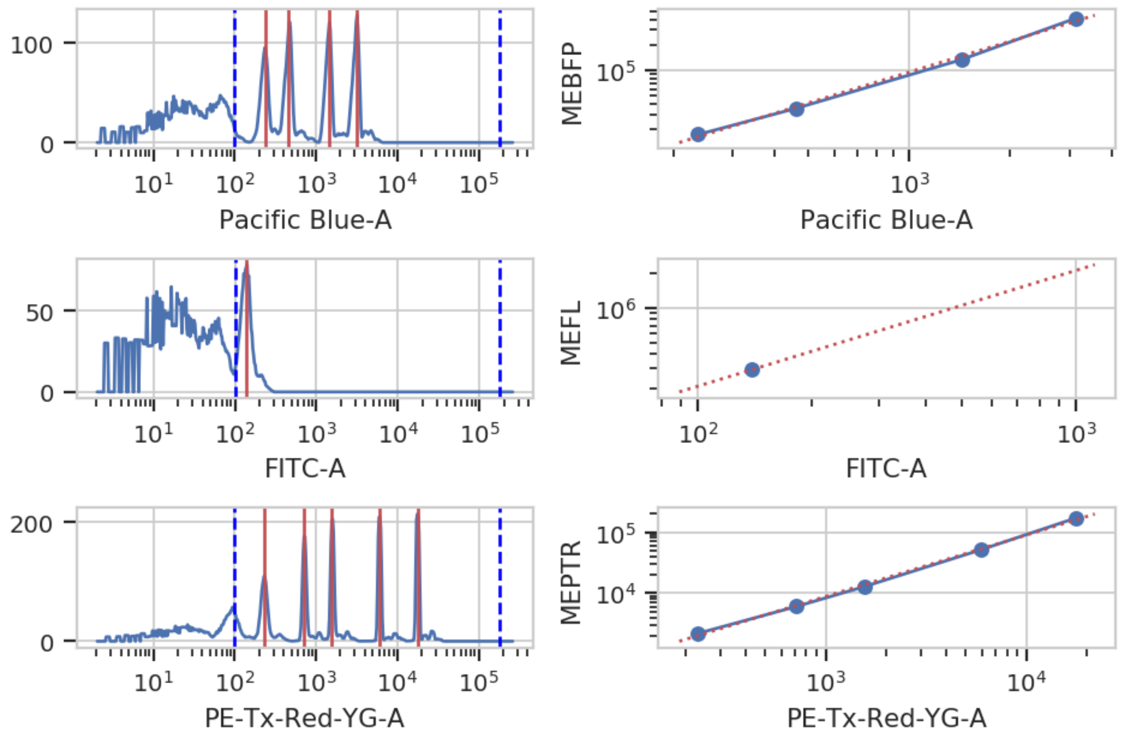
NB: Adding new beads is easy! See the `bead_calibration` module's source code.

```
bead_op = flow.BeadCalibrationOp()
bead_op.beads = flow.BeadCalibrationOp.BEADS["Spherotech RCP-30-5A Lot AA01-AA04, AB01,␣
↪AB02, AC01, GAA01-R"]
bead_op.units = {"Pacific Blue-A" : "MEBFP",
```

(continues on next page)

(continued from previous page)

```
                  "FITC-A" : "MEFL",
                  "PE-Tx-Red-YG-A" : "MEPTR"}

bead_op.beads_file = "controls/BEADS-1_H7_H07_P3.fcs"
bead_op.estimate(ex_bl)

bead_op.default_view().plot(ex_bl)
```



```
ex_beads = bead_op.apply(ex_bl)
```

Color Translation

At the end of the day, we want to be able to compare signals collected in the yellow channel to signals from the blue and red channels in comparable units. Unfortunately, different fluorescent proteins mature at different rates, have different quantum efficiencies, etc – so even if we measure the same absolute fluorescence (in photon flux, say), we still can't say that the number of molecules is the same.

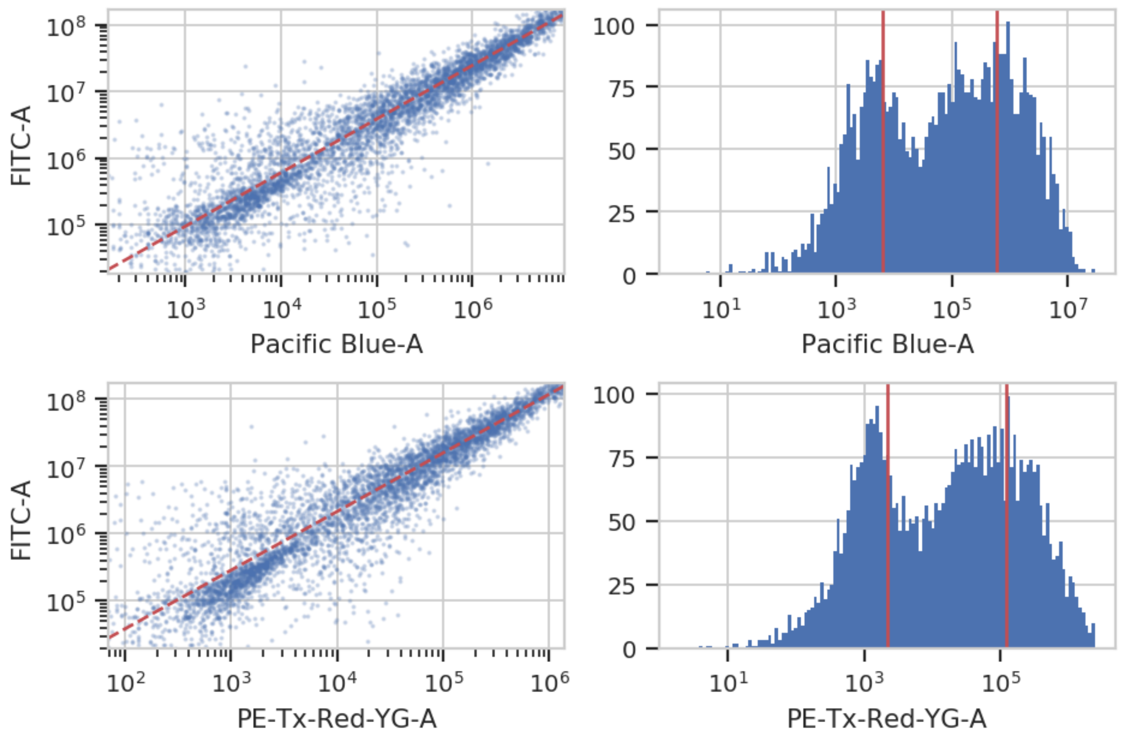
One way around this is to use a biological control where you are releatively certain that the number of molecules *is* the same to compute a conversion factor. For example, EYFP and mKate and EBFP2 all expressed under the same promoter in the same cell line should produce the same amounts of RNA and comparable amounts of protein. This module lets you use a set of controls like this to convert between *biological* signals.

```
ct_op = flow.ColorTranslationOp()
ct_op.controls = {("Pacific Blue-A", "FITC-A") : "controls/RBY-1_H11_H11_P3.fcs",
                  ("PE-Tx-Red-YG-A", "FITC-A") : "controls/RBY-1_H11_H11_P3.fcs"}
ct_op.mixture_model = True
```

(continues on next page)

(continued from previous page)

```
ct_op.estimate(ex_beads, subset = "Morpho1_2 == True")
ct_op.default_view().plot(ex_beads)
```

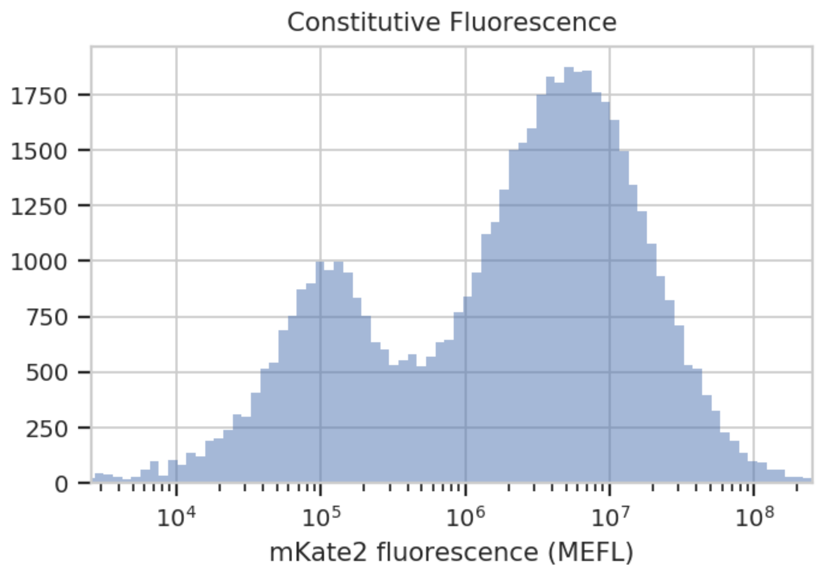


```
ex_calib = ct_op.apply(ex_beads)
```

Binned Analysis

As described above, the example data in this notebook is from a transient transfection of mammalian cells in tissue culture. What this means is that there's a really broad distribution of fluorescence, corresponding to a broad distribution of transfection levels.

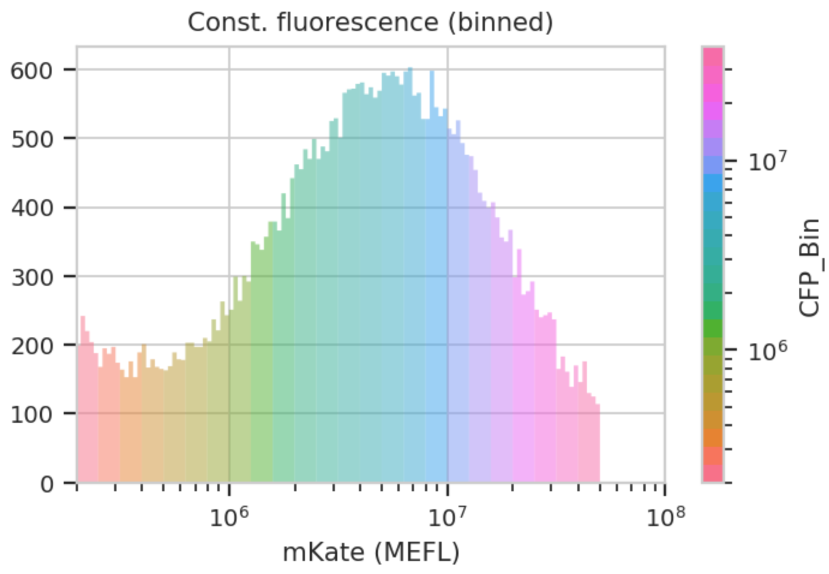
```
flow.HistogramView(channel = "PE-Tx-Red-YG-A",
                   scale = "log",
                   subset = "Morpho1_2 == True").plot(ex_calib,
                                                       title = "Constitutive Fluorescence
↪",
                                                       xlabel = "mKate2 fluorescence␣
↪(MEFL)")
```



See? The left peak is cells that are untransfected; the right peak is cells that were. The cells that were transfected seem to range from about 3x10^5 MEFL up to 10^8 MEFL, over two orders of magnitude.

The way we handle this data is by dividing the cells into bins depending on their transfection levels. We find that cells that recieved few plasmids frequently behave differently (quantitatively speaking) than cells that received many plasmids. The `BinningOp` module applies evenly spaced bins; in this example, we're going to apply them on a log scale, every 0.1 log-units.

```
ex_bin = flow.BinningOp(name = "CFP_Bin",
                        bin_count_name = "CFP_Bin_Count",
                        channel = "PE-Tx-Red-YG-A",
                        scale = "log",
                        bin_width = 0.1).apply(ex_calib)

flow.HistogramView(channel = "PE-Tx-Red-YG-A",
                   huefacet = "CFP_Bin",
                   huescale = "log",
                   scale = "log",
                   subset = "Morpho1_2 == True and "
                            "PE_Tx_Red_YG_A >= 200000 and CFP_Bin_Count > 1000").plot(ex_
↪bin,
                                                                                       ␣
↪xlabel = "mKate (MEFL)",
                                                                                       ␣
↪title = "Const. fluorescence (binned)",
                                                                                       ␣
↪lim = (2e5, 1e8))
```



Now we can start our analysis properly. For each unique combination of [Dox] and bin, we compute four statistics: - The geometric mean of the Pacific Blue channel, the "input fluorescent protein" or IFP - The geometric mean of the ratio of IFP and PE-Texas Red channels - The geometric mean of the FITC channel, the "output fluorescent protein" or OFP - The geometric mean of the ratio of the FITC and PE-Texas Red channels

Remember, because we calibrated the measurements, the CFP, IFP and OFP channels are on the same scale, which makes comparisons between them (ratios, etc) meaningful.

```
ex_stats = flow.ChannelStatisticOp(name = "IFP",
                                   channel = "Pacific Blue-A",
                                   by = ["Dox", "CFP_Bin"],
                                   function = flow.geom_mean,
                                   subset = "Morpho1_2 == True and "
                                            "PE_Tx_Red_YG_A > 200000 and CFP_Bin_Count >␣
↪1000").apply(ex_bin)

ex_stats = flow.FrameStatisticOp(name = "IFP",
                                 by = ["Dox", "CFP_Bin"],
                                 function = lambda x: flow.geom_mean(x["Pacific Blue-A"]␣
↪/ x["PE-Tx-Red-YG-A"]),
                                 statistic_name = "geom_mean_per_cfp",
                                 subset = "Morpho1_2 == True and "
                                          "PE_Tx_Red_YG_A > 200000 and CFP_Bin_Count >␣
↪1000").apply(ex_stats)

ex_stats = flow.ChannelStatisticOp(name = "OFP",
                                   channel = "FITC-A",
                                   by = ["Dox", "CFP_Bin"],
                                   function = flow.geom_mean,
                                   subset = "Morpho1_2 == True and "
                                            "PE_Tx_Red_YG_A > 200000 and CFP_Bin_Count >␣
↪1000").apply(ex_stats)

ex_stats = flow.FrameStatisticOp(name = "OFP",
                                 by = ["Dox", "CFP_Bin"],
```

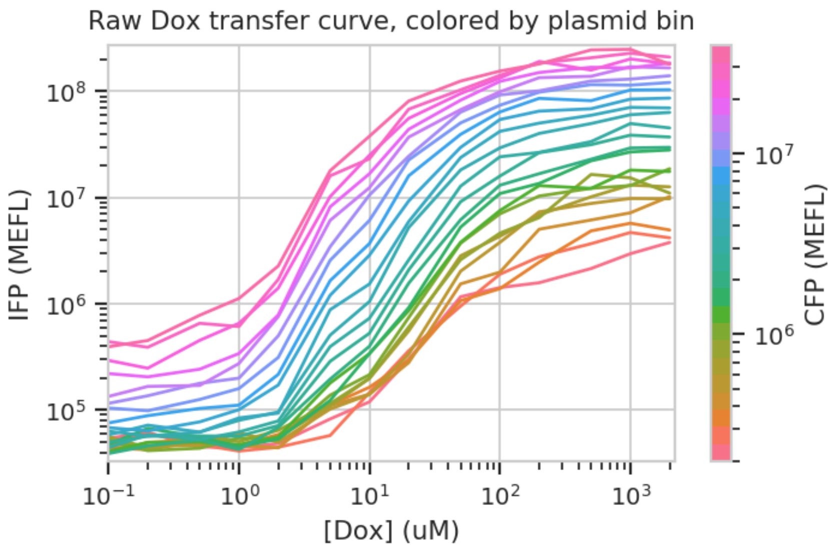
(continues on next page)

(continued from previous page)

```
                                    function = lambda x: flow.geom_mean(x["FITC-A"] / x["PE-
↪Tx-Red-YG-A"]),
                                    statistic_name = "geom_mean_per_cfp",
                                    subset = "Morpho1_2 == True and "
                                             "PE_Tx_Red_YG_A > 200000 and CFP_Bin_Count >␣
↪1000").apply(ex_stats)
```

Let's start with our input fluorescent protein, EBFP2. Do we see more of it as we increase [Dox]? Is the response different in different bins (i.e. different transfection levels?)

```
flow.Stats1DView(statistic = ("IFP", "geom_mean"),
                 variable = "Dox",
                 variable_scale = "log",
                 scale = "log",
                 huefacet = "CFP_Bin",
                 huescale = "log").plot(ex_stats,
                                         xlabel = "[Dox] (uM)",
                                         ylabel = "IFP (MEFL)",
                                         huelabel = "CFP (MEFL)",
                                         title = "Raw Dox transfer curve, colored by␣
↪plasmid bin")
```



The answer to both questions is "yes": there is an increase in IFP signal as we increase [Dox], and the precise quantitative character of the curve is different depending on which bin we select. Of particular interest is how moderately transfected bins (say, 10^5 through 10^6) have greater on-off ratios than bins on either side of the transfection distribution.

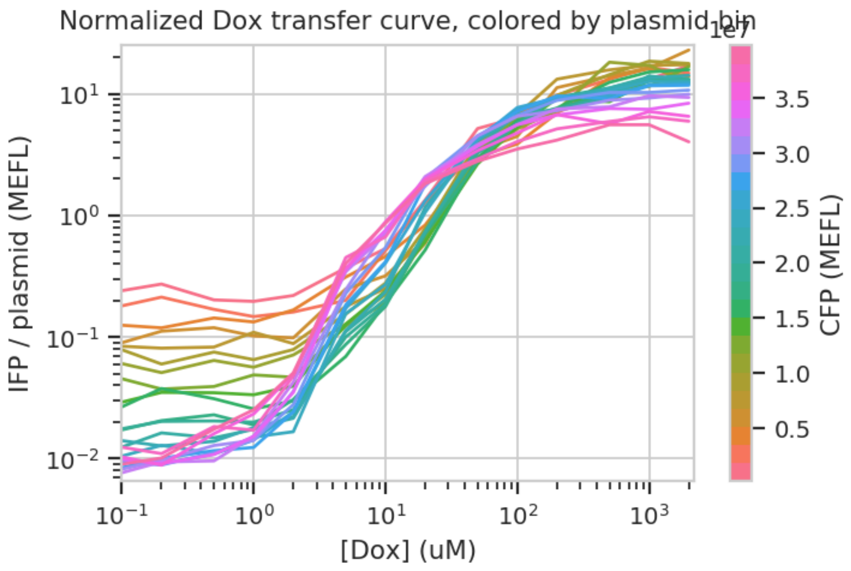
Does this hold when we normalize by transfection (ie, divide by CFP)?

```
flow.Stats1DView(statistic = ("IFP", "geom_mean_per_cfp"),
                 variable = "Dox",
                 scale = "log",
                 variable_scale = "log",
                 huefacet = "CFP_Bin").plot(ex_stats,
```

(continues on next page)

(continued from previous page)

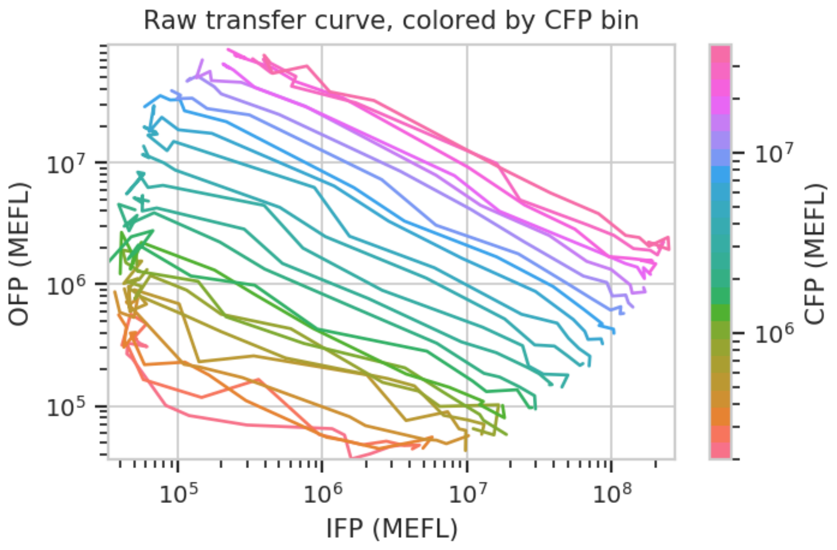
```
                                                 xlabel = "[Dox] (uM)",
                                                 ylabel = "IFP / plasmid (MEFL)",
                                                 huelabel = "CFP (MEFL)",
                                                 title = "Normalized Dox transfer curve,␣
↪colored by plasmid bin")
```



Sure enough, the moderately trasnfected curves fall right on top of eachother, making it really obvious that the fold induction of different bins is different.

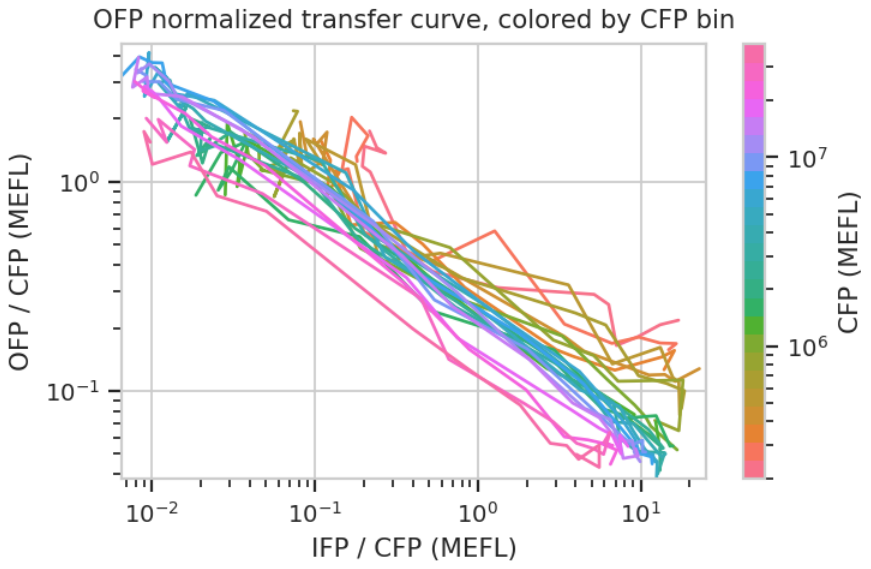
Things get really interesting when we start plotting different statistics against eachother. To do so, the statistics must have the same indices (i.e, the same values passed to by in the module that created the statistic.) In the example below, we plot how the geometric mean of each bin's IFP and OFP change as we vary Dox. There is one line plotted per bin.

```
flow.Stats2DView(xstatistic = ("IFP", "geom_mean"),
                 ystatistic = ("OFP", "geom_mean"),
                 variable = "Dox",
                 xscale = "log",
                 yscale = "log",
                 huescale = "log",
                 huefacet = "CFP_Bin").plot(ex_stats,
                                           xlabel = "IFP (MEFL)",
                                           ylabel = "OFP (MEFL)",
                                           huelabel = "CFP (MEFL)",
                                           title = "Raw transfer curve, colored by CFP␣
↪bin")
```



We can do the same thing with the scaled statistics, too.

```
flow.Stats2DView(xstatistic = ("IFP", "geom_mean_per_cfp"),
                 ystatistic = ("OFP", "geom_mean_per_cfp"),
                 variable = "Dox",
                 xscale = "log",
                 yscale = "log",
                 huescale = "log",
                 huefacet = "CFP_Bin").plot(ex_stats,
                                            xlabel = "IFP / CFP (MEFL)",
                                            ylabel = "OFP / CFP (MEFL)",
                                            huelabel = "CFP (MEFL)",
                                            title = "OFP normalized transfer curve,␣
↪colored by CFP bin")
```



How-To Guides for Developers and Contributors

HOWTO: Install Cytoflow Modules

To use the Cytoflow modules in a Jupyter notebook or your own code

Cytoflow is available as a package for the Anaconda scientific Python distribution. You can install *cytoflow* through the Anaconda Navigator, or by using the command line.

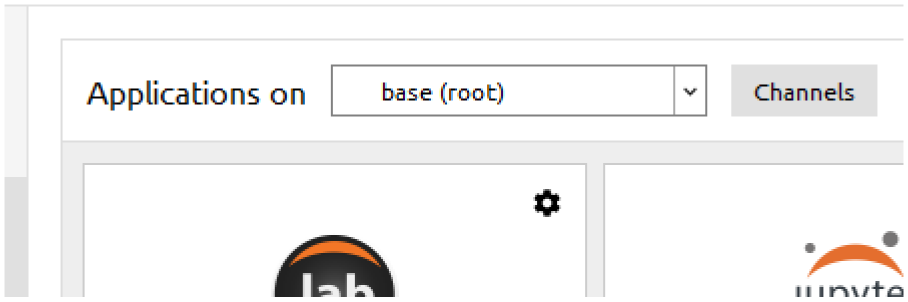
This is not the only way to get Cytoflow up and running, but it is by far the most straightforward.

Installing from the `Anaconda Navigator`

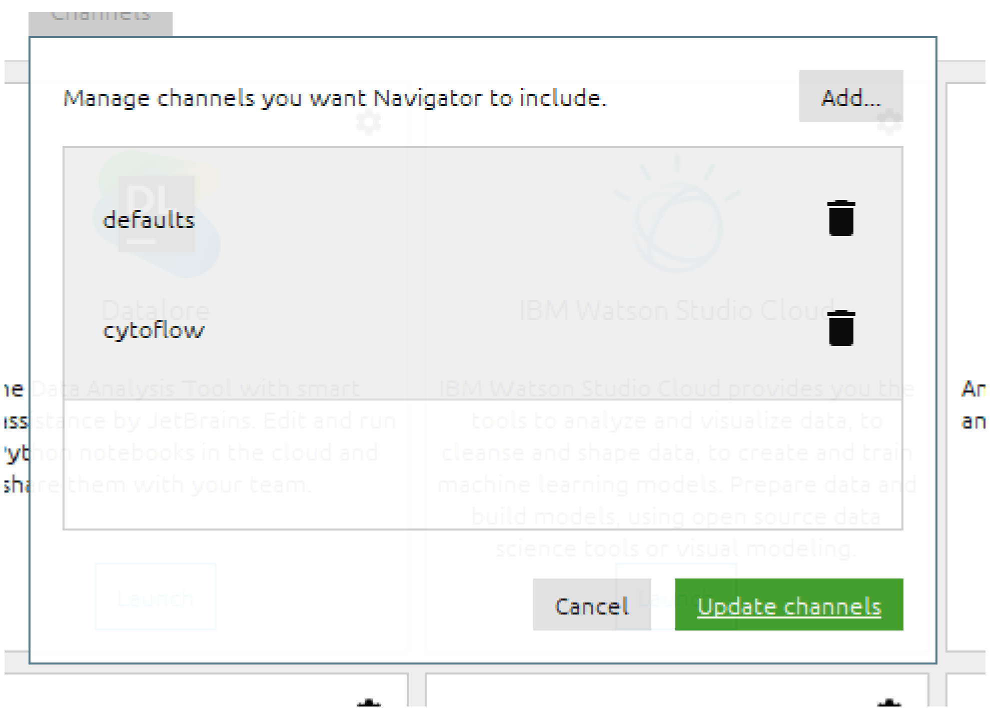
- Start by installing the Anaconda Python distribution. **Make sure to install a 64-bit version, unless you will be building *cytoflow* yourself and you know what you're doing.**

  Download Anaconda here

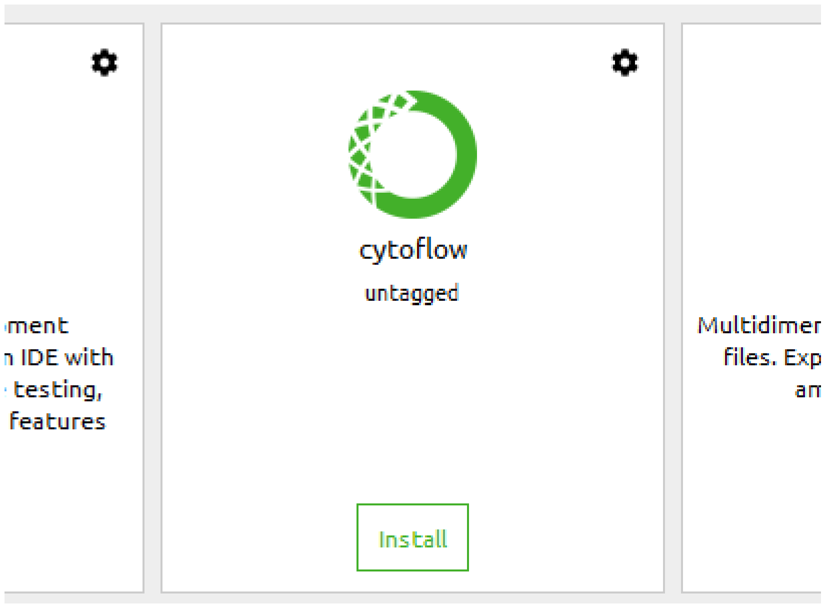
- Either from the Start Menu (Windows) or the Finder (Mac), run the `Anaconda Navigator`
- Click the `Channels` button.



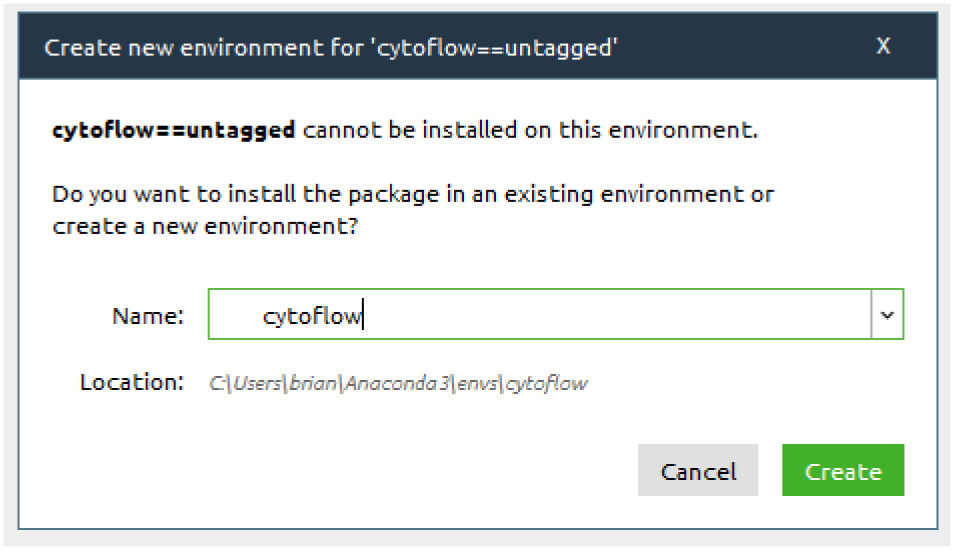
- Click `Add...` and type `cytoflow`. Select "Update channels."



- The application `cytoflow` should appear in the launcher. Click the `Install` button.



- `Navigator` asks if you'd like to install in a new environment. Say `Yes`..


Create new environment for 'cytoflow==untagged'

cytoflow==untagged cannot be installed on this environment.

Do you want to install the package in an existing environment or create a new environment?

Name: cytoflow

Location: C:\Users\brian\Anaconda3\envs\cytoflow

NOTE: Be patient. Anaconda Navigator is slow. NOTE: Make sure that you choose an environment that does not already exist!

- To verify installation, start a Jupyter notebook.
  - First, *make sure you have the ``cytoflow`` environment selected.*
  - From the `Anaconda Navigator`, install and then launch `Jupyter notebook`.
  - Create a new *Python 3* notebook.
  - In the first cell, type `import cytoflow` and press `Shift+Enter`. If Python doesn't complain, you're good to go. (If it does, please submit a bug report at https://github.com/cytoflow/cytoflow/issues)
- **Note: When you install Cytoflow this way, the point-and-click application is installed as well.** Launching it from the `Anaconda Navigator` will be significantly faster than downloading the pre-packaged binary.

Installing from the command line

- Start `Anaconda Prompt` from the Start Menu (Windows) or Finder (Mac).
- Add the `cytoflow` channel:

```
conda config --add channels cytoflow
```

- Create a new environment and install `cytoflow` and the Jupyter notebook. In this example, the new environment will be called `cf` – feel free to choose a different name:

```
conda create --name cf cytoflow notebook
```

- Activate the new environment:

```
conda activate cf
```

- Launch the Jupyter notebook:

```
jupyter notebook
```

- Create a new *Python 3* notebook. In the first cell, type `import cytoflow` and press `Shift+Enter`. If Python doesn't complain, you're good to go. (If it does, please submit a bug report!)

- This method ALSO installs the GUI. You should be able to run it by activating your new environment and running the `cytoflow` script.

To hack on the code

Cytoflow depends on a huge number of libraries from the Scientific Python ecosystem, and a change in any one of their APIs will break the `cytoflow` library. So, I have pinned the versions of all of `cytoflow`'s dependencies, which all but guarantees that you'll need to install into a virtual environment. This will ensure that the rest of your Python installation doesn't break.

I strongly recommend using Anaconda to install the proper dependencies. A PyPI package (installable using `pip`) is also available. The following instructions assume that you have installed Anaconda (as above) and launched an Anaconda prompt.

Finally, `cytoflow` relies on one C++ extension. On Linux, installing the requirements for building it is straightforward. On MacOS it is harder, and on Windows it is extremely difficult. Instead, as part of rolling a new release, the appropriate files are made available on the GitHub releases page. The procedure below includes instructions for downloading and installing the appropriate file.

- Install the development dependencies
  - On Ubuntu: `apt-get git swig python-dev`
  - On Windows: Install a copy of `git`. I use git-for-windows
  - On MacOS: Install a copy of `git` from the Git website.
- If you haven't, add the `cytoflow` channel to conda:

```
conda config --add channels cytoflow
```

- Clone the repository:

```
git clone --recurse-submodules https://github.com/cytoflow/cytoflow.git
```

- Create a new environment. In this example, I have called it `cf_dev`. In the new repository you just cloned, say:

```
conda env create --name cf_dev --file environment.yml
```

Note: On Windows, you must edit `environment.yml` before you execute `conda env create`. Remove the last line, the one that reads `- nomkl # [not win]`

- Activate the new environment:

```
conda activate cf_dev
```

- **On Windows and MacOS only,** do the following to prevent `cytoflow` from trying to build the C++ extension.
  - **On Windows (in CMD)**:

```
set NO_LOGICLE=True
```

  - **On MacOS (or on Windows bash)**:

```
export NO_LOGICLE=True
```

- Install `cytoflow` in developer's mode:

```
python setup.py develop
```

- From the GitHub releases page download the appropriate extension file for the version you're installing.
  - **On Windows (64-bit)**: `_Logicle.cp38-win_amd64.pyd`
  - **On MacOS**: `_Logicle.cpython-38m-darwin.so`
- Copy the file you just download into the `cytoflow/utility/logicle_ext/` folder in your source tree.
- Test that everything works. Start a `python` interpreter and say:

  ```
  import cytoflow
  ```

  If you don't get any errors, you're good to go.

Running the point-and-click GUI program

There are pre-built bundles available at http://cytoflow.github.io/

Alternately, you can follow the instructions above for installing the Anaconda package, then run `cytoflow` through the Anaconda Navigator or via the command line.

HOWTO: Spin a new release

Tests

- We use two continuous integration platforms to run tests and build binaries and documentations: GitHub Actions, ReadTheDocs.

  Finished releases are published to GitHub releases, Anaconda Cloud, and the Python Package Index.
- Make sure that the *`cytoflow`* tests pass, both locally and on GitHub:

  ```
  nose2 -c package/nose2.cfg cytoflow.tests -N 8
  ```

- Make sure the *`cytoflowgui`* tests pass. **You must do this locally; I'm still working on why it doesn't run on the CI platform.**

  ```
  nose2 -c package/nose2.cfg cytoflowgui.tests -N 8
  ```

- Make sure the GitHub Actions are running to completion, at https://github.com/cytoflow/cytoflow/actions

Documentation

- Build the developers' manual and check it for completeness:

  ```
  conda install "sphinx==4.2.0" pandoc
  python setup.py build_sphinx
  ```

- Build the user manual and check it for completeness:

  ```
  sphinx-build docs/user_manual/reference cytoflowgui/help
  ```

- Make sure that the ReadTheDocs build is working at https://readthedocs.org/projects/cytoflow/builds/

Test the packaging

- Build the conda package locally:

```
conda build package/conda_recipes/cytoflow
```

- Install the local package in a new environment:

```
conda create --name cytoflow.test --use-local cytoflow
```

- Activate the test environment, make sure you can import cytoflow, and make sure the GUI runs:

```
conda activate cytoflow.test
python -c "import cytoflow"
cytoflow
```

- Make sure that the `pyinstaller` distribution will build on your local machine (back in your development environment).

```
pip install pyinstaller==4.8
pyinstaller package/pyinstaller.spec
```

- Make sure that `pyinstaller` built the executables on all three supported platforms. On each of the three supported platforms?
  - Download the one-click from GitHub Actions. Make sure it starts and can execute a basic workflow.
  - Download the conda package from GitHub Actions. Create a local `anaconda` environment and install it. Check that it runs as both a module and a GUI

```
conda env create --name cf.test -f environment.yml
conda activate cf.test
conda install ./cytoflow-*******-tar.bz2
python -c "import cytoflow"
cytoflow
```

Versioning and dependencies

- We're using `versioneer` to manage versions. No manual versions required.
- If there are dependencies that don't have packages on Anaconda, add recipes to `package/conda_recipes` (using `conda skeleton`) and upload them to the Anaconda Cloud. Unless there's a really (really!) good reason, please make them no-arch.
- Make sure `install_requires` in `setup.py` matches `requirements.txt`
- Update the README.rst from the README.md. From the project root, say:

```
pandoc --from=markdown --to=rst --output=README.rst README.md
```

Tag and upload the release

- Push the updated docs to GitHub. Give the CI builders ~30 minutes, then check the build status on GitHub and ReadTheDocs.
- Create a new tag on the master branch. This will re-build everything on the CI builders.
- Download the artifacts.

Sign the Windows installer

To get rid of the "Unknown developer" warning in Windows, we sign the installer. This requires a hardware crypto token, so it must be done locally.

- Setup: If not done already, download and install the Windows Platform SDK. I'm using 8.1 because I couldn't get 10 to install.
- Download the Windows installer from Github.
- Open a terminal in C:Program FilesMicrosoft Platform SDKBin.
- Start the signing wizard:

  ```
  signtool.exe signwizard
  ```

- Select the installer binary.
- Under "Signing options", choose "Typical"
- Under "Signature Certificate", choose "Select from store. . . ". If the hardware key is installed and set up properly, Windows should find the correct certificate.
- Add a description such as "Flow cytometry software". For "Web location", specify "http://cytoflow.readthedocs.org"
- Check the box next to "Add a timestamp to data". Enter "http://time.certum.pl". (Probably could use digicert or some other service.)
- When prompted, enter the Common Profile PIN.
- After the wizard closes, double-check that the signing process was completed by right-clicking on the executable and checking the "Digital Signatures" tab.

Upload the artifacts and update the homepage

- Upload artifacts as appropriate to GitHub, Anaconda, and the Python Package Index. (Make sure that in the case of Anaconda, you're uploading to the organization account, not your personal account!) The GitHub action should take care of the GitHub and Anaconda packages, but not PyPI.
- At https://github.com/cytoflow/cytoflow.github.io, update the version in `_config.yml`. Push these changes to update the main download links on http://cytoflow.github.io/
- Verify that the download links at http://cytoflow.github.io/ still work!

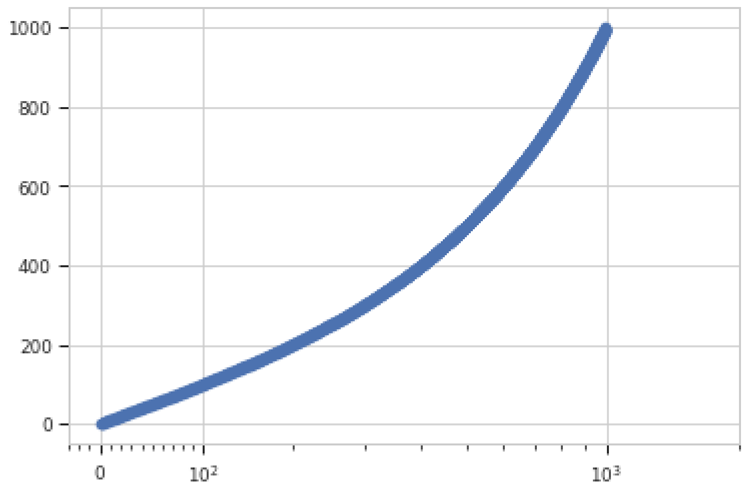
HOWTO: Use the `logicle` scale in other `matplotlib` plots

cytoflow implements an interesting scaling routine called `logicle`. It's a biexponential function that is "linear" near zero and transitions smoothly to a logarithmic scale. This is particularly useful for flow cytometry data, which often has data clustered around zero where a log-base-10 scale would introduce aliasing. (See the papers referenced below for rationale and implementation details; *cytoflow* uses the C++ code from the second paper.)

I have been asked several times how to use a `logicle` scale for other `matplotlib` plots. This is difficult because, unlike the `log` scale, `logicle` is *parameterized* – the precise location of the linear-to-log transition, how many decades of negative data, etc. all need to be specified in order to create a usable scaling function.

However, I recognize that there are occasions when you may want to use a plot that is not baked into *cytoflow*. Here's a code fragment that should get you pointed in the right direction. However, this is an *explicitly unsupported use-case* – please don't file bug reports if you're having trouble with this:

```
import matplotlib.pyplot as plt
import cytoflow as flow

x = range(1, 1000)
y = range(1, 1000)

tube1 = flow.Tube(file = 'data/RFP_Well_A3.fcs',
                  conditions = {'Dox' : 10.0})
tube2 = flow.Tube(file='data/CFP_Well_A4.fcs',
                  conditions = {'Dox' : 1.0})

import_op = flow.ImportOp(conditions = {'Dox' : 'float'},
                          tubes = [tube1, tube2])

ex = import_op.apply()
logicle = flow.utility.scale_factory('logicle', ex, channel = 'V2-A')

# the channel = 'V2-A' parameter in the scale_factory() call tells
# the logicle scale instance to use that channel to estimate its
# parameters.  if you want to use a condition or statistic, say
# condition = ..... or statistic = ..... instead.

plt.scatter(x, y)
ax = plt.gca()
plt.gca().set_xscale('logicle',
                     **logicle.get_mpl_params(ax.get_xaxis()))
```



References

[1] **A new "Logicle" display method avoids deceptive effects of logarithmic** scaling for low signals and compensated data. Parks DR, Roederer M, Moore WA. Cytometry A. 2006 Jun;69(6):541-51. PMID: 16604519 http://onlinelibrary.wiley.com/doi/10.1002/cyto.a.20258/full

[2] **Update for the logicle data scale including operational code** implementations. Moore WA, Parks DR. Cytometry A. 2012 Apr;81(4):273-7. doi: 10.1002/cyto.a.22030 PMID: 22411901 http://onlinelibrary.wiley.com/doi/10.1002/cyto.a.22030/full

Topic Guides

These documents help you understand the Cytoflow codebase at a high level, which is useful for implementing your own modules.

Design Guide

This document explains how Cytoflow is laid out and the justifications for some of the design decisions.

Project Structure

The source is organized into two main components.

- The `cytoflow` package. This package contains the actual tools for operating on cytometry data. Key modules and subpackages:
  - The `Experiment` class is the primary container for cytometry data. See the module docstrings for its use. Modify this class's API only with care, please!
  - The `cytoflow.operations` subpackage. This is where operations on data go, like transformations and gates. Adding a new operation is quite straightforward: see the documentation for *adding a new operation*.
  - The `cytoflow.views` subpackage. This is where visualizations go. These can be traditional visualizations (dotplots, histograms); or statistical summary views (bar plots of population means). There is a significant amount of class hierarchy here – see the documentation for *adding a new view*.

  - The `cytoflow.utility` subpackage. Useful functions and classes, like `geom_mean()`.

- The `cytoflowgui` package. Implements the GUI.
  - The `cytoflowgui.op_plugins` subpackage. Contains instances of `envisage.Plugin` that wrap the operations. See the documentation for *adding a new operation GUI plugin*.
  - The `cytoflowgui.op_views` subpackage. Contains instances of `envisage.Plugin` that wrap the views. See the documentation if you want to *add a new view GUI plugin*.

Design decisions & justifications

- Cytometry analysis as a workflow – an analysis is a set of operations applied sequentially to a dataset. I think this is kind of obvious; it just formalizes the way of doing things that everyone else pretty much already uses.
- Instead of keeping "tubes" or "wells" as first-class objects, represent all the events from all the samples as a big long `pandas.DataFrame`, distinguishing events from different tubes via their varying experimental conditions. Most of my flow analysis experience is with the R Bioconductor package's flowCore, which treats tubes as first-class objects akin to separate microarrays. That's fine if you've got just a few tubes (or a few microarrays), but it rapidly gets cumbersome if you've got multiple plates of samples, each plate of which has two or three experimental variables; I ended up spending more time and code specifying metadata than I did actually doing analysis.

  Cytoflow pushes the metadata down to the event level, doing away entirely with the concept of tubes or wells (after you get your data imported, of course.) This hews much more closely to Hadley Wickham's concept of Tidy Data, and is also (!) much easier to vectorize computations on using `pandas` and `numpy` and `numexpr`. Now, you can access all the events that are, say Dox-induced, by just saying `experiment['Dox']` without having to keep track of which tubes are induced and which weren't.

  Note: If you have tubes that are replicates, just add another experimental condition, perhaps called "replicate". You can specify that condition to the statistics views to get a standard error.

- Gates don't actually subset data (delete or copy it); they just add metadata. I struggled for a long time with the question of how to store and manipulate different subsets of data after gating. Again, my own experience is with Bioconductor's `flowCore`, which defines a tree structure by data that is included or excluded by gates; if a node is a gate, then its children are the subpopulations produced by that gate. Navigating that tree, though, is really difficult, especially if you want to re-combine data after gating (for plotting, for example.)

  Then there was the issue of how to track and manipulate this structure as additional operations were performed. Keep just a single copy and operate on it in-place? Or copy the output of one operation for the input of the next, with the space penalties that implies?

  I finally realized I didn't have to choose; when you copy a `pandas.DataFrame`, you get a "shallow" copy, with the actual data just linked to by reference. This was perfect; if I needed to transform the data from one copy to another, I could just replace the transformed channels; and "gating" events didn't have to create new subsets or containers, it could just add another column specifying the gate membership of each event.

- `cytoflow` discourages wholesale transformation of the underlying data, ie. taking the log of the data set. This is of a part with `cytoflow` enabling *quantitative* analysis – if you want a measure of center of data that is log-normal, you should use the geometric mean instead of log-transforming and taking the arithmetic mean. It is frequently useful to transform data before viewing it, or gating it, etc – those transformations can be passed as parameters to the view modules.

  The obvious exceptions here, of course, are things like bleedthrough correction and calibration using beads. These operations transform the data, but they don't cause the same sorts of shift in data *structure* you see with a log transform. Data that is distributed log-normally before bleedthrough correction, will be distributed log-normally after.

- Easy computation and plotting of summary statistics. The *ChannelStatisticOp* and *FrameStatisticOp* operations create new statistics and add them to the *Experiment.statistics*; and *BarChartView*, *Stats1DView* and *Stats2DView* make it easy to plot them. (A statistic is just a `pandas.Series` with a hierarchical index that encodes data subsets and the value of a summary statistic for each group.) This may be more useful in the GUI, because `pandas.DataFrame.groupby()` provides similar functionality in a notebook setting.
- As is made pretty clear in the example Jupyter notebooks, the semantics for views and operations are
  1. Instantiate a new operation or view
  2. Parameterize the operation or view (possibly by estimating parameters from a provided data set).
  3. Apply the operation or view to an *Experiment*. If applying an operation, `apply()` returns a new Experiment.

  The justification for these semantics is that it makes the *state* of the interacting objects really obvious. An operation or view's state doesn't depend on the data it's applied to; if its parameters do depend on data, those parameters' estimation is a separate operation.

  It also allows for ready separation of the workflow from the data it's applied to, allowing for easy sharing of workflows.
- The module attributes have been replaced by Traits. See the Traits documentation for a good overview, but in short they give Python some of the benefits of statically typed languages like Java, without much of the mess that a fully statically typed language incurs. Their power doesn't see a whole lot of use internal to the cytoflow package, but they make writing the GUI layer a **whole** lot easier.
- The design of the views are strongly influenced by best-in-class statistics visualization packages from R: `lattice` and `ggplot`. If your data is tidy, then each experimental variable you want to plot differently so you can compare them is called a "facet". For example, a facet might be a timepoint or an inducer level (ie an experimental condition); it might also be some metadata added by an operation (ie gate membership or bin). Then, you plot the dataset broken down in various ways by its facets: for example, each timepoint might be put on its own subplot, while each Dox level might be represented by a different color. (Check out the example Jupyter notebook if this is confusing.

Writing new `cytoflow` modules

Creating a new module in *cytoflow* ranges from easy (for simple things) to quite involved. I like to think that *cytoflow* follows the Perl philosophy of making the easy jobs easy and the hard jobs possible.

With that in mind, let's look at the process of creating a new module, progressing from easy to involved.

Basics

All the APIs (both public and internal) are built using Traits. For operations and views in the *cytoflow* package, basic working knowledge of `traits` is sufficient. For GUI work, trait notification is used extensively.

The GUI wrappers also use TraitsUI because it makes wrapping traits with UI elements easy. Have a look at documentation for views, handlers, and of course the trait editors.

Finally, there are some principles that I expect new modules contributed to this codebase to follow:

- **Check for pathological errors and fail early**. I really dislike the tendency of a number of libraries to fail with cryptic errors. (I'm looking at you, `pandas`.) Check for obvious errors and raise a

`CytoflowOpError` or `CytoflowViewError`). If the problem is non-fatal, warn with `CytoflowOpWarning` or `CytoflowViewWarning`. The GUI will also know how to handle these gracefully.

- **Separate experimental data from module state.** There are workflows that require estimating parameters with one data set, then applying those operations to another. Make sure your module supports them.
- **Estimate slow but apply fast.** The GUI re-runs modules' `apply()` methods automatically when parameters change. That means that the `apply()` method must run very quickly.
- **Write tests.** I hate writing unit tests, but they are indispensible for catching bugs. Even in a view's tests are just smoke tests ("It plots something and doesn't crash"), that's better than nothing.

New operations

The base operation API is fairly simple:

- `id` - a required `traits.Constant` containing the UID of the operation
- `friendly_id` - a required `traits.Constant` containing a human-readable name
- `apply()` - takes an `Experiment` and returns a new `Experiment` with the operation applied. `apply()` should `clone()` the old experiment, then modify and return the clone. Don't forget to add the operation to the new `Experiment`'s `history`. A good example of a simple operation is `RatioOp`.

  Note: Be aware of the `deep` parameter for `clone()`! It defaults to `True` – **only** set it to `False` if you are only adding columns to the `Experiment`.

  Note: The resulting `Experiment` must have a `pandas.RangeIndex` for its index – several modules rely on this! If you add or remove events from the `Experiment`, make sure you call `pandas.DataFrame.reset_index` on `Experiment.data` to make the index monotonic again.

- `estimate()` - You may also wish to estimate the operation's parameters from a data set. Crucially, this *might not be the data set you are eventually applying the operation to.* If your operation relies on estimating parameters, implement the `estimate()` function. This may involve selecting a subset of the data in the `Experiment`, or it may involve loading in an an additional FCS file. A good example of the former is `KMeansOp`; a good example of the latter is `AutofluorescenceOp`.

  You may also find that you wish to estimate different parameter sets for different sub-populations (as encoded in the `Experiment`'s `conditions`.) By convention, the conditions that you want to estimate different parameters for are passed using a trait named `by`, which takes a list of conditions and groups the data by unique combinations of those conditions' values before estimating a paramater set for each. Look at `KMeansOp` for an example of this behavior.

- `default_view()` - for some operations, you may want to provide a default view. This view may just be a base view parameterized in a particular way (like the `HistogramView` that is the default view of `BinningOp`), or it may be a visualization of the parameters estimated by the `estimate()` function (like the default view of `AutofluorescenceOp`.) In many cases, the view returned by this function is linked back to the operation that produced it.

New views

The base view API is very simple:

- *id* - a required `traits.Constant` containing the UID of the operation
- *friendly_id* - a required `traits.Constant` containing a human-readable name
- *plot()* - plots `Experiment`.

As I wrote more views, however, I noticed a significant amount of code duplication, which led to bugs and lost time. So, I refactored the view code to use a short hierarchy of classes for particular types of views. You can take advantage of this functionality when writing a new module, or you can simply derive your new view from `traits.HasTraits` and implement the simple API above.

The view base classes are:

- *BaseView* – implements a view with row, column and hue facets. After setting up the facet grid, it calls the derived class's `_grid_plot()` to actually do the plotting. *plot()* also has parameters to set the plot style, legend, axis labels, etc.
- *BaseDataView* – implements a view that plots an *Experiment*'s data (as opposed to a statistic.) Includes functionality for subsetting the data before plotting, and determining axis limits and scales.
- *Base1DView* – implements a 1-dimensional data view. See *HistogramView* for an example.
- *Base2DView* – implements a 2-dimensional data view. See *ScatterplotView* for an example.
- *BaseNDView* – implements an N-dimensional data view. See *RadvizView* for an example.
- *BaseStatisticsView* – implements a view that plots a statistic from an *Experiment* (as opposed to the underlying data.) These views have a "primary" `variable`, and can be subset as well.
- *Base1DStatisticsView* – implements a view that plots one dimension of a statistic. See `BarChartView` for an example.
- *Base2DStatisticsView* – implements a view that plots two dimensions of a statistic. See `Stats2DView` for an example.

New GUI operations

Wrapping an operation for the GUI sometimes feels like it requires more work than writing the operation in the first place. A new operation requires at least five things:

- A class derived from the underlying *cytoflow* operation. The derived operation should be placed in a module in *cytoflowgui.workflow.operations*, and it should:
  - Inherit from `WorkflowOperation` to add support for various GUI event-handling bits (as well as the underlying *cytoflow* class, if appropriate)
  - Override attributes in the underlying *cytoflow* class to add metadata that tells the GUI how to react to changes. (See the *IWorkflowOperation* docstring for details.)
  - Provide an implementation of `get_notebook_code()`, to support exporting to Jupyter notebook.
  - If the module has an `estimate()` method, then implement `clear_estimate()` to clear those parameters.
  - If the module has a `default_view()` method, it should be overridden to return a GUI-enabled view class (see below.)
  - Optionally, override `should_apply()` and `should_clear_estimate()` to only do expensive operations when necessary.

- Serialization logic. *cytoflow* uses camel for sane YAML serialization; a dumper and loader for the class must save and load the operation's parameters. These should also go in *cytoflowgui.workflow.operations*.
- A handler class that defines the default traits.View and provides supporting logic. This class should be derived from *OpHandler* and should be placed in *cytoflowgui.op_plugins*.
- A plugin class derived from envisage.plugin.Plugin and implementing IOperationPlugin. It should also derive from *cytoflowgui.op_plugins.op_plugin_base.PluginHelpMixin*, which adds support for online help.
- Tests. Because of *cytoflowgui*'s split between processes, testing GUI logic for modules can be kind of a synchronization nightmare. This is by design – because the same synchronization issues are present when running the software. See the cytoflowgui/tests directory for (many) examples.
- (Optionally) default view implementations. If the operation has a default view, you should wrap it as well (in the operation plugin module.) See the next section for details.

New GUI views

A new view operation requires at least five things:

- A class derived from the underlying *cytoflow* view. The derived view should be placed in *cytoflowgui.workflow.views*
  - Inherit from *WorkflowView* or one of its children to add support for various GUI event-handling bits
  - Override attributes in the underlying *cytoflow* class to add metadata that tells the GUI how to react to changes. (See the *IWorkflowView* docstring for details.)
  - Provide an implementation of get_notebook_code(), to support exporting to Jupyter notebook.
  - Optionally, override should_plot() to only plot when necessary.
- Serialization logic. *cytoflow* uses camel for sane YAML serialization; a dumper and loader for the class must save and load the operation's parameters. These should also go in *cytoflowgui.workflow.views*.
- A handler class that defines the default traits.View and provides supporting logic. This class should be derived from *ViewHandler* and should be placed in *cytoflowgui.view_plugins*.
- A plugin class derived from envisage.plugin.Plugin and implementing IViewPlugin. It should also derive from cytoflowgui.view_plugins.view_plugin_base.PluginHelpMixin,, which adds support for online help.
- Plot parameters. The parameters to a view's plot() method are stored in an object that derives from BasePlotParams or one of its decendants. Choose data types that are appropriate for the view, and include a default view named view_params_view in the handler class. Don't forget to write serialization code for it as well!
- Tests. Because of *cytoflowgui*'s split between processes, testing GUI logic for modules can be kind of a synchronization nightmare. This is by design – because the same synchronization issues are present when running the software. See the cytoflowgui/tests directory for (many) examples. In the case of a view, most of these are "smoke tests", testing that the view doesn't crash with various sets of parameters.

Note: Why the split between the classes in cytoflowgui.op_modules, *cytoflowgui.workflow.operations*, cytoflowgui.view_modules, and *cytoflowgui.workflow.views*? It's because of the fact that cytoflow runs in two processes – one handles the GUI and the other operates on the workflow. If you load a module containing UI bits, even if you don't explicitly create a QGuiApplication, it starts an event loop. That's why older versions of Cytoflow had two icons in the task bar when running on a Mac. You know how sometimes you go to fix a "little" bug and end up re-writing the whole program? This was one of those times. . . .

Creating multiple plots

TODO: Describe facets / by / enum_plots / plot_name

Also add examples??

Frequently Asked Questions

These are some questions that are commonly asked about Cytoflow's modules, especially by new users.

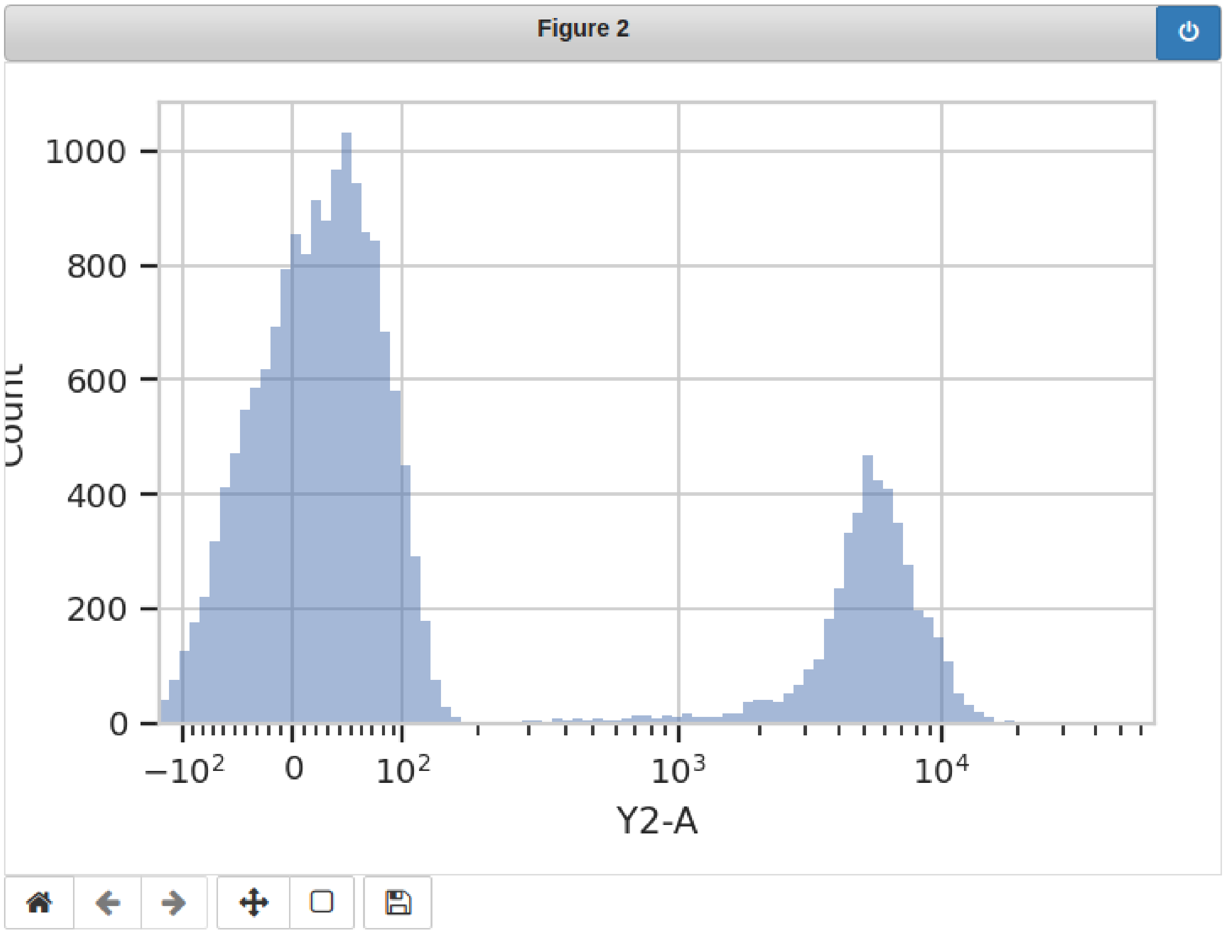
Cytoflow seems like it could be useful for analyzing data other than FCS files. Can I do that?

Maybe! Quite a few portions of the package assume that the data is from a flow cytometer. This is quite explicitly baked into the point-and-click GUI; less so with the Python modules. If you want to give it a go (and are using the Python modules), have a look at the `Experiment` class. You'll have to build one of those manually – instead of creating one with `ImportOp` – and you'll obviously need to avoid modules that explicitly expect other FCS files to parameterize them. But otherwise, once you've got a fully-formed Experiment, you should (probably) be fine to use (most of) the rest of the package(?) Let me know how you get on.

How do I save the plots I've made with the Python modules?

cytoflow uses the `matplotlib.pyplot` stateful interface for making plots. Thus, there are two ways to save the plots. The first is to use the `matplotlib.pyplot.savefig()` – see the `matplotlib` documentation for more details. (This option gives you the greatest flexibility in format, resolution, etc.)

If you are using the Jupyter notebook, the second is to replace the `%matplotlib inline` magic with `%matplotlib notebook`. Then, when you make a plot, the plot remains interactive. Here's an example:



Click the "disk" icon to download a copy of the image.

cytoflow package

cytoflow

`cytoflow` is a package for quantitative, reproducible analysis of flow cytometry data.

Written by Brian Teague, bpteague@gmail.com

Copyright Massachusetts Institute of Technology 2015-2018

Copyright Brian Teague 2018-2021

Subpackages

cytoflow.operations package

cytoflow.operations

This package contains all *cytoflow* operations – classes implementing *IOperation* whose *IOperation.apply* function takes an *Experiment* and returns an *Experiment*.

Submodules

cytoflow.operations.autofluorescence

The *cytoflow.operations.autofluorescence* module contains two classes:

AutofluorescenceOp – corrects an *Experiment* for autofluorescence

AutofluorescenceDiagnosticView – a diagnostic to make sure that *AutofluorescenceOp* estimated its parameters correctly.

`class cytoflow.operations.autofluorescence.`**`AutofluorescenceOp`**

Bases: `traits.has_traits.HasStrictTraits`

Apply autofluorescence correction to a set of fluorescence channels.

The *estimate* function loads a separate FCS file (not part of the input *Experiment*) and computes the untransformed median and standard deviation of the blank cells. Then, *apply* subtracts the median from the experiment data.

To use, set the *blank_file* property to point to an FCS file with unstained or nonfluorescing cells in it; set the *channels* property to a list of channels to correct.

apply also adds the `af_median` and `af_stdev` metadata to the corrected channels, representing the median and standard deviation of the measured blank distributions.

`channels`

The channels to correct.

Type List(Str)

`blank_file`

The filename of a file with "blank" cells (not fluorescent). Used to *estimate* the autofluorescence.

Type File

`blank_file_conditions`

Occasionally, you'll need to specify the experimental conditions that the blank tube was collected under (to apply the operations in the history.) Specify them here.

Type Dict

Examples

Create a small experiment:

```
>>> import cytoflow as flow
>>> import_op = flow.ImportOp()
>>> import_op.tubes = [flow.Tube(file = "tasbe/rby.fcs")]
>>> ex = import_op.apply()
```

Create and parameterize the operation

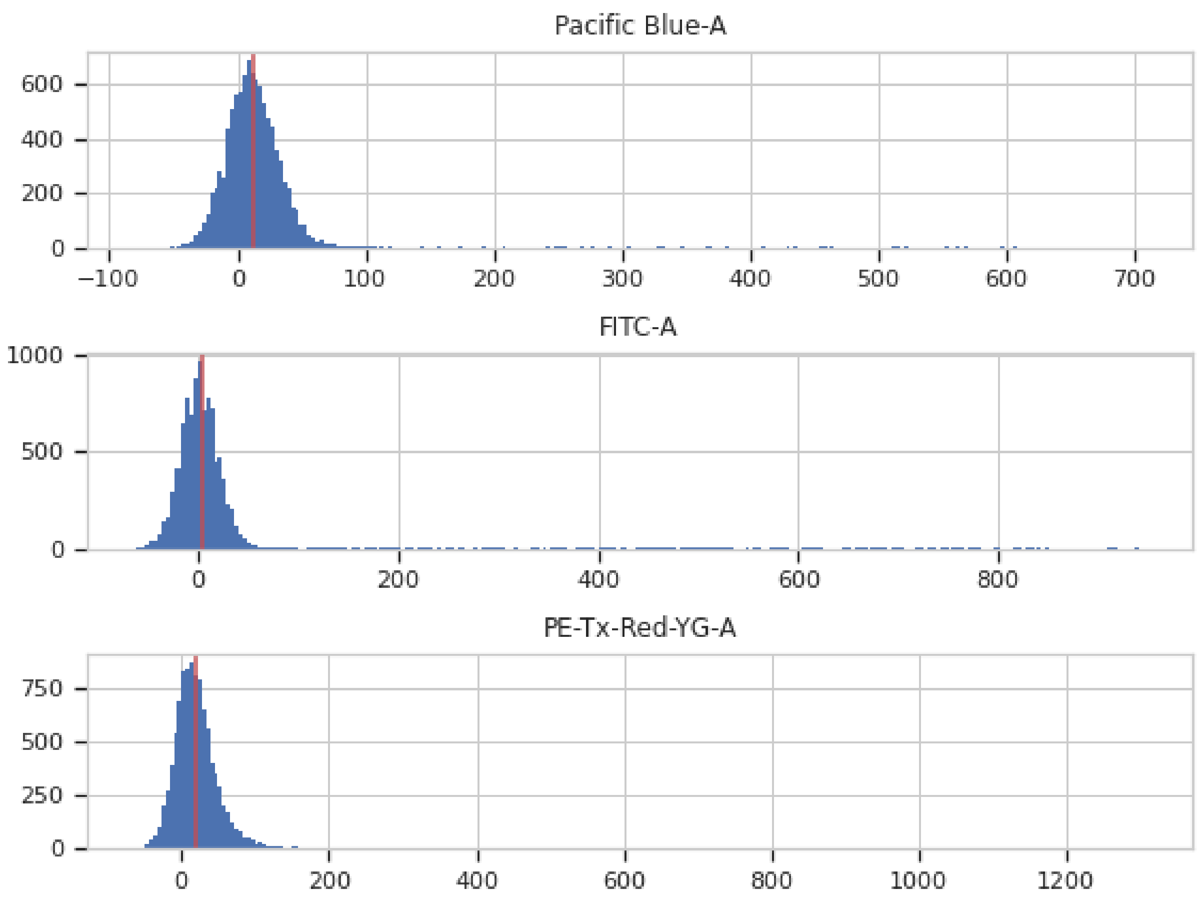
```
>>> af_op = flow.AutofluorescenceOp()
>>> af_op.channels = ["Pacific Blue-A", "FITC-A", "PE-Tx-Red-YG-A"]
>>> af_op.blank_file = "tasbe/blank.fcs"
```

Estimate the model parameters

```
>>> af_op.estimate(ex)
```

Plot the diagnostic plot

```
>>> af_op.default_view().plot(ex)
```



Apply the operation to the experiment

```
>>> ex2 = af_op.apply(ex)
```

estimate(*experiment*, *subset=None*)
Estimate the autofluorescence from *blank_file* in channels specified in *channels*.

Parameters

- **experiment** (*Experiment*) – The experiment to which this operation is applied
- **subset** (*Str (default = "")*) – An expression that specifies the events used to compute the autofluorescence

apply(*experiment*)
Applies the autofluorescence correction to channels in an experiment.

Parameters **experiment** (*Experiment*) – the experiment to which this op is applied

Returns

a new experiment with the autofluorescence median subtracted. The corrected channels have the following metadata added to them:

- **af_median** : Float The median of the non-fluorescent distribution
- **af_stdev** : Float The standard deviation of the non-fluorescent distribution

Return type *Experiment*

default_view(***kwargs*)
Returns a diagnostic plot to see if the autofluorescence estimation is working.

Returns An diagnostic view, call *AutofluorescenceDiagnosticView.plot* to see the diagnostic plots

Return type *IView*

class cytoflow.operations.autofluorescence.**AutofluorescenceDiagnosticView**
Bases: `traits.has_traits.HasStrictTraits`

Plots a histogram of each channel, and its median in red. Serves as a diagnostic for the autofluorescence correction.

op
The *AutofluorescenceOp* whose parameters we're viewing. Set automatically if you created the instance using *AutofluorescenceOp.default_view*.

Type Instance(*AutofluorescenceOp*)

plot(*experiment*, ***kwargs*)
Plot a faceted histogram view of a channel

cytoflow.operations.base_op_views

Base classes for *IOperation* default views:

OpView – a view that has an operation, *OpView.op*, as an attribute.

Op1DView – an *OpView* that has a *Op1DView.channel* attribute and its attendant *Op1DView.scale*. This class overrides *Base1DView* to delegate those attributes to *OpView.op*.

Op2DView – an *OpView* that has *Op2DView.xchannel* (and *Op2DView.xscale*) and *Op2DView.ychannel* (and *Op2DView.yscale*). This class overrides *Base2DView* to delegate those attributes to *OpView.op*.

ByView – an *OpView* that can plot various plots depending on what is passed to *ByView.plot*'s `plot_name` parameter.

AnnotatingView – An *IView* that plots an underlying data plot, then plots some annotations on top of it.

`class cytoflow.operations.base_op_views.`**`OpView`**

Bases: *cytoflow.views.base_views.BaseDataView*

`op`

The *IOperation* that this view is associated with. If you created the view using *default_view*, this is already set.

Type Instance(*IOperation*)

`class cytoflow.operations.base_op_views.`**`Op1DView`**

Bases: *cytoflow.operations.base_op_views.OpView*, *cytoflow.views.base_views.Base1DView*

`channel`

The channel this view is viewing. If you created the view using *default_view*, this is already set.

Type Str

`scale`

The way to scale the x axes. If you created the view using *default_view*, this may be already set.

Type {'linear', 'log', 'logicle'}

`class cytoflow.operations.base_op_views.`**`Op2DView`**

Bases: *cytoflow.operations.base_op_views.OpView*, *cytoflow.views.base_views.Base2DView*

`xchannel`

The channels to use for this view's X axis. If you created the view using *default_view*, this is already set.

Type Str

`ychannel`

The channels to use for this view's Y axis. If you created the view using *default_view*, this is already set.

Type Str

`xscale`

The way to scale the x axis. If you created the view using *default_view*, this may be already set.

Type {'linear', 'log', 'logicle'}

`yscale`

The way to scale the y axis. If you created the view using *default_view*, this may be already set.

Type {'linear', 'log', 'logicle'}

`class cytoflow.operations.base_op_views.`**`ByView`**

Bases: *cytoflow.operations.base_op_views.OpView*

A view that can plot various plots based on the `plot_name` parameter of *plot*.

`facets`

A read-only list of the conditions used to facet this view.

Type List(Str)

`by`

A read-only list of the conditions used to group this view's data before plotting.

Type List(Str)

`enum_plots`(*experiment*)
Returns an iterator over the possible plots that this View can produce. The values returned can be passed to the `plot_name` keyword of *`plot`*.

Parameters **experiment** (*`Experiment`*) – The *`Experiment`* that will be producing the plots.

`plot`(*experiment*, ***kwargs*)
Make the plot.

Parameters **plot_name** (*Str*) – If this *`IView`* can make multiple plots, `plot_name` is the name of the plot to make. Must be one of the values retrieved from *`enum_plots`*.

`class cytoflow.operations.base_op_views.By1DView`
Bases: *`cytoflow.operations.base_op_views.ByView`*, *`cytoflow.operations.base_op_views.Op1DView`*

`class cytoflow.operations.base_op_views.By2DView`
Bases: *`cytoflow.operations.base_op_views.ByView`*, *`cytoflow.operations.base_op_views.Op2DView`*

`class cytoflow.operations.base_op_views.NullView`
Bases: *`cytoflow.views.base_views.BaseDataView`*

An *`IView`* that doesn't actually do any plotting.

`class cytoflow.operations.base_op_views.AnnotatingView`
Bases: *`cytoflow.views.base_views.BaseDataView`*

A *`IView`* that plots an underlying data plot, then plots some annotations on top of it. See *`gaussian.GaussianMixture1DView`* for an example. By default, it assumes that the annotations are to be plotted in the same color as the view's *`huefacet`*, and sets *`huefacet`* accordingly if the annotation isn't already set to a different facet.

Note: The `annotation_facet` and `annotation_plot` parameters that the *`plot`* method consumes are only for internal use, which is why they're not documented in the *`plot`* docstring.

`plot`(*experiment*, ***kwargs*)

Parameters **color** (*matplotlib color*) – The color to plot the annotations. Overrides the default color cycle.

cytoflow.operations.bead_calibration

The *`bead_calibration`* module contains two classes:

`BeadCalibrationOp` – calibrates the raw measurements in a *`Experiment`* using fluorescent particles.

`BeadCalibrationDiagnostic` – a diagnostic view to make sure that *`BeadCalibrationOp`* correctly estimated its parameters.

`class cytoflow.operations.bead_calibration.BeadCalibrationOp`
Bases: `traits.has_traits.HasStrictTraits`

Calibrate arbitrary channels to molecules-of-fluorophore using fluorescent beads (eg, the Spherotech RCP-30-5A rainbow beads.)

Computes a log-linear calibration function that maps arbitrary fluorescence units to physical units (ie molecules equivalent fluorophore, or *MEF*).

To use, set `beads_file` to an FCS file containing events collected *using the same cytometer settings as the data you're calibrating*. Specify which beads you ran by setting `beads` to match one of the values of `BeadCalibrationOp.BEADS`; and set `units` to which channels you want calibrated and in which units. Then, call `estimate` and check the peak-finding with `BeadCalibrationDiagnostic.plot`. If the peak-finding is wacky, try adjusting `bead_peak_quantile` and `bead_brightness_threshold`. When the peaks are successfully identified, call `apply` to scale your experimental data set.

If you can't make the peak finding work, please submit a bug report!

This procedure works best when the beads file is very clean data. It does not do its own gating (maybe a future addition?) In the meantime, I recommend gating the *acquisition* on the FSC/SSC channels in order to get rid of debris, cells, and other noise.

Finally, because you can't have a negative number of fluorescent molecules (MEFLs, etc) (as well as for math reasons), this module filters out negative values.

units
A dictionary specifying the channels you want calibrated (keys) and the units you want them calibrated in (values). The units must be keys of the `beads` attribute.

Type Dict(Str, Str)

beads_file
A file containing the FCS events from the beads.

Type File

beads
The beads' characteristics. Keys are calibrated units (ie, MEFL or MEAP) and values are ordered lists of known fluorophore levels. Common values for this dict are included in `BeadCalibrationOp.BEADS`.

Type Dict(Str, List(Float))

bead_peak_quantile
The quantile threshold used to choose bead peaks.

Type Int (default = 80)

bead_brightness_threshold
How bright must a bead peak be to be considered?

Type Float (default = 100)

bead_brightness_cutoff
If a bead peak is above this, then don't consider it. Takes care of clipping saturated detection. Defaults to 70% of the detector range.

Type Float

bead_histogram_bins
The number of bins to use in computing the bead histogram. Tweak this if the peak find is having difficulty, or if you have a small number of events

Type Int (default = 512)

force_linear
A linear fit in log space doesn't always go through the origin, which means that the calibration function isn't strictly a multiplicative scaling operation. Set `force_linear` to force the such behavior. Keep an eye on the diagnostic plot, though, to see how much error you're introducing!

Type Bool (default = False)

Notes

The peak finding is rather sophisticated.

For each channel, a 256-bin histogram is computed on the log-transformed bead data, and then the histogram is smoothed with a Savitzky-Golay filter (with a window length of 5 and a polynomial order of 1).

Next, a wavelet-based peak-finding algorithm is used: it convolves the smoothed histogram with a series of wavelets and looks for relative maxima at various length-scales. The parameters of the smoothing algorithm were arrived at empircally, using beads collected at a wide range of PMT voltages.

Finally, the peaks are filtered by height (the histogram bin has a quantile greater than *bead_peak_quantile*) and intensity (brighter than *bead_brightness_threshold*).

How to convert from a series of peaks to mean equivalent fluorochrome? If there's one peak, we assume that it's the brightest peak. If there are two peaks, we assume they're the brightest two. If there are `n >=3` peaks, we check all the contiguous `n`-subsets of the bead intensities and find the one whose linear regression (in log space!) has the smallest norm (square-root sum-of-squared-residuals.)

There's a slight subtlety in the fact that we're performing the linear regression in log-space: if the relationship in log10-space is `Y=aX + b`, then the same relationship in linear space is `x = 10**X`, `y = 10**y`, and `y = (10**b) * (x ** a)`.

Note: Adding a new set of beads is easy! Please don't add them directly to *BEADS*, though – instead, add them to `beads.csv`, then run the *cytoflow.scripts.parse_beads* script to convert it into a dict.

Examples

Create a small experiment:

```
>>> import cytoflow as flow
>>> import_op = flow.ImportOp()
>>> import_op.tubes = [flow.Tube(file = "tasbe/rby.fcs")]
>>> ex = import_op.apply()
```

Create and parameterize the operation

```
>>> bead_op = flow.BeadCalibrationOp()
>>> beads = 'RCP-30-5A Lot AA01, AA02, AA03, AA04, AB01, AB02, AC01 & GAA01-R'
>>> bead_op.beads = flow.BeadCalibrationOp.BEADS[beads]
>>> bead_op.units = {"Pacific Blue-A" : "MEBFP",
...                  "FITC-A" : "MEFL",
...                  "PE-Tx-Red-YG-A" : "MEPTR"}
>>>
>>> bead_op.beads_file = "tasbe/beads.fcs"
```

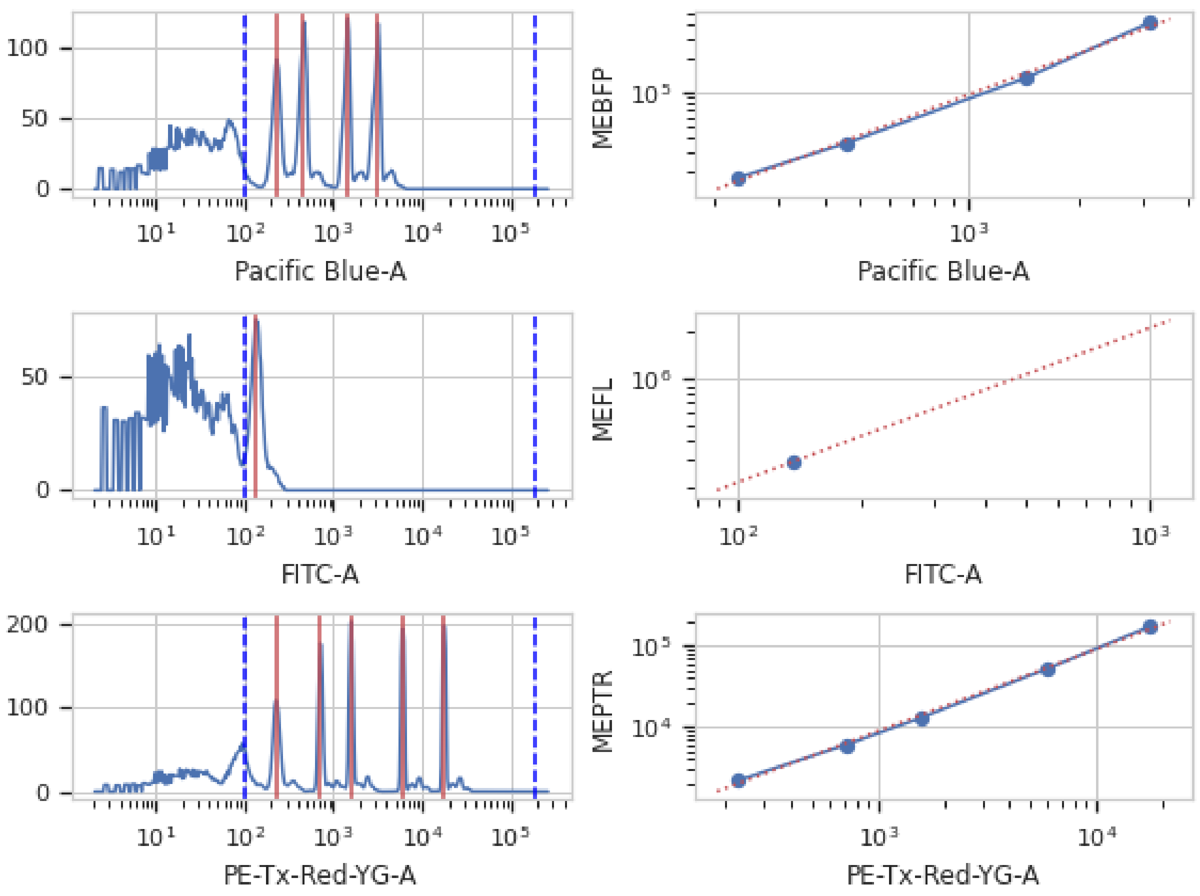
Estimate the model parameters

```
>>> bead_op.estimate(ex)
```

Plot the diagnostic plot

```
>>> bead_op.default_view().plot(ex)
```

Apply the operation to the experiment


100
50
0
Pacific Blue-A
MEBFP
10^5
10^3
Pacific Blue-A
50
0
FITC-A
MEFL
10^6
10^2
10^3
FITC-A
200
100
0
PE-Tx-Red-YG-A
MEPTR
10^5
10^4
10^3
10^4
PE-Tx-Red-YG-A
10^1
10^2
10^3
10^4
10^5

```
>>> ex = bead_op.apply(ex)
```

`estimate`(*experiment*)

Estimate the calibration coefficients from the beads file.

Parameters **experiment** (*Experiment*) – The experiment used to compute the calibration.

`apply`(*experiment*)

Applies the bleedthrough correction to an experiment.

Parameters **experiment** (*Experiment*) – the experiment to which this operation is applied

Returns

A new experiment with the specified channels calibrated in physical units. The calibrated channels also have new metadata:

- **bead_calibration_fn** [Callable (`pandas.Series` –> `pandas.Series`)] The function to calibrate raw data to bead units
- **bead_units** [Str] The units this channel was calibrated to

Return type *Experiment*

`default_view`(***kwargs*)

Returns a diagnostic plot to see if the peak finding is working.

Returns An diagnostic view, call *BeadCalibrationDiagnostic.plot* to see the diagnostic plots

Return type *IView*

```
BEADS = {'ACP 30-2K': {'MEAP': [7880.0, 17400.0, 53100.0, 130000.0, 208000.0]},
'RCP-30-5 Lot AA01, AB01, AB02, AC01 & AD01':  {'MEAP': [6720.0, 17962.0, 30866.0,
51704.0, 146080.0], 'MECY': [12901.0, 36837.0, 76621.0, 261671.0, 1069858.0],
'MEFL': [6028.0, 17493.0, 35674.0, 126907.0, 290983.0], 'MEPE': [4974.0, 13118.0,
26757.0, 94930.0, 250470.0], 'MEPTR': [2198.0, 6063.0, 12887.0, 51686.0, 170219.0]},
'RCP-30-5 Lot AM02, AM01, AL01, AH01, AG01, AF01 & AD03':  {'MEA700':
[7850.64102564103, 26574.358974359, 87058.3333333333, 232771.153846154,
428898.076923077], 'MEA750':  [643.319414830418, 1682.85038466727, 4905.43254465399,
15637.5278759487, 44664.1560269625], 'MEAP': [2395.0, 8273.0, 27652.0, 75669.0,
145428.0], 'MECY5.5':  [8737.0, 28177.0, 93996.0, 334087.0, 1023447.0], 'MEFL':
[4447.0, 14227.0, 46322.0, 133924.0, 276897.0], 'MEKO': [4100.1862606147,
13503.2020463521, 45019.6494865316, 152691.355293162, 391037.895005993], 'MEPB':
[902.826150264251, 2543.25777035562, 7577.3103315121, 19902.1552759773,
35490.1854343577], 'MEPCY7':  [4447.52977431211, 12826.6344100731, 41676.365575392,
149440.701418692, 474797.252391984], 'MEPE': [3236.0, 10754.0, 34842.0, 104483.0,
245894.0], 'METR': [1486.0, 5112.0, 17664.0, 60371.0, 179787.0]}, 'RCP-30-5A
(Euroflow) Lot EAK01, EAG01, EAE01 & EAF01':  {'MEAP': [1366.0, 4706.0, 14822.0,
58161.0, 140920.0, 204810.0, 280646.0], 'MEAPCY7':  [12540.2912310804,
27790.424415523, 57114.7530172427, 137792.795419592, 245420.419831163,
312870.988997371, 387104.388350457], 'MEAXL700':  [1180.89572037678,
3930.96580785077, 11270.6100091203, 45502.2217551306, 109122.081384077,
151781.075700287, 237988.013854553], 'MEBFP': [2584.06901535744, 5291.27272439487,
11909.3205931143, 38804.9157590269, 107470.790614424, 359515.748179449,
736906.777253776], 'MECSB': [3827.07306877147, 6410.85503814999, 12405.91715676,
37097.3178176766, 92487.7710023634, 267142.304767532, 499972.639106986], 'MECY':
[1276.0, 3603.0, 9520.0, 35518.0, 102755.0, 328185.0, 1052496.0], 'MEFL': [806.0,
2159.0, 5640.0, 19900.0, 52630.0, 172155.0, 345870.0], 'MEPCY7':  [9726.6114174626,
25173.4200339912, 57085.1373522142, 136728.333019744, 324682.702797253], 'MEPE':
[409.0, 1250.0, 3428.0, 12229.0, 34294.0, 113118.0, 256134.0], 'MEPTR': [171.0,
538.0, 1514.0, 5659.0, 16972.0, 64615.0, 183897.0]}, 'RCP-30-5A (Euroflow) Lot EAM02
& EAM01':  {'MEAP': [736.0, 1892.0, 4804.0, 14248.0, 42425.0, 113026.0, 227044.0],
'MEAPCY7':  [558.0, 1417.0, 3224.0, 8309.0, 20779.0, 48391.0, 89183.0], 'MEAXL700':
[682.0, 1276.0, 2352.0, 4724.0, 9284.0, 17217.0, 26930.0], 'MEBFP': [1295.0, 2398.0,
4884.0, 13920.0, 34027.0, 95282.0, 172500.0], 'MECSB': [480.0, 698.0, 1192.0,
2947.0, 6546.0, 15893.0, 27382.0], 'MECY': [998.0, 3064.0, 8614.0, 28030.0, 90459.0,
298273.0, 849431.0], 'MEFL': [789.0, 1896.0, 4872.0, 15619.0, 47116.0, 143912.0,
333068.0], 'MEPCY7':  [106.0, 343.0, 1088.0, 4215.0, 15949.0, 62905.0, 208319.0],
'MEPE': [507.0, 1204.0, 3171.0, 10440.0, 34385.0, 114122.0, 311207.0], 'MEPTR':
[187.0, 543.0, 1536.0, 5423.0, 17825.0, 63989.0, 207649.0]}, 'RCP-30-5A Lot AA01,
AA02, AA03, AA04, AB01, AB02, AC01 & GAA01-R': {'MEAP': [578.0, 2433.0, 6720.0,
17962.0, 30866.0, 51704.0, 146080.0], 'MEAPCY7':  [718.0, 1920.0, 5133.0, 9324.0,
14210.0, 26735.0], 'MEBFP': [700.0, 1705.0, 4262.0, 17546.0, 35669.0, 133387.0,
412089.0], 'MECSB': [179.0, 400.0, 993.0, 3203.0, 6083.0, 17777.0, 36331.0], 'MECY':
[1437.0, 4693.0, 12901.0, 36837.0, 76621.0, 261671.0, 1069858.0], 'MEFL': [692.0,
2192.0, 6028.0, 17493.0, 35674.0, 126907.0, 290983.0], 'MEPCY7':  [32907.0,
107787.0, 503797.0], 'MEPE': [505.0, 1777.0, 4974.0, 13118.0, 26757.0, 94930.0,
250470.0], 'MEPTR': [207.0, 750.0, 2198.0, 6063.0, 12887.0, 51686.0, 170219.0]},
'RCP-30-5A Lot AC02, AC03 & AD01':  {'MEAP': [1184.0, 4024.0, 8892.0, 20919.0,
35336.0, 62371.0, 160356.0], 'MEAPCY7':  [1638.44517911143, 4232.87970031999,
8021.74368319802, 14091.8060342384, 27002.3850136509], 'MEAmC': [689.818391446723,
1645.66379392412, 4353.99728581043, 17956.9358902402, 37196.3751279619,
136964.361513719, 422642.751819192], 'MECY': [1434.98342378079, 4725.12552939372,
12977.4579084514, 34975.0986132135, 71893.2990690699, 258366.215367105,
1120607.11147369], 'MEFL': [694.316757280561, 2095.54714141499, 6077.97998121964,
18079.7185767491, 37488.3221945023, 127548.70589779, 292408.003430039], 'MEPB':
[179.490842861769, 399.130271822164, 1004.33041842402, 3326.09220035309,
6280.9781554728, 18181.900886873, 37446.4126347345], 'MEPCY7':  [14858.8422415397,
31336.9778664833, 107223.701409309, 519777.268051615], 'MEPE': [525.988079665053,
1735.72870920068, 5023.48980798513, 13251.1367386049, 27327.3892526052,
96552.680798141, 259880.668965796], 'MEPTR': [218.12305864236, 737.206689077331,
```

are:

- Spherotech ACP 30-2K
- Spherotech RCP-30-5A Lot AN04, AN03, AN02, AN01, AM02, AM01, AL01, AK04, AK03 & AK02**
- Spherotech RCP-30-5A (Euroflow) Lot EAM02 & EAM01
- Sphreotech RCP-30-5A (Euroflow) Lot EAK01, EAG01, EAE01 & EAF01
- Spherotech RCP-30-5A Lot AK01, AJ01, AH02, AH01, AF02, AF01, AD04 & AE01
- Spherotech RCP-30-5A Lot AG01
- Spherotech RCP-30-5A Lot AA01, AA02, AA03, AA04, AB01, AB02, AC01 & GAA01-R
- Spherotech RCP-30-5A Lot AC02, AC03 & AD01
- Spherotech RCP-30-5A Lot Z02 and Z03
- Spherotech RCP-30-5 Lot AA01, AB01, AB02, AC01 & AD01
- Spherotech RCP-30-5 Lot AM02, AM01, AL01, AH01, AG01, AF01 & AD03
- Spherotech RCP-60-5
- Spherotech URCP 38-2K
- Spherotech URCP 38-2K Lot AN01, AM01, AL02, AL01, AK03, AK02, AK01, AJ02 & AJ03
- Spherotech URCP 50-2K Lot AM01 & AJ01
- Spherotech URCP 50-2K

The Spherotech fluorophores labels and the laser / filter sets (that I know about) are:

- **MECSB** (Cascade Blue, 405 –> 450/50)
- **MEBFP** (BFP, 405 –> 530/40)
- **MEFL** (Fluroscein, 488 –> 530/40)
- **MEPE** (Phycoerythrin, 488 –> 575/25)
- **MEPTR** (PE-Texas Red, 488 –> 613/20)
- **MECY** (Cy5, 488 –> 680/30)
- **MEPCY7** (PE-Cy7, 488 –> 750 LP)
- **MEAP** (APC, 633 –> 665/20)
- **MEAPCY7** (APC-Cy7, 635 –> 750 LP)

`class cytoflow.operations.bead_calibration.BeadCalibrationDiagnostic`

Bases: `traits.has_traits.HasStrictTraits`

A diagnostic view for *BeadCalibrationOp*.

Plots the smoothed histogram of the bead data; the peak locations; a scatter plot of the raw bead fluorescence values vs the calibrated unit values; and a line plot of the model that was computed. Make sure that the relationship is linear; if it's not, it likely isn't a good calibration!

op

The operation instance whose parameters we're plotting. Set automatically if you created the instance using *BeadCalibrationOp.default_view*.

Type Instance(*BeadCalibrationOp*)

plot(*experiment*)
Plots the diagnostic view.

Parameters **experiment** (*Experiment*) – The experiment used to create the diagnostic plot.

cytoflow.operations.binning

binning has two classes:

BinningOp – divides events in a channel into bins of equal width (after applying an optional scale)

BinningView – a default view to display the bins.

class cytoflow.operations.binning.**BinningOp**
Bases: `traits.has_traits.HasStrictTraits`

Bin data along an axis.

This operation creates equally spaced bins (in linear or log space) along an axis and adds a condition assigning each event to a bin. The value of the event's condition is the left end of the bin's interval in which the event is located.

name
The operation name. Used to name the new metadata field in the experiment that's created by apply()

Type Str

channel
The name of the channel along which to bin.

Type Str

scale
Make the bins equidistant along what scale?

Type {"linear", "log", "logicle"}

bin_width
The width of the bins. If *scale* is `log`, *bin_width* is in log-10 units; if *scale* is `logicle`, an error is thrown because the units are ill-defined.

Type Float

Examples

Create a small experiment:

```
>>> import cytoflow as flow
>>> import_op = flow.ImportOp()
>>> import_op.tubes = [flow.Tube(file = "tasbe/rby.fcs")]
>>> ex = import_op.apply()
```

Create and parameterize the operation

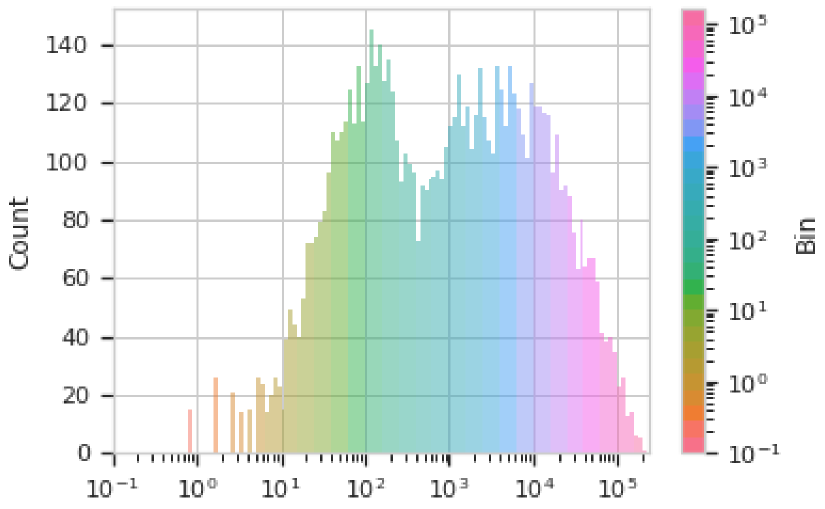
```
>>> bin_op = flow.BinningOp()
>>> bin_op.name = "Bin"
>>> bin_op.channel = "FITC-A"
>>> bin_op.scale = "log"
>>> bin_op.bin_width = 0.2
```

Apply the operation to the experiment

```
>>> ex2 = bin_op.apply(ex)
```

Plot the result

```
>>> bin_op.default_view().plot(ex2)
```



apply(*experiment*)
Applies the binning to an experiment.

Parameters **experiment** (*Experiment*) – the old experiment to which this op is applied

Returns A new experiment with a condition column named `name`, which contains the location of the left-most edge of the bin that the event is in.

Return type *Experiment*

default_view(***kwargs*)
Returns a diagnostic plot to check the binning.

Returns An view instance, call `plot()` to plot the bins.

Return type *IView*

class cytoflow.operations.binning.**BinningView**
Bases: *cytoflow.operations.base_op_views.Op1DView*, *cytoflow.operations.base_op_views.AnnotatingView*, *cytoflow.views.histogram.HistogramView*

Plots a histogram of the current binning op. By default, the different bins are shown in different colors.

channel
The channel this view is viewing. If you created the view using `default_view`, this is already set.

Type Str

scale
The way to scale the x axes. If you created the view using `default_view`, this may be already set.

Type {'linear', 'log', 'logicle'}

`op`
The *`IOperation`* that this view is associated with. If you created the view using *`default_view`*, this is already set.

Type Instance(*`IOperation`*)

`subset`
An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

`xfacet`
Set to one of the *`Experiment.conditions`* in the *`Experiment`*, and a new column of subplots will be added for every unique value of that condition.

Type String

`yfacet`
Set to one of the *`Experiment.conditions`* in the *`Experiment`*, and a new row of subplots will be added for every unique value of that condition.

Type String

`huefacet`
Set to one of the *`Experiment.conditions`* in the in the *`Experiment`*, and a new color will be added to the plot for every unique value of that condition.

Type String

`huescale`
How should the color scale for *`huefacet`* be scaled?

Type {'linear', 'log', 'logicle'}

`plot`(*experiment*, ***kwargs*)
Plot the histogram.

Parameters

- **experiment** (*Experiment*) – The *`Experiment`* to plot using this view.
- **title** (*str*) – Set the plot title
- **xlabel** (*str*) – Set the X axis label
- **ylabel** (*str*) – Set the Y axis label
- **huelabel** (*str*) – Set the label for the hue facet (in the legend)
- **legend** (*bool*) – Plot a legend for the color or hue facet? Defaults to `True`.
- **sharex** (*bool*) – If there are multiple subplots, should they share X axes? Defaults to `True`.
- **sharey** (*bool*) – If there are multiple subplots, should they share Y axes? Defaults to `True`.
- **row_order** (*list*) – Override the row facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **col_order** (*list*) – Override the column facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **hue_order** (*list*) – Override the hue facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **height** (*float*) – The height of *each row* in inches. Default = 3.0

- **aspect** (*float*) – The aspect ratio *of each subplot*. Default = 1.5
- **col_wrap** (*int*) – If `xfacet` is set and `yfacet` is not set, you can "wrap" the subplots around so that they form a multi-row grid by setting this to the number of columns you want.
- **sns_style** (*{"darkgrid", "whitegrid", "dark", "white", "ticks"}*) – Which `seaborn` style to apply to the plot? Default is `whitegrid`.
- **sns_context** (*{"paper", "notebook", "talk", "poster"}*) – Which `seaborn` context to use? Controls the scaling of plot elements such as tick labels and the legend. Default is `talk`.
- **palette** (*palette name, list, or dict*) – Colors to use for the different levels of the hue variable. Should be something that can be interpreted by `seaborn.color_palette`, or a dictionary mapping hue levels to matplotlib colors.
- **despine** (*Bool*) – Remove the top and right axes from the plot? Default is `True`.
- **min_quantile** (*float (>0.0 and <1.0, default = 0.001)*) – Clip data that is less than this quantile.
- **max_quantile** (*float (>0.0 and <1.0, default = 1.00)*) – Clip data that is greater than this quantile.
- **lim** (*(float, float)*) – Set the range of the plot's data axis.
- **orientation** (*{'vertical', 'horizontal'}*)
- **num_bins** (*int*) – The number of bins to plot in the histogram. Clipped to [100, 1000]
- **histtype** (*{'stepfilled', 'step', 'bar'}*) – The type of histogram to draw. `stepfilled` is the default, which is a line plot with a color filled under the curve.
- **density** (*bool*) – If `True`, re-scale the histogram to form a probability density function, so the area under the histogram is 1.
- **linewidth** (*float*) – The width of the histogram line (in points)
- **linestyle** (*['-' | '–' | '-.' | ':' | "None"]*) – The style of the line to plot
- **alpha** (*float (default = 0.5)*) – The alpha blending value, between 0 (transparent) and 1 (opaque).
- **color** (*matplotlib color*) – The color to plot the annotations. Overrides the default color cycle.

cytoflow.operations.bleedthrough_linear

The *`bleedthrough_linear`* module contains two classes:

`BleedthroughLinearOp` – compensates for spectral bleedthrough in a *`Experiment`* using single-color controls

`BleedthroughLinearDiagnostic` – a diagnostic view to make sure that *`BleedthroughLinearOp`* correctly estimated its parameters.

`class cytoflow.operations.bleedthrough_linear.`**`BleedthroughLinearOp`**

Bases: `traits.has_traits.HasStrictTraits`

Apply matrix-based bleedthrough correction to a set of fluorescence channels.

This is a traditional matrix-based compensation for bleedthrough. For each pair of channels, the user specifies the proportion of the first channel that bleeds through into the second; then, the module performs a matrix multiplication to compensate the raw data.

The module can also estimate the bleedthrough matrix using one single-color control per channel.

This works best on data that has had autofluorescence removed first; if that is the case, then the autofluorescence will be subtracted from the single-color controls too.

To use, set up the *controls* dict with the single color controls; call *estimate* to parameterize the operation; check that the bleedthrough plots look good by calling *BleedthroughLinearDiagnostic.plot* on the *BleedthroughLinearDiagnostic* instance returned by *default_view*; and then *apply* on an *Experiment*.

controls
The channel names to correct, and corresponding single-color control FCS files to estimate the correction splines with. Must be set to use *estimate*.

Type Dict(Str, File)

spillover
The spillover "matrix" to use to correct the data. The keys are pairs of channels, and the values are proportions of spectral overlap. If `("channel1", "channel2")` is present as a key, `("channel2", "channel1")` must also be present. The module does not assume that the matrix is symmetric.

Type Dict(Tuple(Str, Str), Float)

control_conditions
Occasionally, you'll need to specify the experimental conditions that the bleedthrough tubes were collected under (to apply the operations in the history.) Specify them here. The key is the channel name; they value is a dictionary of the conditions (same as you would specify for a *cytoflow.operations.import_op.Tube*)

Type Dict(Str, Dict(Str, Any))

Examples

Create a small experiment:

```
>>> import cytoflow as flow
>>> import_op = flow.ImportOp()
>>> import_op.tubes = [flow.Tube(file = "tasbe/rby.fcs")]
>>> ex = import_op.apply()
```

Correct for autofluorescence

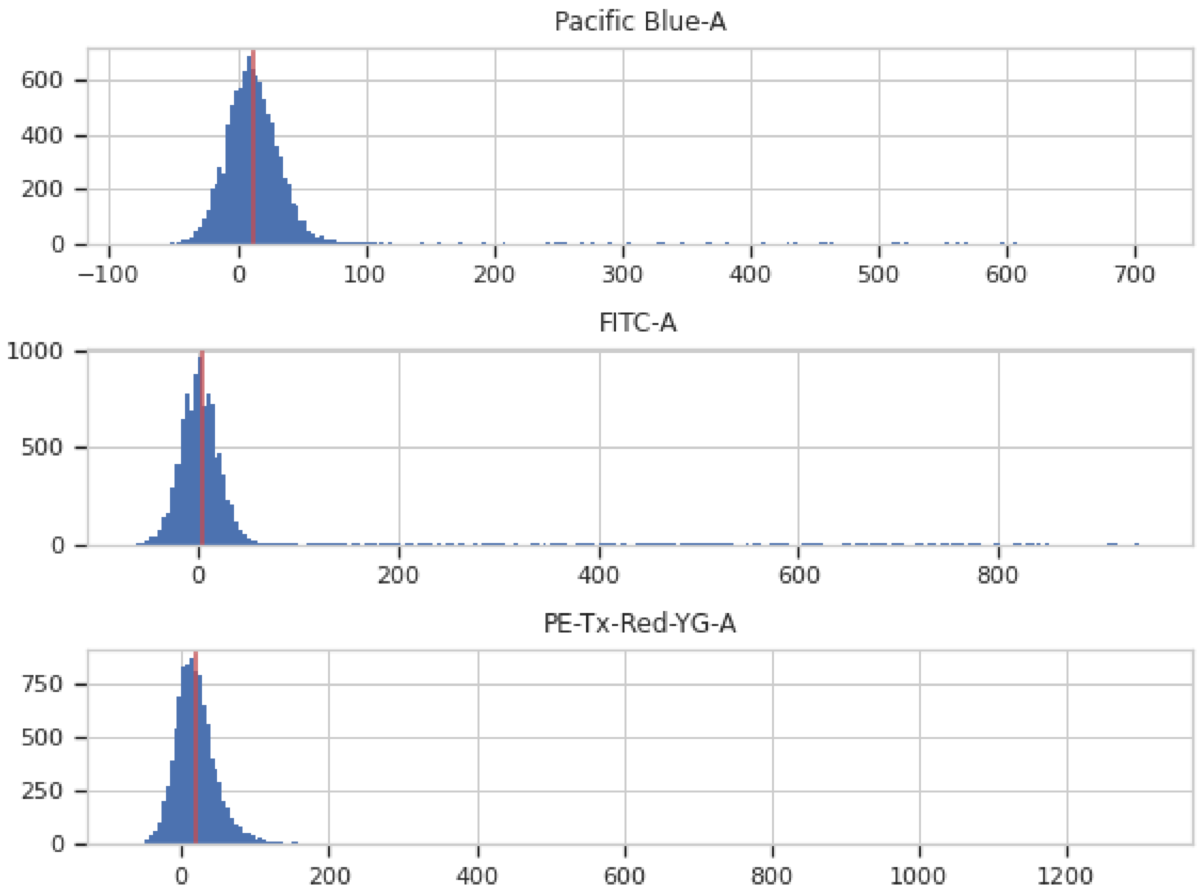
```
>>> af_op = flow.AutofluorescenceOp()
>>> af_op.channels = ["Pacific Blue-A", "FITC-A", "PE-Tx-Red-YG-A"]
>>> af_op.blank_file = "tasbe/blank.fcs"
```

```
>>> af_op.estimate(ex)
>>> af_op.default_view().plot(ex)
```

```
>>> ex2 = af_op.apply(ex)
```

Create and parameterize the operation

```
>>> bl_op = flow.BleedthroughLinearOp()
>>> bl_op.controls = {'Pacific Blue-A' : 'tasbe/ebfp.fcs',
...                   'FITC-A' : 'tasbe/eyfp.fcs',
...                   'PE-Tx-Red-YG-A' : 'tasbe/mkate.fcs'}
```


Pacific Blue-A
0
200
400
600
−100
0
100
200
300
400
500
600
700
FITC-A
0
500
1000
0
200
400
600
800
PE-Tx-Red-YG-A
0
250
500
750
0
200
400
600
800
1000
1200

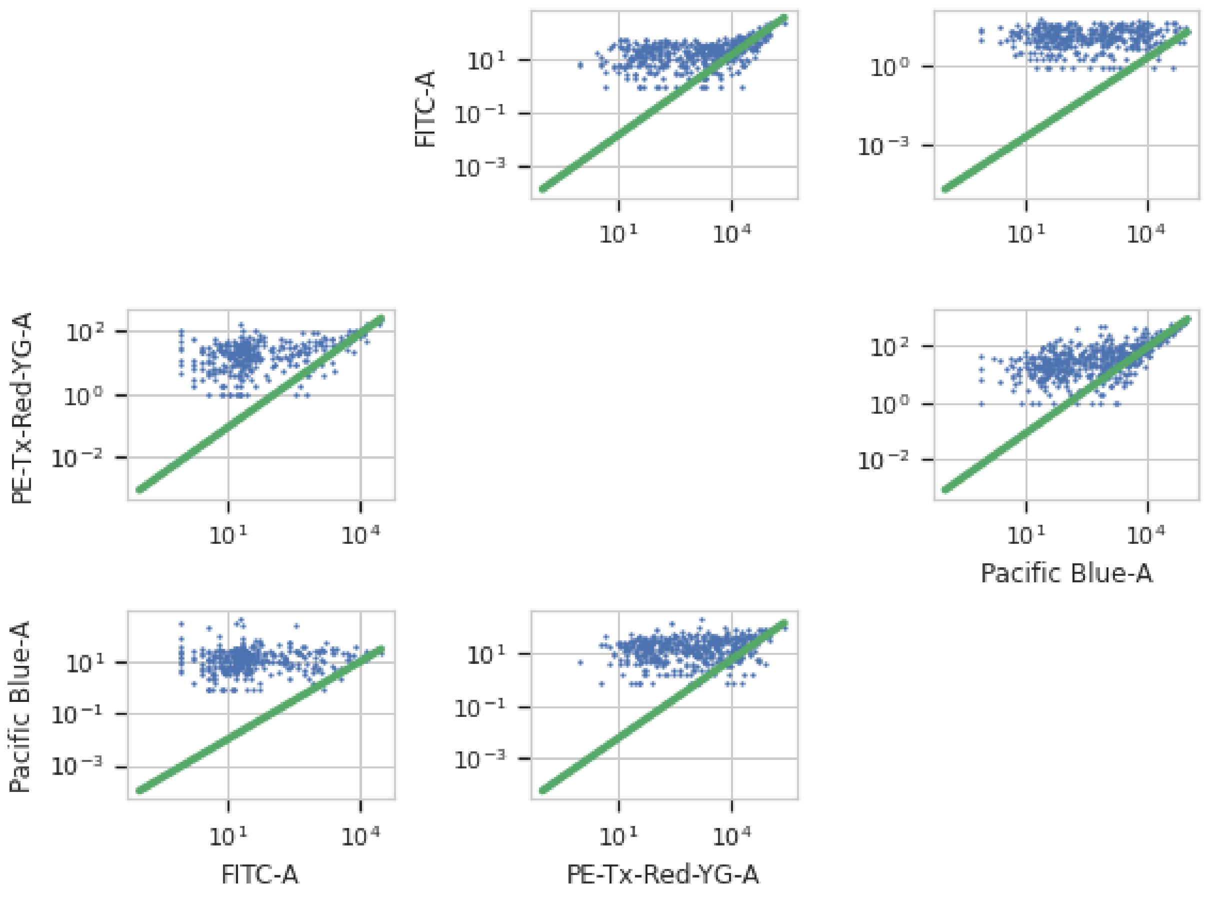
Estimate the model parameters

```
>>> bl_op.estimate(ex2)
```

Plot the diagnostic plot

Note: The diagnostic plots look really bad in the online documentation. They're better in a real-world example, I promise!

```
>>> bl_op.default_view().plot(ex2)
```



Apply the operation to the experiment

```
>>> ex2 = bl_op.apply(ex2)
```

estimate(*experiment*, *subset=None*)

Estimate the bleedthrough from simgle-channel controls in `controls`

apply(*experiment*)

Applies the bleedthrough correction to an experiment.

Parameters **experiment** (`Experiment`) – The experiment to which this operation is applied

Returns

A new `Experiment` with the bleedthrough subtracted out. The corrected channels have the following metadata added:

- **linear_bleedthrough** : Dict(Str : Float) The values for spillover from other channels into this channel.
- **bleedthrough_channels** : List(Str) The channels that were used to correct this one.
- **bleedthrough_fn** : Callable (Tuple(Float) –> Float) The function that will correct one event in this channel. Pass it the values specified in `controls` and it will return the corrected value for this channel.

Return type *Experiment*

`default_view`(***kwargs*)
Returns a diagnostic plot to make sure spillover estimation is working.

Returns An `IView`, call `BleedthroughLinearDiagnostic.plot` to see the diagnostic plots

Return type `IView`

`class cytoflow.operations.bleedthrough_linear.BleedthroughLinearDiagnostic`
Bases: `traits.has_traits.HasStrictTraits`

Plots a scatterplot of each channel vs every other channel and the bleedthrough line

`op`
The operation whose parameters we're viewing. If you made the instance with `BleedthroughLinearOp.default_view`, this is set for you already.

Type Instance(BleedthroughPiecewiseOp)

`subset`
If set, only plot this subset of the underlying data.

Type str

`plot`(*experiment=None*, ***kwargs*)
Plot a diagnostic of the bleedthrough model computation.

cytoflow.operations.channel_stat

Creates a new statistic. `channel_stat` has one class:

`ChannelStatisticOp` – applies a function to subsets of a data set, and adds the resulting statistic to the `Experiment`

`class cytoflow.operations.channel_stat.ChannelStatisticOp`
Bases: `traits.has_traits.HasStrictTraits`

Apply a function to subsets of a data set, and add it as a statistic to the experiment.

The `apply` function groups the data by the variables in `by`, then applies the `function` callable to the `channel` series in each subset. The callable should take a single `pandas.Series` as an argument. The return type is arbitrary, but to be used with the rest of `cytoflow` it should probably be a numeric type or an iterable of numeric types.

`name`
The operation name. Becomes the first element in the `Experiment.statistics` key tuple.

Type Str

`channel`
The channel to apply the function to.

Type Str

`function`
The function used to compute the statistic. *function* must take a `pandas.Series` as its only parameter. The return type is arbitrary, but to be used with the rest of *cytoflow* it should probably be a numeric type or an iterable of numeric types. If *statistic_name* is unset, the name of the function becomes the second in element in the *Experiment.statistics* key tuple.

> **Warning:** Be careful! Sometimes this function is called with an empty input! If this is the case, poorly-behaved functions can return `NaN` or throw an error. If this happens, it will be reported.

Type Callable

`statistic_name`
The name of the function; if present, becomes the second element in the *Experiment.statistics* key tuple. Particularly useful if *function* is a lambda expression.

Type Str

`by`
A list of metadata attributes to aggregate the data before applying the function. For example, if the experiment has two pieces of metadata, `Time` and `Dox`, setting `by = ["Time", "Dox"]` will apply *function* separately to each subset of the data with a unique combination of `Time` and `Dox`.

Type List(Str)

`subset`
A Python expression sent to *Experiment.query* to subset the data before computing the statistic.

Type Str

`fill`
The value to use in the statistic if a slice of the data is empty.

Type Any (default = 0)

Examples

Make a little data set.

```
>>> import cytoflow as flow
>>> import_op = flow.ImportOp()
>>> import_op.tubes = [flow.Tube(file = "Plate01/RFP_Well_A3.fcs",
...                               conditions = {'Dox' : 10.0}),
...                    flow.Tube(file = "Plate01/CFP_Well_A4.fcs",
...                               conditions = {'Dox' : 1.0})]
>>> import_op.conditions = {'Dox' : 'float'}
>>> ex = import_op.apply()
```

Create and parameterize the operation.

```
>>> ch_op = flow.ChannelStatisticOp(name = 'MeanByDox',
...                     channel = 'Y2-A',
...                     function = flow.geom_mean,
...                     by = ['Dox'])
>>> ex2 = ch_op.apply(ex)
```

View the new operation

```
>>> print(ex2.statistics.keys())
dict_keys([('MeanByDox', 'geom_mean')])
```

```
>>> print(ex2.statistics[('MeanByDox', 'geom_mean')])
Dox
1.0      19.805601
10.0    446.981927
dtype: float64
```

apply(*experiment*)

Apply the operation to an *Experiment*.

Parameters **experiment** – The *Experiment* to apply this operation to.

Returns A new *Experiment*, containing a new entry in *Experiment.statistics*. The key of the new entry is a tuple (name, function) (or (name, statistic_name) if *statistic_name* is set.

Return type *Experiment*

cytoflow.operations.color_translation

The *color_translation* module has two classes:

ColorTranslationOp – translates the measurements in one channel to the same scale as another channel, using a two- or three-color control to estimate the transfer function.

ColorTranslationDiagnostic – a diagnostic view showing how the *ColorTranslationOp* operation estimated its parameters.

class cytoflow.operations.color_translation.**ColorTranslationOp**

Bases: traits.has_traits.HasStrictTraits

Translate measurements from one color's scale to another, using a two-color or three-color control.

To use, set up the *controls* dictionary with the channels to convert and the FCS files to compute the mapping. Call *estimate* to parameterize the module; check that the plots look good by calling the *ColorTranslationDiagnostic.plot* method of the *ColorTranslationDiagnostic* instance returned by *default_view*; then call *apply* to apply the translation to an *Experiment*.

controls

Two-color controls used to determine the mapping. They keys are tuples of **from-channel** and **to-channel**. The values are FCS files containing two-color constitutive fluorescent expression data for the mapping.

Type Dict((Str, Str), File)

mixture_model

If True, try to model the **from** channel as a mixture of expressing cells and non-expressing cells (as you would get with a transient transfection), then weight the regression by the probability that the the cell is from the top (transfected) distribution. Make sure you check the diagnostic plots to see that this worked!

Type Bool (default = False)

`linear_model`
Set this to `True` to get a scaling that is strictly multiplicative, mirroring the TASBE approach. Do check the diagnostic plot, though, to see how well (or poorly) your model fits the data.

Type Bool (default = False)

`control_conditions`
Occasionally, you'll need to specify the experimental conditions that the bleedthrough tubes were collected under (to apply the operations in the history.) Specify them here. The key is a tuple of channel names; the value is a dictionary of the conditions (same as you would specify for a `cytoflow.operations.import_op.Tube`)

Type Dict((Str, Str), Dict(Str, Any))

Notes

In the TASBE workflow, this operation happens *after* the application of `AutofluorescenceOp` and `BleedthroughLinearOp`. The entire operation history of the `Experiment` that is passed to `estimate` is replayed on the control files in `controls`, so they are also corrected for autofluorescence and bleedthrough, and have metadata for subsetting.

Examples

Create a small experiment:

```
>>> import cytoflow as flow
>>> import_op = flow.ImportOp()
>>> import_op.tubes = [flow.Tube(file = "tasbe/mkate.fcs")]
>>> ex = import_op.apply()
```

Create and parameterize the operation

```
>>> color_op = flow.ColorTranslationOp()
>>> color_op.controls = {("Pacific Blue-A", "FITC-A") : "tasbe/rby.fcs",
...                      ("PE-Tx-Red-YG-A", "FITC-A") : "tasbe/rby.fcs"}
>>> color_op.mixture_model = True
```

Estimate the model parameters

```
>>> color_op.estimate(ex)
```

Plot the diagnostic plot

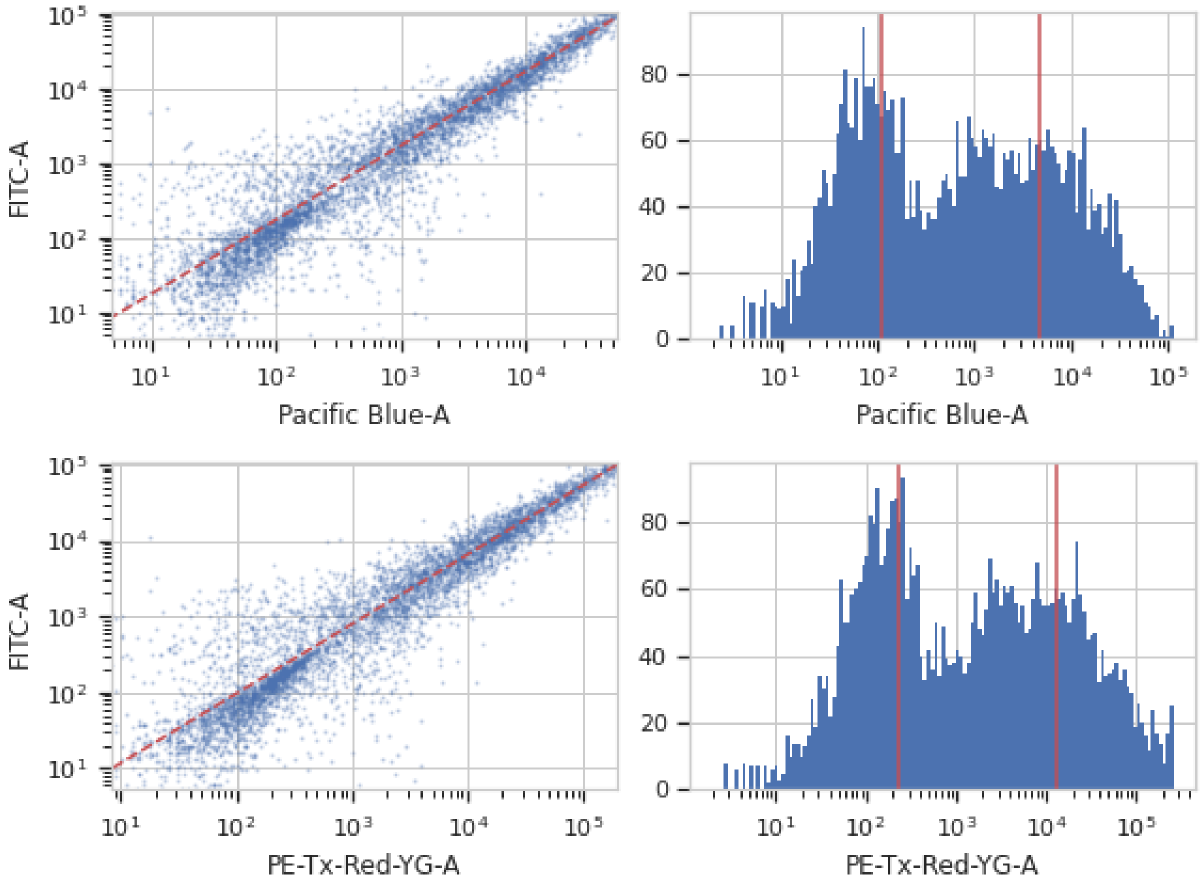
```
>>> color_op.default_view().plot(ex)
```

Apply the operation to the experiment

```
>>> ex = color_op.apply(ex)
```

`estimate`(*experiment*, *subset=None*)
Estimate the mapping from the two-channel controls

Parameters


FITC-A
Pacific Blue-A
Pacific Blue-A
FITC-A
PE-Tx-Red-YG-A
PE-Tx-Red-YG-A

- **experiment** (`Experiment`) – The `Experiment` used to check the voltages, etc. of the control tubes. Also the source of the operation history that is replayed on the control tubes.
- **subset** (*Str*) – A Python expression used to subset the controls before estimating the color translation parameters.

`apply`(*experiment*)
Applies the color translation to an experiment

Parameters **experiment** (`Experiment`) – the old_experiment to which this op is applied

Returns

a new experiment with the color translation applied. The corrected channels also have the following new metadata:

channel_translation : Str Which channel was this one translated to?

channel_translation_fn : Callable (`pandas.Series` –> `pandas.Series`) The function that translated this channel

Return type *Experiment*

`default_view`(***kwargs*)
Returns a diagnostic plot to see if the bleedthrough spline estimation is working.

Returns A diagnostic view, call `ColorTranslationDiagnostic.plot` to see the diagnostic plots

Return type *IView*

`class cytoflow.operations.color_translation.ColorTranslationDiagnostic`
Bases: `traits.has_traits.HasStrictTraits`

`name`
The instance name (for serialization, UI etc.)

Type Str

`op`
The op whose parameters we're viewing

Type Instance(`ColorTranslationOp`)

`subset`
A Python expression specifying a subset of the events in the control FCS files to plot

Type str

`plot`(*experiment*, ***kwargs*)
Plot the plots

Parameters **experiment** (*Experiment*)

cytoflow.operations.density

The `density` module has two classes:

`DensityGateOp` – gates an `Experiment` based on a two-dimensional (smoothed) density plot.

`DensityGateView` – diagnostic view that plots the gate estimated by the `DensityGateOp`.

class `cytoflow.operations.density.`**`DensityGateOp`**

Bases: `traits.has_traits.HasStrictTraits`

This module computes a gate based on a 2D density plot. The user chooses what proportion of events to keep, and the module creates a gate that selects that proportion of events in the highest-density bins of the 2D density histogram.

`name`
The operation name; determines the name of the new metadata column

Type Str

`xchannel`
The X channel to apply the binning to.

Type Str

`ychannel`
The Y channel to apply the binning to.

Type Str

`xscale`
Re-scale the data on the X acis before fitting the data?

Type {“linear”, “logicle”, “log”} (default = “linear”)

`yscale`
Re-scale the data on the Y axis before fitting the data?

Type {“linear”, “logicle”, “log”} (default = “linear”)

`keep`
What proportion of events to keep? Must be >`0` and <`1`

Type Float (default = 0.9)

`bins`
How many bins should there be on each axis? Must be positive.

Type Int (default = 100)

`min_quantile`
Clip values below this quantile

Type Float (default = 0.001)

`max_quantile`
Clip values above this quantile

Type Float (default = 1.0)

`sigma`
What standard deviation to use for the gaussian blur?

Type Float (default = 1.0)

by

A list of metadata attributes to aggregate the data before estimating the gate. For example, if the experiment has two pieces of metadata, `Time` and `Dox`, setting `by = ["Time", "Dox"]` will fit a separate gate to each subset of the data with a unique combination of `Time` and `Dox`.

Type List(Str)

Notes

This gating method was developed by John Sexton, in Jeff Tabor's lab at Rice University.

From http://taborlab.github.io/FlowCal/fundamentals/density_gate.html, the method is as follows:

1. Determines the number of events to keep, based on the user specified gating fraction and the total number of events of the input sample.
2. Divides the 2D channel space into a rectangular grid, and counts the number of events falling within each bin of the grid. The number of counts per bin across all bins comprises a 2D histogram, which is a coarse approximation of the underlying probability density function.
3. Smoothes the histogram generated in Step 2 by applying a Gaussian Blur. Theoretically, the proper amount of smoothing results in a better estimate of the probability density function. Practically, smoothing eliminates isolated bins with high counts, most likely corresponding to noise, and smoothes the contour of the gated region.
4. Selects the bins with the greatest number of events in the smoothed histogram, starting with the highest and proceeding downward until the desired number of events to keep, calculated in step 1, is achieved.

Examples

Make a little data set.

```
>>> import cytoflow as flow
>>> import_op = flow.ImportOp()
>>> import_op.tubes = [flow.Tube(file = "Plate01/RFP_Well_A3.fcs",
...                              conditions = {'Dox' : 10.0}),
...                    flow.Tube(file = "Plate01/CFP_Well_A4.fcs",
...                              conditions = {'Dox' : 1.0})]
>>> import_op.conditions = {'Dox' : 'float'}
>>> ex = import_op.apply()
```

Create and parameterize the operation.

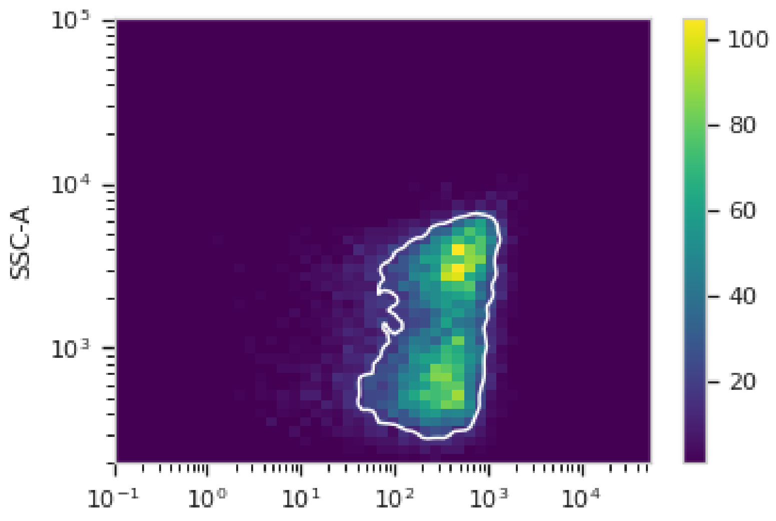
```
>>> dens_op = flow.DensityGateOp(name = 'Density',
...                              xchannel = 'FSC-A',
...                              xscale = 'log',
...                              ychannel = 'SSC-A',
...                              yscale = 'log',
...                              keep = 0.5)
```

Find the bins to keep

```
>>> dens_op.estimate(ex)
```

Plot a diagnostic view

```
>>> dens_op.default_view().plot(ex)
```



Apply the gate

```
>>> ex2 = dens_op.apply(ex)
```

estimate(*experiment*, *subset=None*)

Split the data set into bins and determine which ones to keep.

Parameters

- **experiment** (`Experiment`) – The `Experiment` to use to estimate the gate parameters.
- **subset** (*Str (default = None)*) – If set, determine the gate parameters on only a subset of the `experiment` parameter.

apply(*experiment*)

Creates a new condition based on membership in the gate that was parameterized with `estimate`.

Parameters **experiment** (`Experiment`) – the `Experiment` to apply the gate to.

Returns a new `Experiment` with the new gate applied.

Return type `Experiment`

default_view(***kwargs*)

Returns a diagnostic plot of the Gaussian mixture model.

Returns a diagnostic view, call `DensityGateView.plot` to see the diagnostic plot.

Return type `IView`

class cytoflow.operations.density.**DensityGateView**

Bases: `cytoflow.operations.base_op_views.By2DView`, `cytoflow.operations.base_op_views.AnnotatingView`, `cytoflow.views.densityplot.DensityView`

A diagnostic view for `DensityGateOp`. Draws a density plot, then outlines the selected bins in white.

facets

A read-only list of the conditions used to facet this view.

Type List(Str)

`by`
A read-only list of the conditions used to group this view's data before plotting.

Type List(Str)

`xchannel`
The channels to use for this view's X axis. If you created the view using *`default_view`*, this is already set.

Type Str

`ychannel`
The channels to use for this view's Y axis. If you created the view using *`default_view`*, this is already set.

Type Str

`xscale`
The way to scale the x axis. If you created the view using *`default_view`*, this may be already set.

Type {'linear', 'log', 'logicle'}

`yscale`
The way to scale the y axis. If you created the view using *`default_view`*, this may be already set.

Type {'linear', 'log', 'logicle'}

`op`
The *`IOperation`* that this view is associated with. If you created the view using *`default_view`*, this is already set.

Type Instance(*`IOperation`*)

`huefacet`
You must leave the hue facet unset!

Type None

`subset`
An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

`xfacet`
Set to one of the *`Experiment.conditions`* in the *`Experiment`*, and a new column of subplots will be added for every unique value of that condition.

Type String

`yfacet`
Set to one of the *`Experiment.conditions`* in the *`Experiment`*, and a new row of subplots will be added for every unique value of that condition.

Type String

`huescale`
How should the color scale for *`huefacet`* be scaled?

Type {'linear', 'log', 'logicle'}

`plot`(*experiment*, ***kwargs*)
Plot the plots.

Parameters

- **experiment** (*Experiment*) – The `Experiment` to plot using this view.
- **title** (*str*) – Set the plot title
- **xlabel** (*str*) – Set the X axis label
- **ylabel** (*str*) – Set the Y axis label
- **huelabel** (*str*) – Set the label for the hue facet (in the legend)
- **legend** (*bool*) – Plot a legend for the color or hue facet? Defaults to `True`.
- **sharex** (*bool*) – If there are multiple subplots, should they share X axes? Defaults to `True`.
- **sharey** (*bool*) – If there are multiple subplots, should they share Y axes? Defaults to `True`.
- **row_order** (*list*) – Override the row facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **col_order** (*list*) – Override the column facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **hue_order** (*list*) – Override the hue facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **height** (*float*) – The height of *each row* in inches. Default = 3.0
- **aspect** (*float*) – The aspect ratio *of each subplot*. Default = 1.5
- **col_wrap** (*int*) – If `xfacet` is set and `yfacet` is not set, you can "wrap" the subplots around so that they form a multi-row grid by setting this to the number of columns you want.
- **sns_style** (*{"darkgrid", "whitegrid", "dark", "white", "ticks"}*) – Which `seaborn` style to apply to the plot? Default is `whitegrid`.
- **sns_context** (*{"paper", "notebook", "talk", "poster"}*) – Which `seaborn` context to use? Controls the scaling of plot elements such as tick labels and the legend. Default is `talk`.
- **palette** (*palette name, list, or dict*) – Colors to use for the different levels of the hue variable. Should be something that can be interpreted by `seaborn.color_palette`, or a dictionary mapping hue levels to matplotlib colors.
- **despine** (*Bool*) – Remove the top and right axes from the plot? Default is `True`.
- **min_quantile** (*float (>0.0 and <1.0, default = 0.001)*) – Clip data that is less than this quantile.
- **max_quantile** (*float (>0.0 and <1.0, default = 1.00)*) – Clip data that is greater than this quantile.
- **xlim** (*(float, float)*) – Set the range of the plot's X axis.
- **ylim** (*(float, float)*) – Set the range of the plot's Y axis.
- **gridsize** (*int*) – The size of the grid on each axis. Default = 50
- **smoothed** (*bool*) – Should the resulting mesh be smoothed?
- **smoothed_sigma** (*int*) – The standard deviation of the smoothing kernel. default = 1.
- **cmap** (*cmap*) – An instance of matplotlib.colors.Colormap. By default, the `viridis` colormap is used
- **under_color** (*matplotlib color*) – Sets the color to be used for low out-of-range values.

- **bad_color** (*matplotlib color*) – Set the color to be used for masked values.
- **color** (*matplotlib color*) – The color to plot the annotations. Overrides the default color cycle.
- **plot_name** (*Str*) – If this `IView` can make multiple plots, `plot_name` is the name of the plot to make. Must be one of the values retrieved from `enum_plots`.
- **contour_props** (*Dict*) – The keyword arguments passed to the `matplotlib.axes.Axes.contour` constructor, which controls the visual properties of the contour that's plotted on top of the density plot. Default: `{'colors' : 'w'}`

cytoflow.operations.flowpeaks

The `flowpeaks` module has the classes that support the **flowPeaks** clustering algorithm. It has four classes:

`FlowPeaksOp` – an operation that implements the **flowPeaks** algorithm (see the class documentation for a reference.)

`FlowPeaks1DView` – a diagnostic view that shows how the `FlowPeaksOp` performed its clustering (on a 1D data set, using a histogram).

`FlowPeaks2DView` – a diagnostic view that shows how the `FlowPeaksOp` performed its clustering (on a 2D data set, using a scatter plot).

`FlowPeaks2DDensityView` – a diagnostic view that shows how the `FlowPeaksOp` performed its clustering (on a 2D data set, using a density plot).

`class cytoflow.operations.flowpeaks.FlowPeaksOp`

Bases: `traits.has_traits.HasStrictTraits`

This module uses the **flowPeaks** algorithm to assign events to clusters in an unsupervised manner.

Call `estimate` to compute the clusters.

Calling `apply` creates a new categorical metadata variable named `name`, with possible values `{name}_1` .... `name_n` where `n` is the number of clusters estimated.

The same model may not be appropriate for different subsets of the data set. If this is the case, you can use the `by` attribute to specify metadata by which to aggregate the data before estimating (and applying) a model. The number of clusters is a model parameter and it may vary in each subset.

`name`

The operation name; determines the name of the new metadata column

Type Str

`channels`

The channels to apply the clustering algorithm to.

Type List(Str)

`scale`

Re-scale the data in the specified channels before fitting. If a channel is in `channels` but not in `scale`, the current package-wide default (set with `set_default_scale`) is used.

Type Dict(Str : Enum("linear", "logicle", "log"))

`by`

A list of metadata attributes to aggregate the data before estimating the model. For example, if the experiment has two pieces of metadata, `Time` and `Dox`, setting `by = ["Time", "Dox"]` will fit the model separately to each subset of the data with a unique combination of `Time` and `Dox`.

Type List(Str)

`h`
A scalar value by which to scale the covariance matrices of the underlying density function. (See `Notes`, below, for more details.)

Type Float (default = 1.5)

`h0`
A scalar value by which to smooth the covariance matrices of the underlying density function. (See `Notes`, below, for more details.)

Type Float (default = 1.0)

`tol`
How readily should clusters be merged? Must be between 0 and 1. See `Notes`, below, for more details.

Type Float (default = 0.5)

`merge_dist`
How far apart can clusters be before they are merged? This is a unit-free scalar, and is approximately the maximum number of k-means clusters between peaks.

Type Float (default = 5)

`find_outliers`
Should the algorithm use an extra step to identify outliers?

Note: I have disabled this code until I can try to make it faster.

Type Bool (default = False)

Notes

This algorithm uses kmeans to find a large number of clusters, then hierarchically merges those clusters. Thus, the user does not need to specify the number of clusters in advance, and it can find non-convex clusters. It also operates in an arbitrary number of dimensions.

The merging happens in two steps. First, the cluster centroids are used to estimate an underlying density function. Then, the local maxima of the density function are found using a numerical optimization starting from each centroid, and k-means clusters that converge to the same local maximum are merged. Finally, these clusters-of-clusters are merged if their local maxima are (a) close enough, and (b) the density function between them is smooth enough. Thus, the final assignment of each event depends on the k-means cluster it ends up in, and which cluster-of-clusters that k-means centroid is assigned to.

There are a lot of parameters that affect this process. The k-means clustering is pretty robust (though somewhat sensitive to the number of clusters, which is currently not exposed in the API.) The most important are exposed as attributes of the *`FlowPeaksOp`* class. These include:

- ***`h`*, *`h0`*: sometimes the density function is too "rough" to** find good local maxima. These parameters smooth it out by widening the covariance matrices. Increasing *`h`* makes the density rougher; increasing *`h0`* makes it smoother.

- ***`tol`*: How smooth does the density function have to be between two** density maxima to merge them? Must be between 0 and 1.

- ***`merge_dist`*: How close must two maxima be to merge them? This** value is a unit-free scalar, and is approximately the number of k-means clusters between the two maxima.

For details and a theoretical justification, see[1]

References

Examples

Make a little data set.

```
>>> import cytoflow as flow
>>> import_op = flow.ImportOp()
>>> import_op.tubes = [flow.Tube(file = "Plate01/RFP_Well_A3.fcs",
...                              conditions = {'Dox' : 10.0}),
...                    flow.Tube(file = "Plate01/CFP_Well_A4.fcs",
...                              conditions = {'Dox' : 1.0})]
>>> import_op.conditions = {'Dox' : 'float'}
>>> ex = import_op.apply()
```

Create and parameterize the operation.

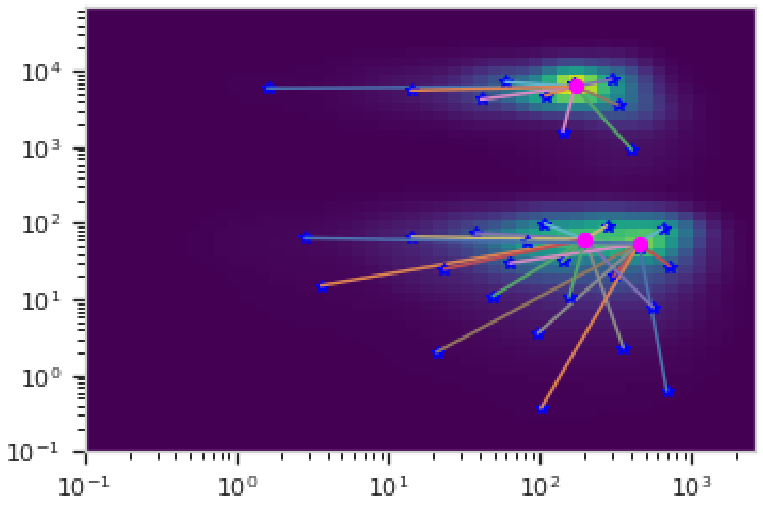
```
>>> fp_op = flow.FlowPeaksOp(name = 'Flow',
...                          channels = ['V2-A', 'Y2-A'],
...                          scale = {'V2-A' : 'log',
...                                   'Y2-A' : 'log'},
...                          h0 = 3)
```

Estimate the clusters

```
>>> fp_op.estimate(ex)
```

Plot a diagnostic view of the underlying density

```
>>> fp_op.default_view(density = True).plot(ex)
```



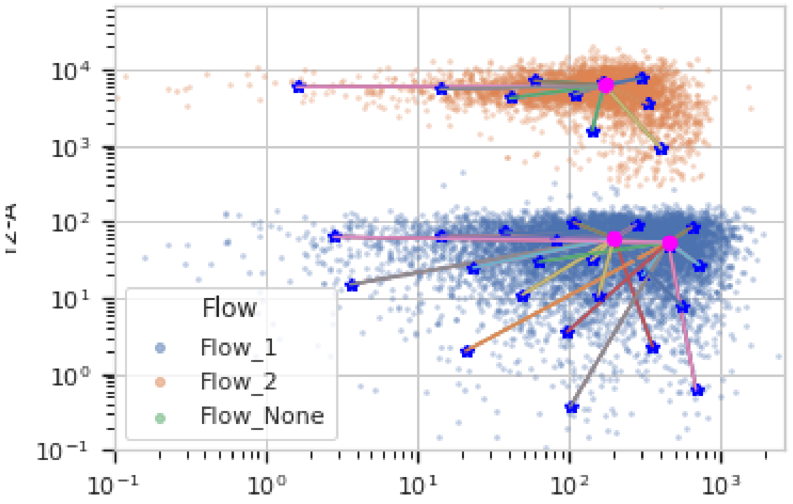
[1] Ge, Yongchao and Sealfon, Stuart C. "flowPeaks: a fast unsupervised clustering for flow cytometry data via K-means and density peak finding" Bioinformatics (2012) 28 (15): 2052-2058.

Apply the gate

```
>>> ex2 = fp_op.apply(ex)
```

Plot a diagnostic view with the event assignments

```
>>> fp_op.default_view().plot(ex2)
```



estimate(*experiment*, *subset=None*)

Estimate the k-means clusters, then hierarchically merge them.

Parameters

- **experiment** (*Experiment*) – The `Experiment` to use to estimate the k-means clusters
- **subset** (*str (default = None)*) – A Python expression that specifies a subset of the data in `experiment` to use to parameterize the operation.

apply(*experiment*)

Assign events to a cluster.

Assigns each event to one of the k-means centroids from `estimate`, then groups together events in the same cluster hierarchy.

Parameters **experiment** (*Experiment*) – the `Experiment` to apply the gate to.

Returns A new `Experiment` with the gate applied to it. TODO - document the extra statistics

Return type `Experiment`

default_view(***kwargs*)

Returns a diagnostic plot of the Gaussian mixture model.

Parameters

- **channels** (*List(Str)*) – Which channels to plot? Must be contain either one or two channels.
- **scale** (*List({'linear', 'log', 'logicle'})*) – How to scale the channels before plotting them
- **density** (*bool*) – Should we plot a scatterplot or the estimated density function?

Returns an `IView`, call `plot` to see the diagnostic plot.

Return type *IView*

class cytoflow.operations.flowpeaks.**FlowPeaks1DView**

Bases: *cytoflow.operations.base_op_views.By1DView*, *cytoflow.operations.base_op_views.AnnotatingView*, *cytoflow.views.histogram.HistogramView*

A one-dimensional diagnostic view for *FlowPeaksOp*. Plots a histogram of the channel, then overlays the k-means centroids in blue.

facets
A read-only list of the conditions used to facet this view.

Type List(Str)

by
A read-only list of the conditions used to group this view's data before plotting.

Type List(Str)

channel
The channel this view is viewing. If you created the view using *default_view*, this is already set.

Type Str

scale
The way to scale the x axes. If you created the view using *default_view*, this may be already set.

Type {'linear', 'log', 'logicle'}

op
The *IOperation* that this view is associated with. If you created the view using *default_view*, this is already set.

Type Instance(*IOperation*)

subset
An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

xfacet
Set to one of the *Experiment.conditions* in the *Experiment*, and a new column of subplots will be added for every unique value of that condition.

Type String

yfacet
Set to one of the *Experiment.conditions* in the *Experiment*, and a new row of subplots will be added for every unique value of that condition.

Type String

huefacet
Set to one of the *Experiment.conditions* in the in the *Experiment*, and a new color will be added to the plot for every unique value of that condition.

Type String

huescale
How should the color scale for *huefacet* be scaled?

Type {'linear', 'log', 'logicle'}

`plot`(*experiment*, ***kwargs*)
Plot the plots.

Parameters

- **experiment** (*Experiment*) – The `Experiment` to plot using this view.
- **title** (*str*) – Set the plot title
- **xlabel** (*str*) – Set the X axis label
- **ylabel** (*str*) – Set the Y axis label
- **huelabel** (*str*) – Set the label for the hue facet (in the legend)
- **legend** (*bool*) – Plot a legend for the color or hue facet? Defaults to `True`.
- **sharex** (*bool*) – If there are multiple subplots, should they share X axes? Defaults to `True`.
- **sharey** (*bool*) – If there are multiple subplots, should they share Y axes? Defaults to `True`.
- **row_order** (*list*) – Override the row facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **col_order** (*list*) – Override the column facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **hue_order** (*list*) – Override the hue facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **height** (*float*) – The height of *each row* in inches. Default = 3.0
- **aspect** (*float*) – The aspect ratio *of each subplot*. Default = 1.5
- **col_wrap** (*int*) – If `xfacet` is set and `yfacet` is not set, you can "wrap" the subplots around so that they form a multi-row grid by setting this to the number of columns you want.
- **sns_style** (*{"darkgrid", "whitegrid", "dark", "white", "ticks"}*) – Which `seaborn` style to apply to the plot? Default is `whitegrid`.
- **sns_context** (*{"paper", "notebook", "talk", "poster"}*) – Which `seaborn` context to use? Controls the scaling of plot elements such as tick labels and the legend. Default is `talk`.
- **palette** (*palette name, list, or dict*) – Colors to use for the different levels of the hue variable. Should be something that can be interpreted by `seaborn.color_palette`, or a dictionary mapping hue levels to matplotlib colors.
- **despine** (*Bool*) – Remove the top and right axes from the plot? Default is `True`.
- **min_quantile** (*float (>0.0 and <1.0, default = 0.001)*) – Clip data that is less than this quantile.
- **max_quantile** (*float (>0.0 and <1.0, default = 1.00)*) – Clip data that is greater than this quantile.
- **lim** (*(float, float)*) – Set the range of the plot's data axis.
- **orientation** (*{'vertical', 'horizontal'}*)
- **num_bins** (*int*) – The number of bins to plot in the histogram. Clipped to [100, 1000]
- **histtype** (*{'stepfilled', 'step', 'bar'}*) – The type of histogram to draw. `stepfilled` is the default, which is a line plot with a color filled under the curve.
- **density** (*bool*) – If `True`, re-scale the histogram to form a probability density function, so the area under the histogram is 1.

- **linewidth** (*float*) – The width of the histogram line (in points)
- **linestyle** (*['-' | '–' | '-.' | ':' | "None"]*) – The style of the line to plot
- **alpha** (*float (default = 0.5)*) – The alpha blending value, between 0 (transparent) and 1 (opaque).
- **color** (*matplotlib color*) – The color to plot the annotations. Overrides the default color cycle.
- **plot_name** (*Str*) – If this *IView* can make multiple plots, `plot_name` is the name of the plot to make. Must be one of the values retrieved from *enum_plots*.

`class cytoflow.operations.flowpeaks.`**`FlowPeaks2DView`**

Bases: *cytoflow.operations.base_op_views.By2DView*, *cytoflow.operations.base_op_views.AnnotatingView*, *cytoflow.views.scatterplot.ScatterplotView*

A two-dimensional diagnostic view for *FlowPeaksOp*. Plots a scatter-plot of the two channels, then overlays the k-means centroids in blue and the clusters-of-k-means in pink.

`facets`

A read-only list of the conditions used to facet this view.

Type List(Str)

`by`

A read-only list of the conditions used to group this view's data before plotting.

Type List(Str)

`xchannel`

The channels to use for this view's X axis. If you created the view using *default_view*, this is already set.

Type Str

`ychannel`

The channels to use for this view's Y axis. If you created the view using *default_view*, this is already set.

Type Str

`xscale`

The way to scale the x axis. If you created the view using *default_view*, this may be already set.

Type {'linear', 'log', 'logicle'}

`yscale`

The way to scale the y axis. If you created the view using *default_view*, this may be already set.

Type {'linear', 'log', 'logicle'}

`op`

The *IOperation* that this view is associated with. If you created the view using *default_view*, this is already set.

Type Instance(*IOperation*)

`subset`

An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

`xfacet`
Set to one of the `Experiment.conditions` in the `Experiment`, and a new column of subplots will be added for every unique value of that condition.

Type String

`yfacet`
Set to one of the `Experiment.conditions` in the `Experiment`, and a new row of subplots will be added for every unique value of that condition.

Type String

`huefacet`
Set to one of the `Experiment.conditions` in the in the `Experiment`, and a new color will be added to the plot for every unique value of that condition.

Type String

`huescale`
How should the color scale for `huefacet` be scaled?

Type {'linear', 'log', 'logicle'}

`plot`(*experiment*, ***kwargs*)
Plot the plots.

Parameters

- **experiment** (*Experiment*) – The `Experiment` to plot using this view.
- **title** (*str*) – Set the plot title
- **xlabel** (*str*) – Set the X axis label
- **ylabel** (*str*) – Set the Y axis label
- **huelabel** (*str*) – Set the label for the hue facet (in the legend)
- **legend** (*bool*) – Plot a legend for the color or hue facet? Defaults to `True`.
- **sharex** (*bool*) – If there are multiple subplots, should they share X axes? Defaults to `True`.
- **sharey** (*bool*) – If there are multiple subplots, should they share Y axes? Defaults to `True`.
- **row_order** (*list*) – Override the row facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **col_order** (*list*) – Override the column facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **hue_order** (*list*) – Override the hue facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **height** (*float*) – The height of *each row* in inches. Default = 3.0
- **aspect** (*float*) – The aspect ratio *of each subplot*. Default = 1.5
- **col_wrap** (*int*) – If `xfacet` is set and `yfacet` is not set, you can "wrap" the subplots around so that they form a multi-row grid by setting this to the number of columns you want.
- **sns_style** (*{"darkgrid", "whitegrid", "dark", "white", "ticks"}*) – Which `seaborn` style to apply to the plot? Default is `whitegrid`.
- **sns_context** (*{"paper", "notebook", "talk", "poster"}*) – Which `seaborn` context to use? Controls the scaling of plot elements such as tick labels and the legend. Default is `talk`.

- **palette** (*palette name, list, or dict*) – Colors to use for the different levels of the hue variable. Should be something that can be interpreted by `seaborn.color_palette`, or a dictionary mapping hue levels to matplotlib colors.
- **despine** (*Bool*) – Remove the top and right axes from the plot? Default is `True`.
- **min_quantile** (*float (>0.0 and <1.0, default = 0.001)*) – Clip data that is less than this quantile.
- **max_quantile** (*float (>0.0 and <1.0, default = 1.00)*) – Clip data that is greater than this quantile.
- **xlim** (*(float, float)*) – Set the range of the plot's X axis.
- **ylim** (*(float, float)*) – Set the range of the plot's Y axis.
- **alpha** (*float (default = 0.25)*) – The alpha blending value, between 0 (transparent) and 1 (opaque).
- **s** (*int (default = 2)*) – The size in points^2.
- **marker** (*a matplotlib marker style, usually a string*) – Specfies the glyph to draw for each point on the scatterplot. See matplotlib.markers for examples. Default: 'o'
- **color** (*matplotlib color*) – The color to plot the annotations. Overrides the default color cycle.
- **plot_name** (*Str*) – If this *`IView`* can make multiple plots, `plot_name` is the name of the plot to make. Must be one of the values retrieved from *`enum_plots`*.

`class cytoflow.operations.flowpeaks.`**`FlowPeaks2DDensityView`**

Bases: *`cytoflow.operations.base_op_views.By2DView`*, *`cytoflow.operations.base_op_views.AnnotatingView`*, *`cytoflow.operations.base_op_views.NullView`*

A two-dimensional diagnostic view for *`FlowPeaksOp`*. Plots the estimated density function of the two channels, then overlays the k-means centroids in blue and the clusters-of-k-means in pink.

`facets`

A read-only list of the conditions used to facet this view.

Type List(Str)

`by`

A read-only list of the conditions used to group this view's data before plotting.

Type List(Str)

`xchannel`

The channels to use for this view's X axis. If you created the view using *`default_view`*, this is already set.

Type Str

`ychannel`

The channels to use for this view's Y axis. If you created the view using *`default_view`*, this is already set.

Type Str

`xscale`

The way to scale the x axis. If you created the view using *`default_view`*, this may be already set.

Type {'linear', 'log', 'logicle'}

`yscale`
The way to scale the y axis. If you created the view using `default_view`, this may be already set.

Type {'linear', 'log', 'logicle'}

`op`
The `IOperation` that this view is associated with. If you created the view using `default_view`, this is already set.

Type Instance(`IOperation`)

`subset`
An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

`xfacet`
Set to one of the `Experiment.conditions` in the `Experiment`, and a new column of subplots will be added for every unique value of that condition.

Type String

`yfacet`
Set to one of the `Experiment.conditions` in the `Experiment`, and a new row of subplots will be added for every unique value of that condition.

Type String

`huefacet`
Set to one of the `Experiment.conditions` in the in the `Experiment`, and a new color will be added to the plot for every unique value of that condition.

Type String

`huescale`
How should the color scale for `huefacet` be scaled?

Type {'linear', 'log', 'logicle'}

`plot`(*experiment*, ***kwargs*)
Plot the plots.

Parameters

- **experiment** (*Experiment*) – The `Experiment` to plot using this view.
- **title** (*str*) – Set the plot title
- **xlabel** (*str*) – Set the X axis label
- **ylabel** (*str*) – Set the Y axis label
- **huelabel** (*str*) – Set the label for the hue facet (in the legend)
- **legend** (*bool*) – Plot a legend for the color or hue facet? Defaults to `True`.
- **sharex** (*bool*) – If there are multiple subplots, should they share X axes? Defaults to `True`.
- **sharey** (*bool*) – If there are multiple subplots, should they share Y axes? Defaults to `True`.
- **row_order** (*list*) – Override the row facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **col_order** (*list*) – Override the column facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.

- **hue_order** (*list*) – Override the hue facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **height** (*float*) – The height of *each row* in inches. Default = 3.0
- **aspect** (*float*) – The aspect ratio *of each subplot*. Default = 1.5
- **col_wrap** (*int*) – If `xfacet` is set and `yfacet` is not set, you can "wrap" the subplots around so that they form a multi-row grid by setting this to the number of columns you want.
- **sns_style** (*{"darkgrid", "whitegrid", "dark", "white", "ticks"}*) – Which `seaborn` style to apply to the plot? Default is `whitegrid`.
- **sns_context** (*{"paper", "notebook", "talk", "poster"}*) – Which `seaborn` context to use? Controls the scaling of plot elements such as tick labels and the legend. Default is `talk`.
- **palette** (*palette name, list, or dict*) – Colors to use for the different levels of the hue variable. Should be something that can be interpreted by `seaborn.color_palette`, or a dictionary mapping hue levels to matplotlib colors.
- **despine** (*Bool*) – Remove the top and right axes from the plot? Default is `True`.
- **min_quantile** (*float (>0.0 and <1.0, default = 0.001)*) – Clip data that is less than this quantile.
- **max_quantile** (*float (>0.0 and <1.0, default = 1.00)*) – Clip data that is greater than this quantile.
- **color** (*matplotlib color*) – The color to plot the annotations. Overrides the default color cycle.
- **xlim** (*(float, float)*) – Set the range of the plot's X axis.
- **ylim** (*(float, float)*) – Set the range of the plot's Y axis.
- **plot_name** (*Str*) – If this `IView` can make multiple plots, `plot_name` is the name of the plot to make. Must be one of the values retrieved from `enum_plots`.

cytoflow.operations.frame_stat

The `frame_stat` module contains one class:

`FrameStatisticOp` – applies a function to subsets of a data set, and adds the resulting statistic to the `Experiment`. Unlike `ChannelStatisticOp`, which operates on a single channel, this operation operates on entire `pandas.DataFrame`.

class `cytoflow.operations.frame_stat.`**`FrameStatisticOp`**

Bases: `traits.has_traits.HasStrictTraits`

Apply a function to subsets of a data set, and add it as a statistic to the experiment.

The `apply` function groups the data by the variables in `by`, then applies the `function` callable to each `pandas.DataFrame` subset. The callable should take a `pandas.DataFrame` as its only parameter. The return type is arbitrary, but to be used with the rest of `cytoflow` it should probably be a numeric type or an iterable of numeric types.

`name`

The operation name. Becomes the first element in the `Experiment.statistics` key tuple.

Type Str

function
The function used to compute the statistic. Must take a `pandas.DataFrame` as its only argument. The return type is arbitrary, but to be used with the rest of *cytoflow* it should probably be a numeric type or an iterable of numeric types. If *statistic_name* is unset, the name of the function becomes the second in element in the *Experiment.statistics* key tuple.

Type Callable

statistic_name
The name of the function; if present, becomes the second element in the *Experiment.statistics* key tuple. Particularly useful if *function* is a lambda.

Type Str

by
A list of metadata attributes to aggregate the data before applying the function. For example, if the experiment has two pieces of metadata, `Time` and `Dox`, setting `by = ["Time", "Dox"]` will apply *function* separately to each subset of the data with a unique combination of `Time` and `Dox`.

Type List(Str)

subset
A Python expression sent to Experiment.query() to subset the data before computing the statistic.

Type Str

fill
The value to use in the statistic if a slice of the data is empty.

Type Any (default = 0)

Examples

```
>>> stats_op = FrameStatisticOp(name = "ByDox",
...                              function = lambda x: np.mean(x["FITC-A"],
...                              statistic_name = "Mean",
...                              by = ["Dox"])
>>> ex2 = stats_op.apply(ex)
```

apply(*experiment*)

cytoflow.operations.gaussian

gaussian contains three classes:

GaussianMixtureOp – an operation that fits a Gaussian mixture model to one or more channels.

GaussianMixture1DView – a diagnostic view that shows how the *GaussianMixtureOp* estimated its model (on a 1D data set, using a histogram).

GaussianMixture2DView – a diagnostic view that shows how the *GaussianMixtureOp* estimated its model (on a 2D data set, using a scatter plot).

class cytoflow.operations.gaussian.**GaussianMixtureOp**
Bases: `traits.has_traits.HasStrictTraits`

This module fits a Gaussian mixture model with a specified number of components to one or more channels.

If *num_components* > 1, *apply* creates a new categorical metadata variable named `name`, with possible values `{name}_1` .... `name_n` where `n` is the number of components. An event is assigned to `name_i` category if it has the highest posterior probability of having been produced by component `i`. If an event has a value that is outside the range of one of the channels' scales, then it is assigned to `{name}_None`.

Optionally, if *sigma* is greater than 0, *apply* creates new `boolean` metadata variables named `{name}_1` ... `{name}_n` where `n` is the number of components. The column `{name}_i` is `True` if the event is less than *sigma* standard deviations from the mean of component `i`. If *num_components* is `1`, *sigma* must be greater than 0.

Note: The *sigma* attribute does NOT affect how events are assigned to components in the new `name` variable. That is to say, if an event is more than *sigma* standard deviations from ALL of the components, you might expect it would be labeled as `{name}_None`. It is *not*. An event is only labeled `{name}_None` if it has a value that is outside of the channels' scales.

Optionally, if *posteriors* is `True`, *apply* creates a new `double` metadata variables named `{name}_1_posterior` ... `{name}_n_posterior` where `n` is the number of components. The column `{name}_i_posterior` contains the posterior probability that this event is a member of component `i`.

Finally, the same mixture model (mean and standard deviation) may not be appropriate for every subset of the data. If this is the case, you can use the *by* attribute to specify metadata by which to aggregate the data before estimating (and applying) a mixture model. The number of components must be the same across each subset, though.

name
The operation name; determines the name of the new metadata column

Type Str

channels
The channels to apply the mixture model to.

Type List(Str)

scale
Re-scale the data in the specified channels before fitting. If a channel is in *channels* but not in *scale*, the current package-wide default (set with *set_default_scale*) is used.

Type Dict(Str : {"linear", "logicle", "log"})

num_components
How many components to fit to the data? Must be a positive integer.

Type Int (default = 1)

sigma
If not None, use this operation as a "gate": for each component, create a new boolean variable `{name}_i` and if the event is within *sigma* standard deviations, set that variable to `True`. If *num_components* is `1`, must be > `0`.

Type Float

by
A list of metadata attributes to aggregate the data before estimating the model. For example, if the experiment has two pieces of metadata, `Time` and `Dox`, setting *by* to `["Time", "Dox"]` will fit the model separately to each subset of the data with a unique combination of `Time` and `Dox`.

Type List(Str)

posteriors

If `True`, add columns named `{name}_{i}_posterior` giving the posterior probability that the event is in component `i`. Useful for filtering out low-probability events.

Type Bool (default = False)

Notes

We use the Mahalnobis distance as a multivariate generalization of the number of standard deviations an event is from the mean of the multivariate gaussian. If $\vec{x}$ is an observation from a distribution with mean $\vec{\mu}$ and S is the covariance matrix, then the Mahalanobis distance is $\sqrt{(x - \mu)^T \cdot S^{-1} \cdot (x - \mu)}$.

Examples

Make a little data set.

```
>>> import cytoflow as flow
>>> import_op = flow.ImportOp()
>>> import_op.tubes = [flow.Tube(file = "Plate01/RFP_Well_A3.fcs",
...                              conditions = {'Dox' : 10.0}),
...                    flow.Tube(file = "Plate01/CFP_Well_A4.fcs",
...                              conditions = {'Dox' : 1.0})]
>>> import_op.conditions = {'Dox' : 'float'}
>>> ex = import_op.apply()
```

Create and parameterize the operation.

```
>>> gm_op = flow.GaussianMixtureOp(name = 'Gauss',
...                                channels = ['Y2-A'],
...                                scale = {'Y2-A' : 'log'},
...                                num_components = 2)
```

Estimate the clusters

```
>>> gm_op.estimate(ex)
```

Plot a diagnostic view

```
>>> gm_op.default_view().plot(ex)
```

Apply the gate

```
>>> ex2 = gm_op.apply(ex)
```

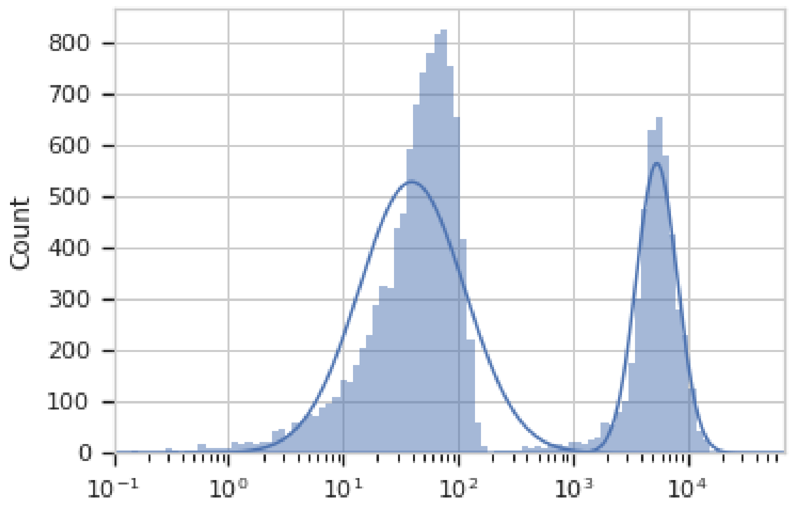
Plot a diagnostic view with the event assignments

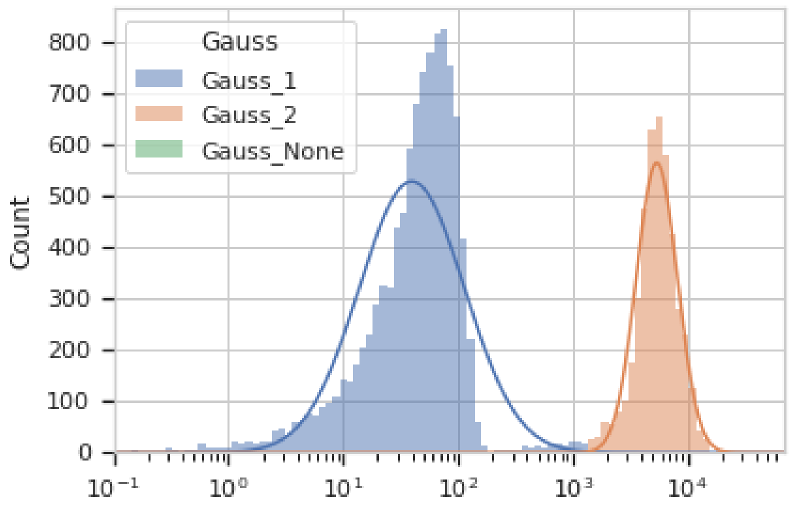
```
>>> gm_op.default_view().plot(ex2)
```

And with two channels:

```
>>> gm_op = flow.GaussianMixtureOp(name = 'Gauss',
...                                channels = ['V2-A', 'Y2-A'],
...                                scale = {'V2-A' : 'log',
...                                         'Y2-A' : 'log'},
```

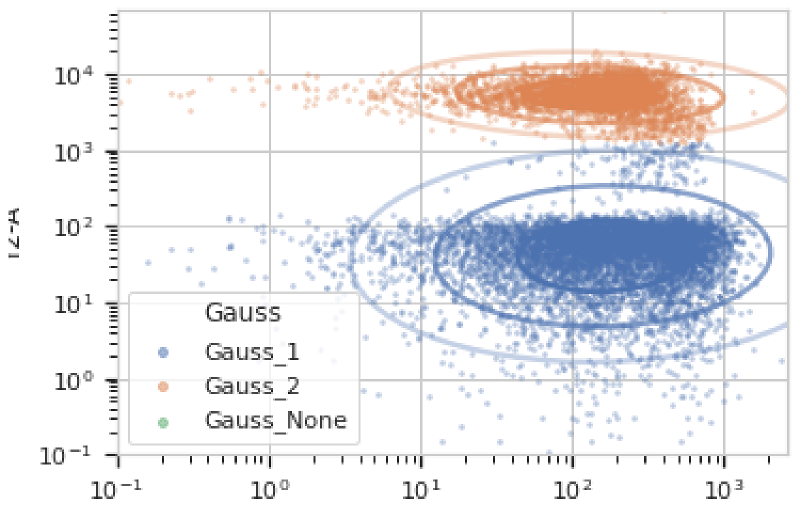
(continues on next page)


Count
0
100
200
300
400
500
600
700
800
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}


Gauss
Gauss_1
Gauss_2
Gauss_None
Count
0
100
200
300
400
500
600
700
800
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}

(continued from previous page)

```
...                                           num_components = 2)
>>> gm_op.estimate(ex)
>>> ex2 = gm_op.apply(ex)
>>> gm_op.default_view().plot(ex2)
```



estimate(*experiment*, *subset=None*)
Estimate the Gaussian mixture model parameters

Parameters

- **experiment** (*Experiment*) – The data to use to estimate the mixture parameters
- **subset** (*str (default = None)*) – If set, a Python expression to determine the subset of the data to use to in the estimation.

apply(*experiment*)
Assigns new metadata to events using the mixture model estimated in `estimate`.

Returns

A new `Experiment` with the new condition variables as described in the class documentation. Also adds the following new statistics:

- **mean** [Float] the mean of the fitted gaussian in each channel for each component.
- **sigma** [(Float, Float)] the locations the mean +/- one standard deviation in each channel for each component.
- **correlation** [Float] the correlation coefficient between each pair of channels for each component.
- **proportion** [Float] the proportion of events in each component of the mixture model. only added if `num_components` > 1.

Return type *Experiment*

default_view(***kwargs*)
Returns a diagnostic plot of the Gaussian mixture model.

Returns An `IView`, call `plot` to see the diagnostic plot.

Return type *IView*

class cytoflow.operations.gaussian.**GaussianMixture1DView**

Bases: *cytoflow.operations.base_op_views.By1DView*, *cytoflow.operations.base_op_views.AnnotatingView*, *cytoflow.views.histogram.HistogramView*

A default view for *GaussianMixtureOp* that plots the histogram of a single channel, then the estimated Gaussian distributions on top of it.

facets

A read-only list of the conditions used to facet this view.

Type List(Str)

by

A read-only list of the conditions used to group this view's data before plotting.

Type List(Str)

channel

The channel this view is viewing. If you created the view using *default_view*, this is already set.

Type Str

scale

The way to scale the x axes. If you created the view using *default_view*, this may be already set.

Type {'linear', 'log', 'logicle'}

op

The *IOperation* that this view is associated with. If you created the view using *default_view*, this is already set.

Type Instance(*IOperation*)

subset

An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

xfacet

Set to one of the *Experiment.conditions* in the *Experiment*, and a new column of subplots will be added for every unique value of that condition.

Type String

yfacet

Set to one of the *Experiment.conditions* in the *Experiment*, and a new row of subplots will be added for every unique value of that condition.

Type String

huefacet

Set to one of the *Experiment.conditions* in the in the *Experiment*, and a new color will be added to the plot for every unique value of that condition.

Type String

huescale

How should the color scale for *huefacet* be scaled?

Type {'linear', 'log', 'logicle'}

`plot`(*experiment*, ***kwargs*)

Plot the plots.

Parameters

- **experiment** (*Experiment*) – The `Experiment` to plot using this view.
- **title** (*str*) – Set the plot title
- **xlabel** (*str*) – Set the X axis label
- **ylabel** (*str*) – Set the Y axis label
- **huelabel** (*str*) – Set the label for the hue facet (in the legend)
- **legend** (*bool*) – Plot a legend for the color or hue facet? Defaults to `True`.
- **sharex** (*bool*) – If there are multiple subplots, should they share X axes? Defaults to `True`.
- **sharey** (*bool*) – If there are multiple subplots, should they share Y axes? Defaults to `True`.
- **row_order** (*list*) – Override the row facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **col_order** (*list*) – Override the column facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **hue_order** (*list*) – Override the hue facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **height** (*float*) – The height of *each row* in inches. Default = 3.0
- **aspect** (*float*) – The aspect ratio *of each subplot*. Default = 1.5
- **col_wrap** (*int*) – If `xfacet` is set and `yfacet` is not set, you can "wrap" the subplots around so that they form a multi-row grid by setting this to the number of columns you want.
- **sns_style** (*{"darkgrid", "whitegrid", "dark", "white", "ticks"}*) – Which `seaborn` style to apply to the plot? Default is `whitegrid`.
- **sns_context** (*{"paper", "notebook", "talk", "poster"}*) – Which `seaborn` context to use? Controls the scaling of plot elements such as tick labels and the legend. Default is `talk`.
- **palette** (*palette name, list, or dict*) – Colors to use for the different levels of the hue variable. Should be something that can be interpreted by `seaborn.color_palette`, or a dictionary mapping hue levels to matplotlib colors.
- **despine** (*Bool*) – Remove the top and right axes from the plot? Default is `True`.
- **min_quantile** (*float (>0.0 and <1.0, default = 0.001)*) – Clip data that is less than this quantile.
- **max_quantile** (*float (>0.0 and <1.0, default = 1.00)*) – Clip data that is greater than this quantile.
- **lim** (*(float, float)*) – Set the range of the plot's data axis.
- **orientation** (*{'vertical', 'horizontal'}*)
- **num_bins** (*int*) – The number of bins to plot in the histogram. Clipped to [100, 1000]
- **histtype** (*{'stepfilled', 'step', 'bar'}*) – The type of histogram to draw. `stepfilled` is the default, which is a line plot with a color filled under the curve.
- **density** (*bool*) – If `True`, re-scale the histogram to form a probability density function, so the area under the histogram is 1.

- **linewidth** (*float*) – The width of the histogram line (in points)
- **linestyle** (*['-' | '–' | '-.' | ':' | "None"]*) – The style of the line to plot
- **alpha** (*float (default = 0.5)*) – The alpha blending value, between 0 (transparent) and 1 (opaque).
- **color** (*matplotlib color*) – The color to plot the annotations. Overrides the default color cycle.
- **plot_name** (*Str*) – If this `IView` can make multiple plots, `plot_name` is the name of the plot to make. Must be one of the values retrieved from `enum_plots`.

`cytoflow.operations.gaussian.`**`poly_area`**(*x*, *y*)

`class cytoflow.operations.gaussian.`**`GaussianMixture2DView`**

Bases: `cytoflow.operations.base_op_views.By2DView`, `cytoflow.operations.base_op_views.AnnotatingView`, `cytoflow.views.scatterplot.ScatterplotView`

A default view for `GaussianMixtureOp` that plots the scatter plot of a two channels, then the estimated 2D Gaussian distributions on top of it.

`facets`

A read-only list of the conditions used to facet this view.

Type List(Str)

`by`

A read-only list of the conditions used to group this view's data before plotting.

Type List(Str)

`xchannel`

The channels to use for this view's X axis. If you created the view using `default_view`, this is already set.

Type Str

`ychannel`

The channels to use for this view's Y axis. If you created the view using `default_view`, this is already set.

Type Str

`xscale`

The way to scale the x axis. If you created the view using `default_view`, this may be already set.

Type {'linear', 'log', 'logicle'}

`yscale`

The way to scale the y axis. If you created the view using `default_view`, this may be already set.

Type {'linear', 'log', 'logicle'}

`op`

The `IOperation` that this view is associated with. If you created the view using `default_view`, this is already set.

Type Instance(`IOperation`)

`subset`

An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

`xfacet`
Set to one of the `Experiment.conditions` in the `Experiment`, and a new column of subplots will be added for every unique value of that condition.

Type String

`yfacet`
Set to one of the `Experiment.conditions` in the `Experiment`, and a new row of subplots will be added for every unique value of that condition.

Type String

`huefacet`
Set to one of the `Experiment.conditions` in the in the `Experiment`, and a new color will be added to the plot for every unique value of that condition.

Type String

`huescale`
How should the color scale for `huefacet` be scaled?

Type {'linear', 'log', 'logicle'}

`plot`(*experiment*, ***kwargs*)
Plot the plots.

Parameters

- **experiment** (*Experiment*) – The `Experiment` to plot using this view.
- **title** (*str*) – Set the plot title
- **xlabel** (*str*) – Set the X axis label
- **ylabel** (*str*) – Set the Y axis label
- **huelabel** (*str*) – Set the label for the hue facet (in the legend)
- **legend** (*bool*) – Plot a legend for the color or hue facet? Defaults to `True`.
- **sharex** (*bool*) – If there are multiple subplots, should they share X axes? Defaults to `True`.
- **sharey** (*bool*) – If there are multiple subplots, should they share Y axes? Defaults to `True`.
- **row_order** (*list*) – Override the row facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **col_order** (*list*) – Override the column facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **hue_order** (*list*) – Override the hue facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **height** (*float*) – The height of *each row* in inches. Default = 3.0
- **aspect** (*float*) – The aspect ratio *of each subplot*. Default = 1.5
- **col_wrap** (*int*) – If `xfacet` is set and `yfacet` is not set, you can "wrap" the subplots around so that they form a multi-row grid by setting this to the number of columns you want.
- **sns_style** (*{"darkgrid", "whitegrid", "dark", "white", "ticks"}*) – Which `seaborn` style to apply to the plot? Default is `whitegrid`.
- **sns_context** (*{"paper", "notebook", "talk", "poster"}*) – Which `seaborn` context to use? Controls the scaling of plot elements such as tick labels and the legend. Default is `talk`.

- **palette** (*palette name, list, or dict*) – Colors to use for the different levels of the hue variable. Should be something that can be interpreted by `seaborn.color_palette`, or a dictionary mapping hue levels to matplotlib colors.
- **despine** (*Bool*) – Remove the top and right axes from the plot? Default is `True`.
- **min_quantile** (*float (>0.0 and <1.0, default = 0.001)*) – Clip data that is less than this quantile.
- **max_quantile** (*float (>0.0 and <1.0, default = 1.00)*) – Clip data that is greater than this quantile.
- **xlim** (*(float, float)*) – Set the range of the plot's X axis.
- **ylim** (*(float, float)*) – Set the range of the plot's Y axis.
- **alpha** (*float (default = 0.25)*) – The alpha blending value, between 0 (transparent) and 1 (opaque).
- **s** (*int (default = 2)*) – The size in points^2.
- **marker** (*a matplotlib marker style, usually a string*) – Specfies the glyph to draw for each point on the scatterplot. See matplotlib.markers for examples. Default: 'o'
- **color** (*matplotlib color*) – The color to plot the annotations. Overrides the default color cycle.
- **plot_name** (*Str*) – If this *`IView`* can make multiple plots, `plot_name` is the name of the plot to make. Must be one of the values retrieved from *`enum_plots`*.

cytoflow.operations.i_operation

`i_operation` contains just one class:

`IOperation` – an `traits.has_traits.Interface` that all operation classes must implement.

class `cytoflow.operations.i_operation.`**`IOperation`**(*adaptee*, *default=<class 'traits.adaptation.adaptation_error.AdaptationError'>*)

Bases: `traits.has_traits.Interface`

The basic interface for an operation on cytometry data.

`id`

a unique identifier for this class. prefix: `edu.mit.synbio.cytoflow.operations`

Type Str

`friendly_id`

The operation's human-readable id (like `Threshold` or `K-means`). Used for UI implementations.

Type Str

`name`

The name of this IOperation instance (like `Debris_Filter`). Useful for UI implementations; sometimes used for naming gates' metadata

Type Str

`estimate`(*experiment*, *subset=None*)

Estimate this operation's parameters from some data.

For operations that are data-driven (for example, a mixture model), estimate the operation's parameters from an experiment.

Parameters

- **experiment** (`Experiment`) – the `Experiment` to use in the estimation.
- **subset** (*Str (optional)*) – a string passed to `pandas.DataFrame.query` to select the subset of data on which to run the parameter estimation.

Raises **`CytoflowOpException`** – If the operation can't be be completed because of bad op parameters.

`apply`(*experiment*)

Apply an operation to an experiment.

Parameters **experiment** (`Experiment`) – the `Experiment` to apply this op to

Returns the old `Experiment` with this operation applied

Return type `Experiment`

Raises **`CytoflowOpException`** – If the operation can't be be completed because of bad op parameters.

`default_view`(***kwargs*)

Many operations have a "default" view. This can either be a diagnostic for the operation's `estimate` method, an interactive for setting gates, etc. Frequently it makes sense to link the properties of the view to the properties of the `IOperation`; sometimes, `default_view` is the only way to get the view (ie, it's not useful when it doesn't reference an `IOperation` instance.)

Parameters ****kwargs** (*Dict*) – The keyword args passed to the view's constructor

Returns the `IView` instance

Return type `IView`

cytoflow.operations.import_op

`import_op` has two classes:

`Tube` – represents a tube in a flow cytometry experiment – an FCS file name and a dictionary of experimental conditions.

`ImportOp` – the operation that actually creates a new `Experiment` from a list of `Tube`.

There are a few utility functions as well:

- `parse_tube` – parse an FCS file.
- `check_tube` – checks an FCS file's parameters against an `Experiment`.
- `autodetect_name_metadata` – see if `$PnN` or `$PnS` has the channel names

class `cytoflow.operations.import_op.`**`Tube`**

Bases: `traits.has_traits.HasTraits`

Represents a tube or plate well we want to import.

`file`

The file name of the FCS file to import

Type File

`conditions`

A dictionary containing this tube's experimental conditions. Keys are condition names, values are condition values.

Type Dict(Str, Any)

Examples

```
>>> tube1 = flow.Tube(file = 'RFP_Well_A3.fcs', conditions = {"Dox" : 10.0})
>>> tube2 = flow.Tube(file='CFP_Well_A4.fcs', conditions = {"Dox" : 1.0})
```

conditions_equal(*other*)

class cytoflow.operations.import_op.**ImportOp**
Bases: `traits.has_traits.HasStrictTraits`

An operation for importing data and making an *Experiment*.

To use, set the *conditions* dict to a mapping between condition name and NumPy `dtype`. Useful dtypes include `category`, `float`, `int`, `bool`.

Next, set *tubes* to a list of *Tube* containing FCS filenames and the corresponding conditions.

If you would rather not analyze every single event in every FCS file, set *events* to the number of events from each FCS file you want to load.

Call *apply* to load the data. The usual `experiment` parameter can be `None`.

conditions
A dictionary mapping condition names (keys) to NumPy `dtype``s (values). Useful ``dtype``s include ``category`, `float`, `int`, and `bool`.

Type Dict(Str, Str)

tubes
A list of `Tube` instances, which map FCS files to their corresponding experimental conditions. Each `Tube` must have a `Tube.conditions` dict whose keys match those of *conditions*.

Type List(*Tube*)

channels
If you only need a subset of the channels available in the data set, specify them here. Each `(key, value)` pair specifies a channel to include in the output experiment. The key is the channel name in the FCS file, and the value is the name of the channel in the Experiment. You can use this to rename channels as you import data (because flow channel names are frequently not terribly informative.) New channel names must be valid Python identifiers: start with a letter or _, and all characters must be letters, numbers or _. If *channels* is empty, load all channels in the FCS files.

Type Dict(Str, Str)

events
If not None, import only a random subset of events of size *events*. Presumably the analysis will go faster but less precisely; good for interactive data exploration. Then, unset *events* and re-run the analysis non-interactively.

Type Int

name_metadata
Which FCS metadata is the channel name? If `None`, attempt to autodetect.

Type {None, “$PnN”, “$PnS”} (default = None)

data_set
The FCS standard allows you to encode multiple data sets in a single FCS file. Some software (such as the Beckman-Coulter software) also encode the same data in two different formats – for example, FCS2.0 and FCS3.0. To access a data set other than the first one, set *data_set* to the 0-based index of the data set you would like to use. This will be used for *all FCS files imported by this operation.*

Type Int (default = 0)

ignore_v

cytoflow is designed to operate on an *Experiment* containing tubes that were all collected under the same instrument settings. In particular, the same PMT voltages ensure that data can be compared across samples.

Very rarely, you may need to set up an *Experiment* with different voltage settings on different *Tube* instances. This is likely only to be the case when you are trying to figure out which voltages should be used in future experiments. If so, set *ignore_v* to a list of channel names to ignore particular channels.

Warning: THIS WILL BREAK REAL EXPERIMENTS

Type List(Str)

Examples

```
>>> tube1 = flow.Tube(file = 'RFP_Well_A3.fcs', conditions = {"Dox" : 10.0})
>>> tube2 = flow.Tube(file='CFP_Well_A4.fcs', conditions = {"Dox" : 1.0})
>>> import_op = flow.ImportOp(conditions = {"Dox" : "float"},
...                           tubes = [tube1, tube2])
>>> ex = import_op.apply()
```

apply(*experiment=None*, *metadata_only=False*)

Load a new *Experiment*.

Parameters

- **experiment** (*Experiment*) – Ignored
- **metadata_only** (*bool (default = False)*) – Only "import" the metadata, creating an Experiment with all the expected metadata and structure but 0 events.

Returns

The new *Experiment*. New channels have the following metadata:

- **voltage - int** The voltage that this channel was collected at. Determined by the `$PnV` field from the first FCS file.
- **range - int** The maximum range of this channel. Determined by the `$PnR` field from the first FCS file.

New experimental conditions do not have **voltage** or **range** metadata, obviously. Instead, they have **experiment** set to `True`, to distinguish the experimental variables from the conditions that were added by gates, etc.

If *ignore_v* is set, it is added as a key to the *Experiment*-wide metadata.

Return type *Experiment*

cytoflow.operations.import_op.**check_tube**(*filename*, *experiment*, *data_set=0*)

Check to see if an FCS file can be parsed, and that the tube's parameters are the same as those already in the *Experiment*. If not, raises *CytoflowError*. At the moment, only checks $PnV, the detector voltages.

Parameters

- **filename** (*string*) – An FCS filename
- **experiment** (*Experiment*) – The *Experiment* to check `filename` against.

- **data_set** (*int (optional, default = 0)*) – The FCS standard allows for multiple data sets; `data_set` specifies which one to check.

Raises *CytoflowError* – If the FCS file can't be read, or if the voltages in `filename` are different than those in `experiment`.

`cytoflow.operations.import_op.`**`autodetect_name_metadata`**(*filename*, *data_set=0*)

Tries to determine whether the channel names should come from $PnN or $PnS.

Parameters

- **filename** (*string*) – The name of the FCS file to operate on
- **data_set** (*int (optional, default = 0)*) – Which data set in the FCS file to operate on

Returns

- *The name of the parameter to parse channel names from,*
- *either "$PnN" or "$PnS"*

`cytoflow.operations.import_op.`**`parse_tube`**(*filename*, *experiment=None*, *data_set=0*, *metadata_only=False*)

Parses an FCS file. A thin wrapper over `fcsparser.parse`.

Parameters

- **filename** (*string*) – The file to parse.
- **experiment** (*Experiment* (optional, default: None)) – If provided, check the tube's parameters against this experiment first.
- **data_set** (*int (optional, default: 0)*) – Which data set in the FCS file to parse?
- **metadata_only** (*bool (optional, default: False)*) – If `True`, only parse the metadata. Because this is at the beginning of the FCS file, this happens much faster than parsing the entire file.

Returns

- **tube_metadata** (*dict*) – The metadata from the FCS file
- **tube_data** (`pandas.DataFrame`) – The actual tabular data from the FCS file. Each row is an event, and each column is a channel.

cytoflow.operations.kmeans

Use k-means clustering to cluster events in any number of dimensions. *kmeans* has three classes:

KMeansOp – the *IOperation* to perform the clustering.

KMeans1DView – a diagnostic view of the clustering (1D, using a histogram)

KMeans2DView – a diagnostic view of the clustering (2D, using a scatterplot)

class `cytoflow.operations.kmeans.`**`KMeansOp`**

Bases: `traits.has_traits.HasStrictTraits`

Use a K-means clustering algorithm to cluster events.

Call *estimate* to compute the cluster centroids.

Calling *apply* creates a new categorical metadata variable named *name*, with possible values `{name}_1` .... `name_n` where `n` is the number of clusters, specified with *num_clusters*.

The same model may not be appropriate for different subsets of the data set. If this is the case, you can use the *by* attribute to specify metadata by which to aggregate the data before estimating (and applying) a model. The number of clusters is the same across each subset, though.

name

The operation name; determines the name of the new metadata column

Type Str

channels

The channels to apply the clustering algorithm to.

Type List(Str)

scale

Re-scale the data in the specified channels before fitting. If a channel is in *channels* but not in *scale*, the current package-wide default (set with *set_default_scale*) is used.

Note: Sometimes you may see events labeled `{name}_None` – this results from events for which the selected scale is invalid. For example, if an event has a negative measurement in a channel and that channel's scale is set to "log", this event will be set to `{name}_None`.

Type Dict(Str : {"linear", "logicle", "log"})

num_clusters

How many components to fit to the data? Must be a positive integer.

Type Int (default = 2)

by

A list of metadata attributes to aggregate the data before estimating the model. For example, if the experiment has two pieces of metadata, `Time` and `Dox`, setting *by* to `["Time", "Dox"]` will fit the model separately to each subset of the data with a unique combination of `Time` and `Dox`.

Type List(Str)

Examples

Make a little data set.

```
>>> import cytoflow as flow
>>> import_op = flow.ImportOp()
>>> import_op.tubes = [flow.Tube(file = "Plate01/RFP_Well_A3.fcs",
...                               conditions = {'Dox' : 10.0}),
...                     flow.Tube(file = "Plate01/CFP_Well_A4.fcs",
...                               conditions = {'Dox' : 1.0})]
>>> import_op.conditions = {'Dox' : 'float'}
>>> ex = import_op.apply()
```

Create and parameterize the operation.

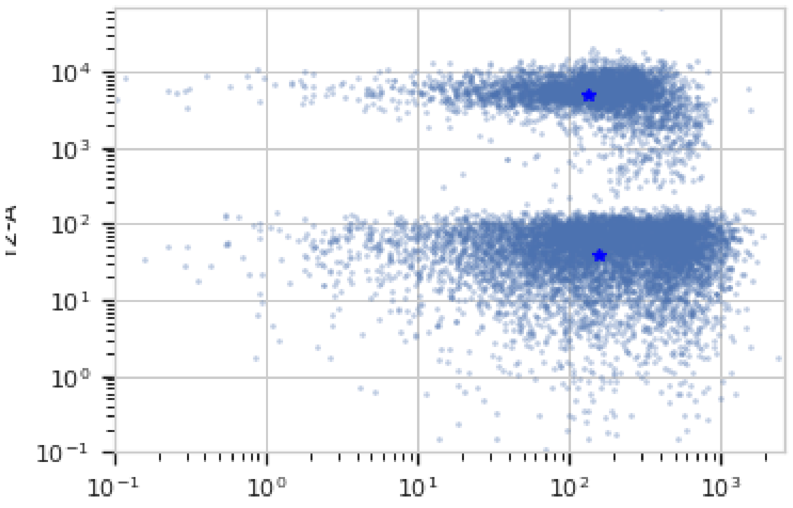
```
>>> km_op = flow.KMeansOp(name = 'KMeans',
...                       channels = ['V2-A', 'Y2-A'],
...                       scale = {'V2-A' : 'log',
...                                'Y2-A' : 'log'},
...                       num_clusters = 2)
```

Estimate the clusters

```
>>> km_op.estimate(ex)
```

Plot a diagnostic view

```
>>> km_op.default_view().plot(ex)
```

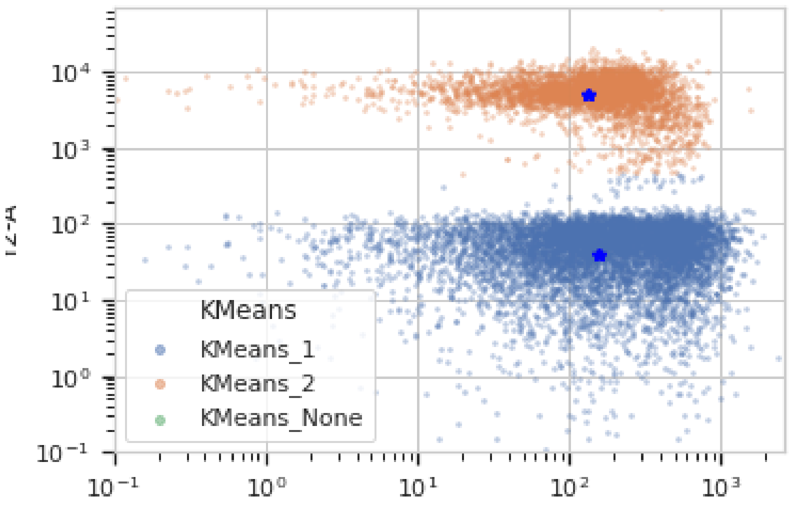


Apply the gate

```
>>> ex2 = km_op.apply(ex)
```

Plot a diagnostic view with the event assignments

```
>>> km_op.default_view().plot(ex2)
```



estimate(*experiment*, *subset=None*)

Estimate the k-means clusters

Parameters

- **experiment** (*Experiment*) – The *Experiment* to use to estimate the k-means clusters
- **subset** (*str (default = None)*) – A Python expression that specifies a subset of the data in `experiment` to use to parameterize the operation.

apply(*experiment*)

Apply the KMeans clustering to the data.

Returns

a new Experiment with one additional entry in *Experiment.conditions* named *name*, of type `category`. The new category has values `name_1`, `name_2`, etc to indicate which k-means cluster an event is a member of.

The new *Experiment* also has one new statistic called `centers`, which is a list of tuples encoding the centroids of each k-means cluster.

Return type *Experiment*

default_view(***kwargs*)

Returns a diagnostic plot of the k-means clustering.

Returns **IView**

Return type an IView, call *KMeans1DView.plot* to see the diagnostic plot.

class cytoflow.operations.kmeans.**KMeans1DView**

Bases: *cytoflow.operations.base_op_views.By1DView*, *cytoflow.operations.base_op_views.AnnotatingView*, *cytoflow.views.histogram.HistogramView*

A diagnostic view for *KMeansOp* (1D, using a histogram)

op

The op whose parameters we're viewing.

Type Instance(*KMeansOp*)

facets

A read-only list of the conditions used to facet this view.

Type List(Str)

by

A read-only list of the conditions used to group this view's data before plotting.

Type List(Str)

channel

The channel this view is viewing. If you created the view using *default_view*, this is already set.

Type Str

scale

The way to scale the x axes. If you created the view using *default_view*, this may be already set.

Type {'linear', 'log', 'logicle'}

subset

An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

`xfacet`
Set to one of the *`Experiment.conditions`* in the *`Experiment`*, and a new column of subplots will be added for every unique value of that condition.

Type String

`yfacet`
Set to one of the *`Experiment.conditions`* in the *`Experiment`*, and a new row of subplots will be added for every unique value of that condition.

Type String

`huefacet`
Set to one of the *`Experiment.conditions`* in the in the *`Experiment`*, and a new color will be added to the plot for every unique value of that condition.

Type String

`huescale`
How should the color scale for *`huefacet`* be scaled?

Type {'linear', 'log', 'logicle'}

`plot`(*experiment*, ***kwargs*)
Plot the plots.

Parameters

- **experiment** (*Experiment*) – The *`Experiment`* to plot using this view.
- **title** (*str*) – Set the plot title
- **xlabel** (*str*) – Set the X axis label
- **ylabel** (*str*) – Set the Y axis label
- **huelabel** (*str*) – Set the label for the hue facet (in the legend)
- **legend** (*bool*) – Plot a legend for the color or hue facet? Defaults to `True`.
- **sharex** (*bool*) – If there are multiple subplots, should they share X axes? Defaults to `True`.
- **sharey** (*bool*) – If there are multiple subplots, should they share Y axes? Defaults to `True`.
- **row_order** (*list*) – Override the row facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **col_order** (*list*) – Override the column facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **hue_order** (*list*) – Override the hue facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **height** (*float*) – The height of *each row* in inches. Default = 3.0
- **aspect** (*float*) – The aspect ratio *of each subplot*. Default = 1.5
- **col_wrap** (*int*) – If *`xfacet`* is set and *`yfacet`* is not set, you can "wrap" the subplots around so that they form a multi-row grid by setting this to the number of columns you want.
- **sns_style** (*{"darkgrid", "whitegrid", "dark", "white", "ticks"}*) – Which `seaborn` style to apply to the plot? Default is `whitegrid`.

- **sns_context** (*{"paper", "notebook", "talk", "poster"}*) – Which `seaborn` context to use? Controls the scaling of plot elements such as tick labels and the legend. Default is `talk`.
- **palette** (*palette name, list, or dict*) – Colors to use for the different levels of the hue variable. Should be something that can be interpreted by `seaborn.color_palette`, or a dictionary mapping hue levels to matplotlib colors.
- **despine** (*Bool*) – Remove the top and right axes from the plot? Default is `True`.
- **min_quantile** (*float (>0.0 and <1.0, default = 0.001)*) – Clip data that is less than this quantile.
- **max_quantile** (*float (>0.0 and <1.0, default = 1.00)*) – Clip data that is greater than this quantile.
- **lim** (*(float, float)*) – Set the range of the plot's data axis.
- **orientation** (*{'vertical', 'horizontal'}*)
- **num_bins** (*int*) – The number of bins to plot in the histogram. Clipped to [100, 1000]
- **histtype** (*{'stepfilled', 'step', 'bar'}*) – The type of histogram to draw. `stepfilled` is the default, which is a line plot with a color filled under the curve.
- **density** (*bool*) – If `True`, re-scale the histogram to form a probability density function, so the area under the histogram is 1.
- **linewidth** (*float*) – The width of the histogram line (in points)
- **linestyle** (*['-' | '–' | '-.' | ':' | "None"]*) – The style of the line to plot
- **alpha** (*float (default = 0.5)*) – The alpha blending value, between 0 (transparent) and 1 (opaque).
- **color** (*matplotlib color*) – The color to plot the annotations. Overrides the default color cycle.
- **plot_name** (*Str*) – If this `IView` can make multiple plots, `plot_name` is the name of the plot to make. Must be one of the values retrieved from `enum_plots`.

`class cytoflow.operations.kmeans.KMeans2DView`

Bases: `cytoflow.operations.base_op_views.By2DView`, `cytoflow.operations.base_op_views.AnnotatingView`, `cytoflow.views.scatterplot.ScatterplotView`

A diagnostic view for `KMeansOp` (2D, using a scatterplot).

`op`

The op whose parameters we're viewing.

Type Instance(*KMeansOp*)

`facets`

A read-only list of the conditions used to facet this view.

Type List(Str)

`by`

A read-only list of the conditions used to group this view's data before plotting.

Type List(Str)

`xchannel`

The channels to use for this view's X axis. If you created the view using `default_view`, this is already set.

Type Str

`ychannel`
The channels to use for this view's Y axis. If you created the view using *`default_view`*, this is already set.

Type Str

`xscale`
The way to scale the x axis. If you created the view using *`default_view`*, this may be already set.

Type {'linear', 'log', 'logicle'}

`yscale`
The way to scale the y axis. If you created the view using *`default_view`*, this may be already set.

Type {'linear', 'log', 'logicle'}

`subset`
An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

`xfacet`
Set to one of the *`Experiment.conditions`* in the *`Experiment`*, and a new column of subplots will be added for every unique value of that condition.

Type String

`yfacet`
Set to one of the *`Experiment.conditions`* in the *`Experiment`*, and a new row of subplots will be added for every unique value of that condition.

Type String

`huefacet`
Set to one of the *`Experiment.conditions`* in the in the *`Experiment`*, and a new color will be added to the plot for every unique value of that condition.

Type String

`huescale`
How should the color scale for *`huefacet`* be scaled?

Type {'linear', 'log', 'logicle'}

`plot`(*experiment*, ***kwargs*)
Plot the plots.

Parameters

- **experiment** (*Experiment*) – The *`Experiment`* to plot using this view.
- **title** (*str*) – Set the plot title
- **xlabel** (*str*) – Set the X axis label
- **ylabel** (*str*) – Set the Y axis label
- **huelabel** (*str*) – Set the label for the hue facet (in the legend)
- **legend** (*bool*) – Plot a legend for the color or hue facet? Defaults to `True`.
- **sharex** (*bool*) – If there are multiple subplots, should they share X axes? Defaults to `True`.
- **sharey** (*bool*) – If there are multiple subplots, should they share Y axes? Defaults to `True`.

- **row_order** (*list*) – Override the row facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **col_order** (*list*) – Override the column facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **hue_order** (*list*) – Override the hue facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **height** (*float*) – The height of *each row* in inches. Default = 3.0
- **aspect** (*float*) – The aspect ratio *of each subplot*. Default = 1.5
- **col_wrap** (*int*) – If `xfacet` is set and `yfacet` is not set, you can "wrap" the subplots around so that they form a multi-row grid by setting this to the number of columns you want.
- **sns_style** (*{"darkgrid", "whitegrid", "dark", "white", "ticks"}*) – Which `seaborn` style to apply to the plot? Default is `whitegrid`.
- **sns_context** (*{"paper", "notebook", "talk", "poster"}*) – Which `seaborn` context to use? Controls the scaling of plot elements such as tick labels and the legend. Default is `talk`.
- **palette** (*palette name, list, or dict*) – Colors to use for the different levels of the hue variable. Should be something that can be interpreted by `seaborn.color_palette`, or a dictionary mapping hue levels to matplotlib colors.
- **despine** (*Bool*) – Remove the top and right axes from the plot? Default is `True`.
- **min_quantile** (*float (>0.0 and <1.0, default = 0.001)*) – Clip data that is less than this quantile.
- **max_quantile** (*float (>0.0 and <1.0, default = 1.00)*) – Clip data that is greater than this quantile.
- **xlim** (*(float, float)*) – Set the range of the plot's X axis.
- **ylim** (*(float, float)*) – Set the range of the plot's Y axis.
- **alpha** (*float (default = 0.25)*) – The alpha blending value, between 0 (transparent) and 1 (opaque).
- **s** (*int (default = 2)*) – The size in points^2.
- **marker** (*a matplotlib marker style, usually a string*) – Specfies the glyph to draw for each point on the scatterplot. See matplotlib.markers for examples. Default: 'o'
- **color** (*matplotlib color*) – The color to plot the annotations. Overrides the default color cycle.
- **plot_name** (*Str*) – If this `IView` can make multiple plots, `plot_name` is the name of the plot to make. Must be one of the values retrieved from `enum_plots`.

cytoflow.operations.pca

Apply principal component analysis (PCA) to flow data – decomposes the multivariate data set into orthogonal components that explain the maximum amount of variance. `pca` has one class:

`PCAOp` – the `IOperation` that applies PCA to an `Experiment`.

class `cytoflow.operations.pca.PCAOp`

Bases: `traits.has_traits.HasStrictTraits`

Use principal components analysis (PCA) to decompose a multivariate data set into orthogonal components that explain a maximum amount of variance.

Call `estimate` to compute the optimal decomposition.

Calling `apply` creates new "channels" named `{name}_1 ... {name}_n`, where `name` is the `name` attribute and `n` is `num_components`.

The same decomposition may not be appropriate for different subsets of the data set. If this is the case, you can use the `by` attribute to specify metadata by which to aggregate the data before estimating (and applying) a model. The PCA parameters such as the number of components and the kernel are the same across each subset, though.

name

The operation name; determines the name of the new columns.

Type Str

channels

The channels to apply the decomposition to.

Type List(Str)

scale

Re-scale the data in the specified channels before fitting. If a channel is in `channels` but not in `scale`, the current package-wide default (set with `set_default_scale`) is used.

Type Dict(Str : {"linear", "logicle", "log"})

num_components

How many components to fit to the data? Must be a positive integer.

Type Int (default = 2)

by

A list of metadata attributes to aggregate the data before estimating the model. For example, if the experiment has two pieces of metadata, `Time` and `Dox`, setting `by` to `["Time", "Dox"]` will fit the model separately to each subset of the data with a unique combination of `Time` and `Dox`.

Type List(Str)

whiten

Scale each component to unit variance? May be useful if you will be using unsupervized clustering (such as K-means).

Type Bool (default = False)

Examples

Make a little data set.

```
>>> import cytoflow as flow
>>> import_op = flow.ImportOp()
>>> import_op.tubes = [flow.Tube(file = "Plate01/RFP_Well_A3.fcs",
...                               conditions = {'Dox' : 10.0}),
...                     flow.Tube(file = "Plate01/CFP_Well_A4.fcs",
...                               conditions = {'Dox' : 1.0})]
>>> import_op.conditions = {'Dox' : 'float'}
>>> ex = import_op.apply()
```

Create and parameterize the operation.

```
>>> pca = flow.PCAOp(name = 'PCA',
...                  channels = ['V2-A', 'V2-H', 'Y2-A', 'Y2-H'],
...                  scale = {'V2-A' : 'log',
...                           'V2-H' : 'log',
...                           'Y2-A' : 'log',
...                           'Y2-H' : 'log'},
...                  num_components = 2,
...                  by = ["Dox"])
```

Estimate the decomposition

```
>>> pca.estimate(ex)
```

Apply the operation

```
>>> ex2 = pca.apply(ex)
```

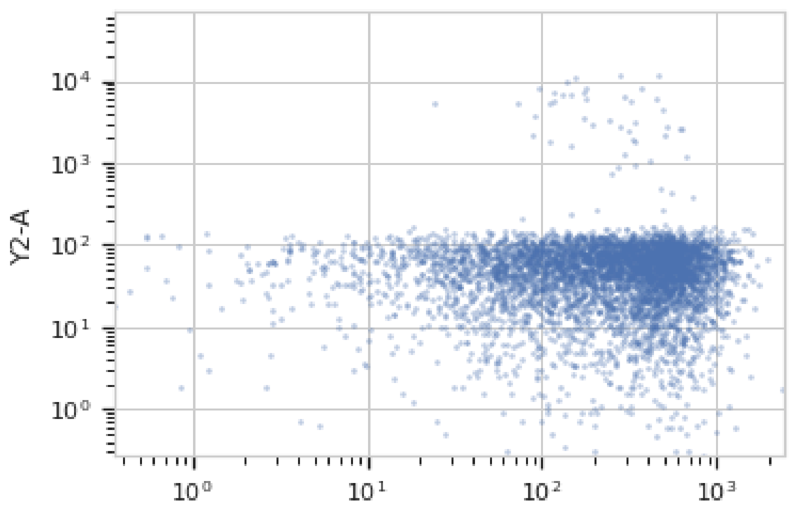
Plot a scatterplot of the PCA. Compare to a scatterplot of the underlying channels.

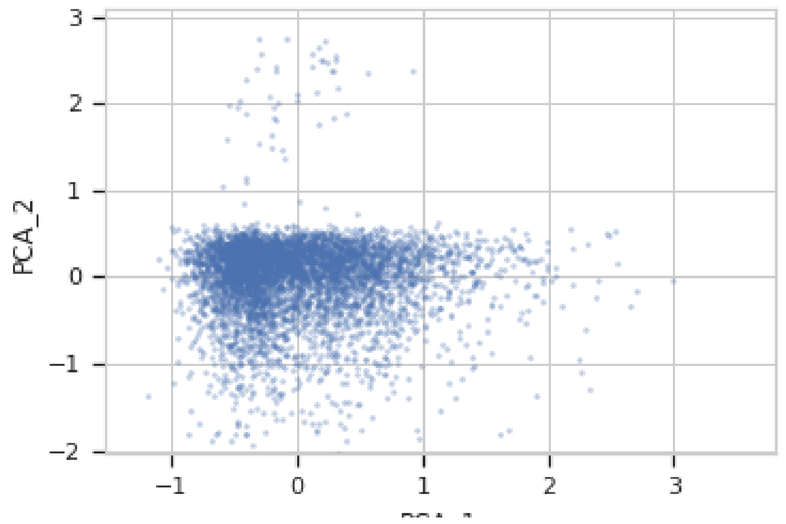
```
>>> flow.ScatterplotView(xchannel = "V2-A",
...                      xscale = "log",
...                      ychannel = "Y2-A",
...                      yscale = "log",
...                      subset = "Dox == 1.0").plot(ex2)
```

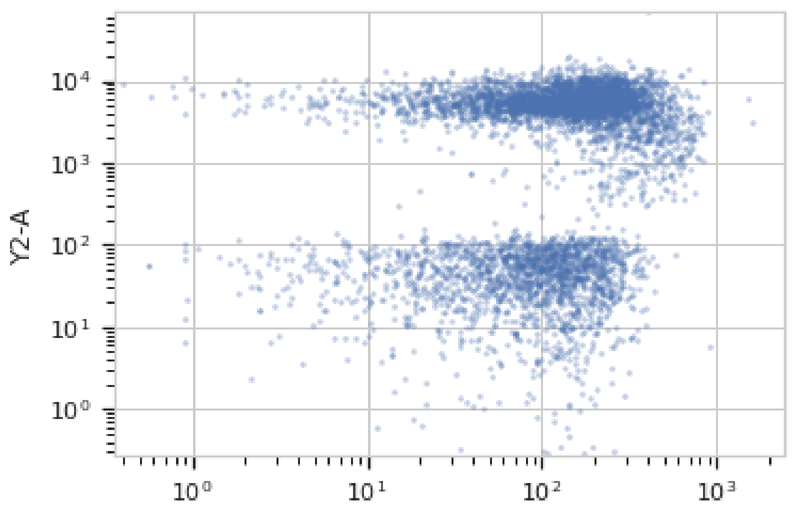
```
>>> flow.ScatterplotView(xchannel = "PCA_1",
...                      ychannel = "PCA_2",
...                      subset = "Dox == 1.0").plot(ex2)
```

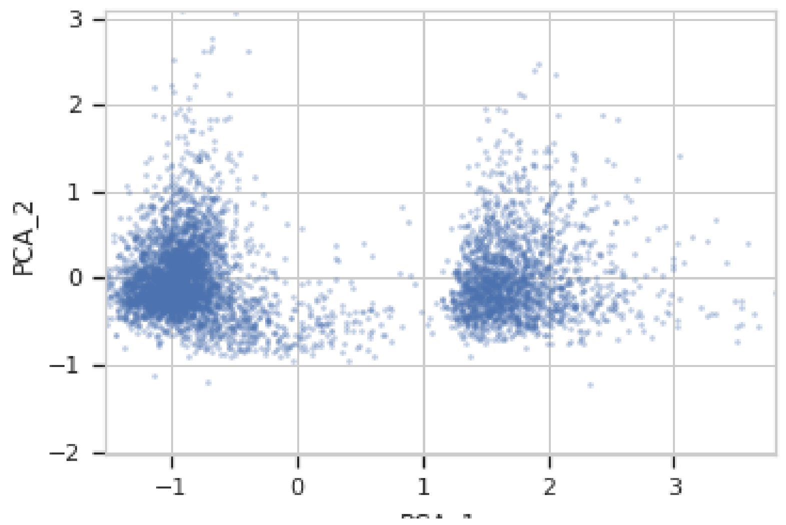
```
>>> flow.ScatterplotView(xchannel = "V2-A",
...                      xscale = "log",
...                      ychannel = "Y2-A",
...                      yscale = "log",
...                      subset = "Dox == 10.0").plot(ex2)
```

```
>>> flow.ScatterplotView(xchannel = "PCA_1",
...                      ychannel = "PCA_2",
...                      subset = "Dox == 10.0").plot(ex2)
```


Y2-A
10^4
10^3
10^2
10^1
10^0
10^0
10^1
10^2
10^3


PCA_2
3
2
1
0
−1
−2
−1
0
1
2
3


Y2-A
10^4
10^3
10^2
10^1
10^0
10^0
10^1
10^2
10^3


PCA_2
3
2
1
0
−1
−2
−1
0
1
2
3

`estimate`(*experiment*, *subset=None*)

Estimate the decomposition

Parameters

- **experiment** (*Experiment*) – The `Experiment` to use to estimate the k-means clusters
- **subset** (*str (default = None)*) – A Python expression that specifies a subset of the data in `experiment` to use to parameterize the operation.

`apply`(*experiment*)

Apply the PCA decomposition to the data.

Returns a new Experiment with additional `Experiment.channels` named `name_1 ... name_n`

Return type *Experiment*

cytoflow.operations.polygon

Apply a polygon gate to two channels in an `Experiment`. `polygon` has two classes:

`PolygonOp` – Applies the gate, given a set of vertices.

`ScatterplotPolygonSelectionView` – an `IView` that allows you to view the polygon and/or interactively set the vertices on a scatterplot.

`DensityPolygonSelectionView` – an `IView` that allows you to view the polygon and/or interactively set the vertices on a scatterplot.

class cytoflow.operations.polygon.**PolygonOp**

Bases: `traits.has_traits.HasStrictTraits`

Apply a polygon gate to a cytometry experiment.

name

The operation name. Used to name the new metadata field in the experiment that's created by `apply`

Type Str

xchannel, ychannel

The names of the x and y channels to apply the gate.

Type Str

xscale, yscale

The scales applied to the data before drawing the polygon.

Type {'linear', 'log', 'logicle'} (default = 'linear')

vertices

The polygon verticies. An ordered list of 2-tuples, representing the x and y coordinates of the vertices.

Type List((Float, Float))

Notes

You can set the verticies by hand, I suppose, but it's much easier to use the interactive view you get from *default_view* to do so. Set *ScatterplotPolygonSelectionView.interactive* to `True`, then single-click to set vertices. Click the first vertex a second time to close the polygon. You'll need to do this in a Jupyter notebook with `%matplotlib notebook` – see the `Interactive Plots` demo for an example.

Examples

Make a little data set.

```
>>> import cytoflow as flow
>>> import_op = flow.ImportOp()
>>> import_op.tubes = [flow.Tube(file = "Plate01/RFP_Well_A3.fcs",
...                              conditions = {'Dox' : 10.0}),
...                    flow.Tube(file = "Plate01/CFP_Well_A4.fcs",
...                              conditions = {'Dox' : 1.0})]
>>> import_op.conditions = {'Dox' : 'float'}
>>> ex = import_op.apply()
```

Create and parameterize the operation.

```
>>> p = flow.PolygonOp(name = "Polygon",
...                    xchannel = "V2-A",
...                    ychannel = "Y2-A")
>>> p.vertices = [(23.411982294776319, 5158.7027015021222),
...               (102.22182270573683, 23124.058843387455),
...               (510.94519955277201, 23124.058843387455),
...               (1089.5215641232173, 3800.3424832180476),
...               (340.56382570202402, 801.98947404942271),
...               (65.42597937575897, 1119.3133482602157)]
```

Show the default view.

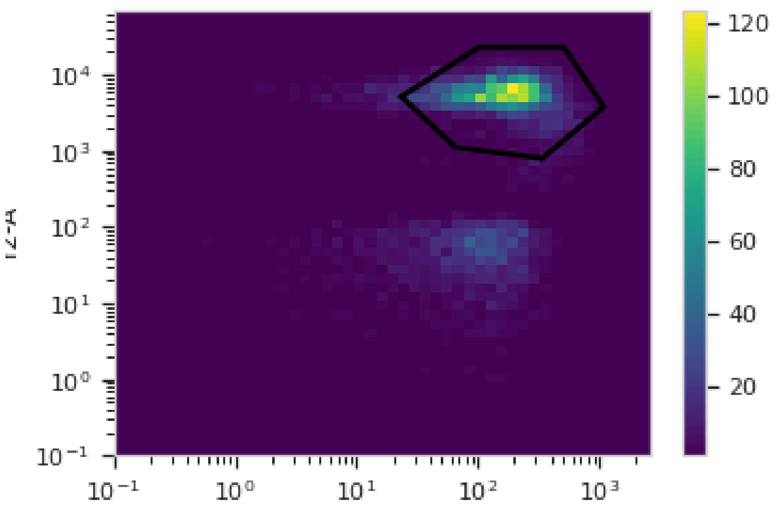
```
>>> df = p.default_view(huefacet = "Dox",
...                     xscale = 'log',
...                     yscale = 'log',
...                     density = True)
```

```
>>> df.plot(ex)
```

Note: If you want to use the interactive default view in a Jupyter notebook, make sure you say `%matplotlib notebook` in the first cell (instead of `%matplotlib inline` or similar). Then call *default_view* with `interactive = True`:

```
df = p.default_view(huefacet = "Dox",
                    xscale = 'log',
                    yscale = 'log',
                    interactive = True)
df.plot(ex)
```

Apply the gate, and show the result

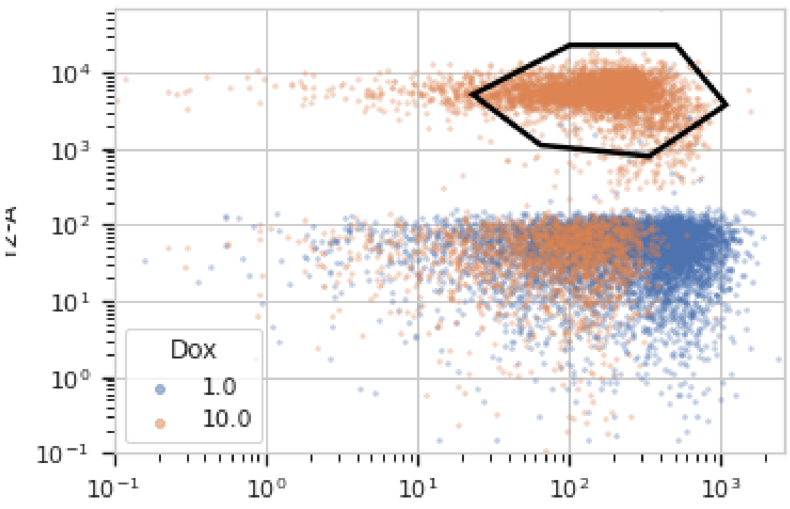


```
>>> ex2 = p.apply(ex)
>>> ex2.data.groupby('Polygon').size()
Polygon
False    15875
True      4125
dtype: int64
```

You can also get (or draw) the polygon on a density plot instead of a scatterplot:

```
>>> df = p.default_view(huefacet = "Dox",
...                         xscale = 'log',
...                         yscale = 'log')
```

```
>>> df.plot(ex)
```



apply(*experiment*)
Applies the threshold to an experiment.

Parameters **experiment** (*Experiment*) – the old `Experiment` to which this op is applied

Returns a new 'Experiment`, the same as `experiment` but with a new column of type `bool` with the same as the operation name. The bool is `True` if the event's measurement is within the polygon, and `False` otherwise.

Return type *Experiment*

Raises `CytoflowOpError` – if for some reason the operation can't be applied to this experiment. The reason is in the `args` attribute.

default_view(***kwargs*)
Returns an `IView` that allows a user to view the polygon or interactively draw it.

Parameters **density** (*bool, default = False*) – If `True`, return a density plot instead of a scatterplot.

class cytoflow.operations.polygon.**ScatterplotPolygonSelectionView**
Bases: `cytoflow.operations.polygon._PolygonSelection`, `cytoflow.views.scatterplot.ScatterplotView`

Plots, and lets the user interact with, a 2D polygon selection on a scatterplot.

interactive
is this view interactive? Ie, can the user set the polygon verticies with mouse clicks?

Type bool

xchannel
The channels to use for this view's X axis. If you created the view using `default_view`, this is already set.

Type Str

ychannel
The channels to use for this view's Y axis. If you created the view using `default_view`, this is already set.

Type Str

xscale
The way to scale the x axis. If you created the view using `default_view`, this may be already set.

Type {'linear', 'log', 'logicle'}

yscale
The way to scale the y axis. If you created the view using `default_view`, this may be already set.

Type {'linear', 'log', 'logicle'}

op
The `IOperation` that this view is associated with. If you created the view using `default_view`, this is already set.

Type Instance(`IOperation`)

subset
An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

xfacet
Set to one of the *Experiment.conditions* in the *Experiment*, and a new column of subplots will be added for every unique value of that condition.

Type String

yfacet
Set to one of the *Experiment.conditions* in the *Experiment*, and a new row of subplots will be added for every unique value of that condition.

Type String

huefacet
Set to one of the *Experiment.conditions* in the in the *Experiment*, and a new color will be added to the plot for every unique value of that condition.

Type String

huescale
How should the color scale for *huefacet* be scaled?

Type {'linear', 'log', 'logicle'}

Examples

In a Jupyter notebook with `%matplotlib notebook`

```
>>> s = flow.PolygonOp(xchannel = "V2-A",
...                    ychannel = "Y2-A")
>>> poly = s.default_view()
>>> poly.plot(ex2)
>>> poly.interactive = True
```

plot(*experiment*, ***kwargs*)
Plot the default view, and then draw the selection on top of it.

Parameters

- **experiment** (*Experiment*) – The *Experiment* to plot using this view.
- **title** (*str*) – Set the plot title
- **xlabel** (*str*) – Set the X axis label
- **ylabel** (*str*) – Set the Y axis label
- **huelabel** (*str*) – Set the label for the hue facet (in the legend)
- **legend** (*bool*) – Plot a legend for the color or hue facet? Defaults to `True`.
- **sharex** (*bool*) – If there are multiple subplots, should they share X axes? Defaults to `True`.
- **sharey** (*bool*) – If there are multiple subplots, should they share Y axes? Defaults to `True`.
- **row_order** (*list*) – Override the row facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **col_order** (*list*) – Override the column facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **hue_order** (*list*) – Override the hue facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.

- **height** (*float*) – The height of *each row* in inches. Default = 3.0
- **aspect** (*float*) – The aspect ratio *of each subplot*. Default = 1.5
- **col_wrap** (*int*) – If `xfacet` is set and `yfacet` is not set, you can "wrap" the subplots around so that they form a multi-row grid by setting this to the number of columns you want.
- **sns_style** (*{"darkgrid", "whitegrid", "dark", "white", "ticks"}*) – Which `seaborn` style to apply to the plot? Default is `whitegrid`.
- **sns_context** (*{"paper", "notebook", "talk", "poster"}*) – Which `seaborn` context to use? Controls the scaling of plot elements such as tick labels and the legend. Default is `talk`.
- **palette** (*palette name, list, or dict*) – Colors to use for the different levels of the hue variable. Should be something that can be interpreted by `seaborn.color_palette`, or a dictionary mapping hue levels to matplotlib colors.
- **despine** (*Bool*) – Remove the top and right axes from the plot? Default is `True`.
- **min_quantile** (*float (>0.0 and <1.0, default = 0.001)*) – Clip data that is less than this quantile.
- **max_quantile** (*float (>0.0 and <1.0, default = 1.00)*) – Clip data that is greater than this quantile.
- **xlim** (*(float, float)*) – Set the range of the plot's X axis.
- **ylim** (*(float, float)*) – Set the range of the plot's Y axis.
- **alpha** (*float (default = 0.25)*) – The alpha blending value, between 0 (transparent) and 1 (opaque).
- **s** (*int (default = 2)*) – The size in points^2.
- **marker** (*a matplotlib marker style, usually a string*) – Specfies the glyph to draw for each point on the scatterplot. See matplotlib.markers for examples. Default: 'o'
- **patch_props** (*Dict*) – The properties of the `matplotlib.patches.Patch` that are drawn on top of the scatterplot or density view. They're passed directly to the `matplotlib.patches.Patch` constructor. Default: `{edgecolor : 'black', linewidth : 2, fill : False}`

`class cytoflow.operations.polygon.`**`DensityPolygonSelectionView`**

Bases: `cytoflow.operations.polygon._PolygonSelection`, *`cytoflow.views.densityplot.DensityView`*

Plots, and lets the user interact with, a 2D polygon selection on a density plot.

`interactive`

is this view interactive? Ie, can the user set the polygon verticies with mouse clicks?

Type bool

`xchannel`

The channels to use for this view's X axis. If you created the view using *`default_view`*, this is already set.

Type Str

`ychannel`

The channels to use for this view's Y axis. If you created the view using *`default_view`*, this is already set.

Type Str

xscale
The way to scale the x axis. If you created the view using *default_view*, this may be already set.

Type {'linear', 'log', 'logicle'}

yscale
The way to scale the y axis. If you created the view using *default_view*, this may be already set.

Type {'linear', 'log', 'logicle'}

op
The *IOperation* that this view is associated with. If you created the view using *default_view*, this is already set.

Type Instance(*IOperation*)

huefacet
You must leave the hue facet unset!

Type None

subset
An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

xfacet
Set to one of the *Experiment.conditions* in the *Experiment*, and a new column of subplots will be added for every unique value of that condition.

Type String

yfacet
Set to one of the *Experiment.conditions* in the *Experiment*, and a new row of subplots will be added for every unique value of that condition.

Type String

huescale
How should the color scale for *huefacet* be scaled?

Type {'linear', 'log', 'logicle'}

Examples

In a Jupyter notebook with `%matplotlib notebook`

```
>>> s = flow.PolygonOp(xchannel = "V2-A",
...                    ychannel = "Y2-A")
>>> poly = s.default_view(density = True)
>>> poly.plot(ex2)
>>> poly.interactive = True
```

plot(*experiment*, ***kwargs*)
Plot the default view, and then draw the selection on top of it.

Parameters **patch_props** (*Dict*) – The properties of the `matplotlib.patches.Patch` that are drawn on top of the scatterplot or density view. They're passed directly to the `matplotlib.patches.Patch` constructor. Default: {edgecolor : 'black', linewidth : 2, fill : False}

cytoflow.operations.quad

Applies a (2D) quad gate to an *Experiment*. *quad* has two classes:

QuadOp – Applies the gate, given a pair of thresholds

ScatterplotQuadSelectionView – an *IView* that allows you to view the quadrants and/or interactively set the thresholds on a scatterplot.

ScatterplotQuadSelectionView – an *IView* that allows you to view the quadrants and/or interactively set the thresholds on a density plot.

class `cytoflow.operations.quad.`**`QuadOp`**

Bases: `traits.has_traits.HasStrictTraits`

Apply a quadrant gate to a cytometry experiment.

Creates a new metadata column named *name*, with values `name_1` (upper-left quadrant), `name_2` (upper-right), `name_3` (lower-left), and `name_4` (lower-right). This ordering is arbitrary, and was chosen to match the FACSDiva order.

`name`

The operation name. Used to name the new metadata field in the experiment that's created by *apply*

Type Str

`xchannel`

The name of the first channel to apply the range gate.

Type Str

`xthreshold`

The threshold in the xchannel to gate with.

Type Float

`ychannel`

The name of the secon channel to apply the range gate.

Type Str

`ythreshold`

The threshold in ychannel to gate with.

Type Float

Examples

Make a little data set.

```
>>> import cytoflow as flow
>>> import_op = flow.ImportOp()
>>> import_op.tubes = [flow.Tube(file = "Plate01/RFP_Well_A3.fcs",
...                               conditions = {'Dox' : 10.0}),
...                     flow.Tube(file = "Plate01/CFP_Well_A4.fcs",
...                               conditions = {'Dox' : 1.0})]
>>> import_op.conditions = {'Dox' : 'float'}
>>> ex = import_op.apply()
```

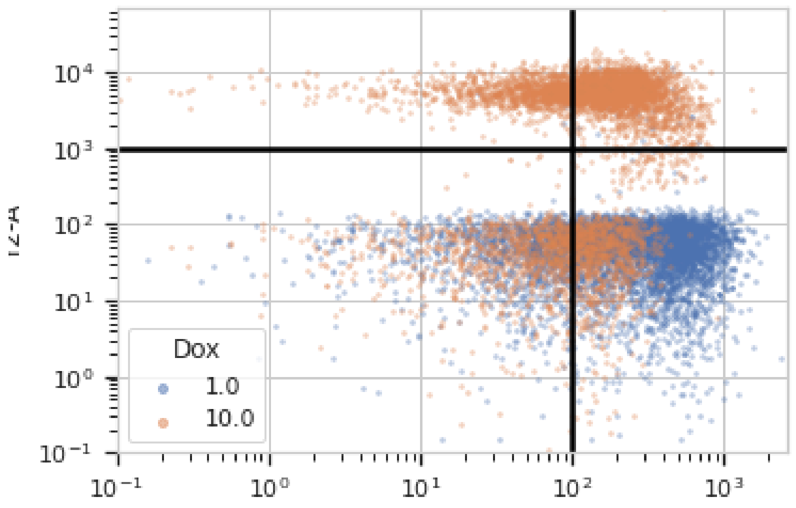
Create and parameterize the operation.

```
>>> quad = flow.QuadOp(name = "Quad",
...                    xchannel = "V2-A",
...                    xthreshold = 100,
...                    ychannel = "Y2-A",
...                    ythreshold = 1000)
```

Show the default view

```
>>> qv = quad.default_view(huefacet = "Dox",
...                        xscale = 'log',
...                        yscale = 'log')
...
```

```
>>> qv.plot(ex)
```



Note: If you want to use the interactive default view in a Jupyter notebook, make sure you say `%matplotlib notebook` in the first cell (instead of `%matplotlib inline` or similar). Then call `default_view()` with `interactive = True`:

```
qv = quad.default_view(huefacet = "Dox",
                       xscale = 'log',
                       yscale = 'log',
                       interactive = True)
qv.plot(ex)
```

Apply the gate and show the result

```
>>> ex2 = quad.apply(ex)
>>> ex2.data.groupby('Quad').size()
Quad
Quad_1    1783
Quad_2    2584
```

(continues on next page)

(continued from previous page)

```
Quad_3    8236
Quad_4    7397
dtype: int64
```

apply(*experiment*)

Applies the quad gate to an experiment.

Parameters **experiment** (`Experiment`) – the `Experiment` to which this op is applied

Returns a new `Experiment`, the same as the old `Experiment` but with a new column the same as the operation `name`. The new column is of type *Category*, with values `name_1`, `name_2`, `name_3`, and `name_4`, applied to events CLOCKWISE from upper-left.

Return type *Experiment*

Raises `CytoflowOpError` – if for some reason the operation can't be applied to this experiment. The reason is in the `args` attribute of `CytoflowOpError`.

default_view(***kwargs*)

Returns an `IView` that allows a user to view the quad selector or interactively draw it.

Parameters **density** (*bool, default = False*) – If `True`, return a density plot instead of a scatterplot.

class cytoflow.operations.quad.**ScatterplotQuadSelectionView**

Bases: `cytoflow.operations.quad._QuadSelection`, `cytoflow.views.scatterplot.ScatterplotView`

Plots, and lets the user interact with, a quadrant gate.

interactive

is this view interactive? Ie, can the user set the threshold with a mouse click?

Type Bool

xchannel

The channels to use for this view's X axis. If you created the view using `default_view`, this is already set.

Type Str

ychannel

The channels to use for this view's Y axis. If you created the view using `default_view`, this is already set.

Type Str

xscale

The way to scale the x axis. If you created the view using `default_view`, this may be already set.

Type {'linear', 'log', 'logicle'}

yscale

The way to scale the y axis. If you created the view using `default_view`, this may be already set.

Type {'linear', 'log', 'logicle'}

op

The `IOperation` that this view is associated with. If you created the view using `default_view`, this is already set.

Type Instance(`IOperation`)

subset
An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

xfacet
Set to one of the *Experiment.conditions* in the *Experiment*, and a new column of subplots will be added for every unique value of that condition.

Type String

yfacet
Set to one of the *Experiment.conditions* in the *Experiment*, and a new row of subplots will be added for every unique value of that condition.

Type String

huefacet
Set to one of the *Experiment.conditions* in the in the *Experiment*, and a new color will be added to the plot for every unique value of that condition.

Type String

huescale
How should the color scale for *huefacet* be scaled?

Type {'linear', 'log', 'logicle'}

Examples

In an Jupyter notebook with `%matplotlib notebook`

```
>>> q = flow.QuadOp(name = "Quad",
...                 xchannel = "V2-A",
...                 ychannel = "Y2-A"))
>>> qv = q.default_view()
>>> qv.interactive = True
>>> qv.plot(ex2)
```

plot(*experiment*, ***kwargs*)
Plot the default view, and then draw the quad selection on top of it.

Parameters

- **experiment** (*Experiment*) – The *Experiment* to plot using this view.
- **title** (*str*) – Set the plot title
- **xlabel** (*str*) – Set the X axis label
- **ylabel** (*str*) – Set the Y axis label
- **huelabel** (*str*) – Set the label for the hue facet (in the legend)
- **legend** (*bool*) – Plot a legend for the color or hue facet? Defaults to `True`.
- **sharex** (*bool*) – If there are multiple subplots, should they share X axes? Defaults to `True`.
- **sharey** (*bool*) – If there are multiple subplots, should they share Y axes? Defaults to `True`.
- **row_order** (*list*) – Override the row facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.

- **col_order** (*list*) – Override the column facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **hue_order** (*list*) – Override the hue facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **height** (*float*) – The height of *each row* in inches. Default = 3.0
- **aspect** (*float*) – The aspect ratio *of each subplot*. Default = 1.5
- **col_wrap** (*int*) – If `xfacet` is set and `yfacet` is not set, you can "wrap" the subplots around so that they form a multi-row grid by setting this to the number of columns you want.
- **sns_style** (*{"darkgrid", "whitegrid", "dark", "white", "ticks"}*) – Which `seaborn` style to apply to the plot? Default is `whitegrid`.
- **sns_context** (*{"paper", "notebook", "talk", "poster"}*) – Which `seaborn` context to use? Controls the scaling of plot elements such as tick labels and the legend. Default is `talk`.
- **palette** (*palette name, list, or dict*) – Colors to use for the different levels of the hue variable. Should be something that can be interpreted by `seaborn.color_palette`, or a dictionary mapping hue levels to matplotlib colors.
- **despine** (*Bool*) – Remove the top and right axes from the plot? Default is `True`.
- **min_quantile** (*float (>0.0 and <1.0, default = 0.001)*) – Clip data that is less than this quantile.
- **max_quantile** (*float (>0.0 and <1.0, default = 1.00)*) – Clip data that is greater than this quantile.
- **xlim** (*(float, float)*) – Set the range of the plot's X axis.
- **ylim** (*(float, float)*) – Set the range of the plot's Y axis.
- **alpha** (*float (default = 0.25)*) – The alpha blending value, between 0 (transparent) and 1 (opaque).
- **s** (*int (default = 2)*) – The size in points^2.
- **marker** (*a matplotlib marker style, usually a string*) – Specfies the glyph to draw for each point on the scatterplot. See matplotlib.markers for examples. Default: 'o'
- **line_props** (*Dict*) – The properties of the `matplotlib.lines.Line2D` that are drawn on top of the scatterplot or density view. They're passed directly to the `matplotlib.lines.Line2D` constructor. Default: `{color :  'black', linewidth :  2}`

`class cytoflow.operations.quad.`**`DensityQuadSelectionView`**

Bases: `cytoflow.operations.quad._QuadSelection`, *`cytoflow.views.densityplot.DensityView`*

Plots, and lets the user interact with, a quadrant gate on a density view

`interactive`

is this view interactive? Ie, can the user set the threshold with a mouse click?

Type Bool

`xchannel`

The channels to use for this view's X axis. If you created the view using *`default_view`*, this is already set.

Type Str

ychannel
The channels to use for this view's Y axis. If you created the view using *default_view*, this is already set.

Type Str

xscale
The way to scale the x axis. If you created the view using *default_view*, this may be already set.

Type {'linear', 'log', 'logicle'}

yscale
The way to scale the y axis. If you created the view using *default_view*, this may be already set.

Type {'linear', 'log', 'logicle'}

op
The *IOperation* that this view is associated with. If you created the view using *default_view*, this is already set.

Type Instance(*IOperation*)

huefacet
You must leave the hue facet unset!

Type None

subset
An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

xfacet
Set to one of the *Experiment.conditions* in the *Experiment*, and a new column of subplots will be added for every unique value of that condition.

Type String

yfacet
Set to one of the *Experiment.conditions* in the *Experiment*, and a new row of subplots will be added for every unique value of that condition.

Type String

huescale
How should the color scale for *huefacet* be scaled?

Type {'linear', 'log', 'logicle'}

Examples

In an Jupyter notebook with `%matplotlib notebook`

```
>>> q = flow.QuadOp(name = "Quad",
...                 xchannel = "V2-A",
...                 ychannel = "Y2-A"))
>>> qv = q.default_view(density = True)
>>> qv.interactive = True
>>> qv.plot(ex2)
```

`plot`(*experiment*, ***kwargs*)

Plot the default view, and then draw the quad selection on top of it.

Parameters

- **experiment** (*Experiment*) – The `Experiment` to plot using this view.
- **title** (*str*) – Set the plot title
- **xlabel** (*str*) – Set the X axis label
- **ylabel** (*str*) – Set the Y axis label
- **huelabel** (*str*) – Set the label for the hue facet (in the legend)
- **legend** (*bool*) – Plot a legend for the color or hue facet? Defaults to `True`.
- **sharex** (*bool*) – If there are multiple subplots, should they share X axes? Defaults to `True`.
- **sharey** (*bool*) – If there are multiple subplots, should they share Y axes? Defaults to `True`.
- **row_order** (*list*) – Override the row facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **col_order** (*list*) – Override the column facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **hue_order** (*list*) – Override the hue facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **height** (*float*) – The height of *each row* in inches. Default = 3.0
- **aspect** (*float*) – The aspect ratio *of each subplot*. Default = 1.5
- **col_wrap** (*int*) – If `xfacet` is set and `yfacet` is not set, you can "wrap" the subplots around so that they form a multi-row grid by setting this to the number of columns you want.
- **sns_style** (*{"darkgrid", "whitegrid", "dark", "white", "ticks"}*) – Which `seaborn` style to apply to the plot? Default is `whitegrid`.
- **sns_context** (*{"paper", "notebook", "talk", "poster"}*) – Which `seaborn` context to use? Controls the scaling of plot elements such as tick labels and the legend. Default is `talk`.
- **palette** (*palette name, list, or dict*) – Colors to use for the different levels of the hue variable. Should be something that can be interpreted by `seaborn.color_palette`, or a dictionary mapping hue levels to matplotlib colors.
- **despine** (*Bool*) – Remove the top and right axes from the plot? Default is `True`.
- **min_quantile** (*float (>0.0 and <1.0, default = 0.001)*) – Clip data that is less than this quantile.
- **max_quantile** (*float (>0.0 and <1.0, default = 1.00)*) – Clip data that is greater than this quantile.
- **xlim** (*(float, float)*) – Set the range of the plot's X axis.
- **ylim** (*(float, float)*) – Set the range of the plot's Y axis.
- **gridsize** (*int*) – The size of the grid on each axis. Default = 50
- **smoothed** (*bool*) – Should the resulting mesh be smoothed?
- **smoothed_sigma** (*int*) – The standard deviation of the smoothing kernel. default = 1.

- **cmap** (*cmap*) – An instance of matplotlib.colors.Colormap. By default, the `viridis` colormap is used
- **under_color** (*matplotlib color*) – Sets the color to be used for low out-of-range values.
- **bad_color** (*matplotlib color*) – Set the color to be used for masked values.
- **line_props** (*Dict*) – The properties of the `matplotlib.lines.Line2D` that are drawn on top of the scatterplot or density view. They're passed directly to the `matplotlib.lines.Line2D` constructor. Default: `{color :  'black', linewidth :  2}`

cytoflow.operations.range

Applies a (1D) range gate to an *`Experiment`*. *`range`* has two classes:

`RangeOp` – Applies the gate, given a pair of thresholds

`RangeSelection` – an *`IView`* that allows you to view the range and/or interactively set the thresholds.

`class cytoflow.operations.range.`**`RangeOp`**

Bases: `traits.has_traits.HasStrictTraits`

Apply a range gate to a cytometry experiment.

`name`

The operation name. Used to name the new metadata field in the experiment that's created by *`apply`*

Type Str

`channel`

The name of the channel to apply the range gate.

Type Str

`low`

The lowest value to include in this gate.

Type Float

`high`

The highest value to include in this gate.

Type Float

Examples

Make a little data set.

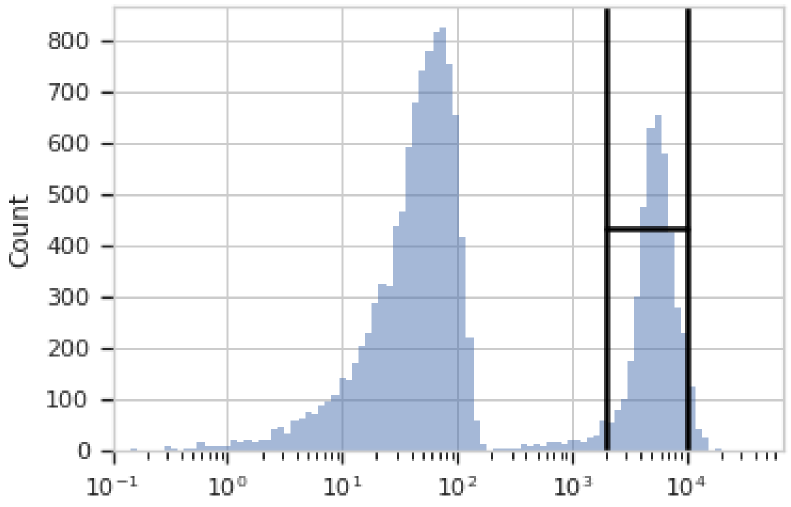
```
>>> import cytoflow as flow
>>> import_op = flow.ImportOp()
>>> import_op.tubes = [flow.Tube(file = "Plate01/RFP_Well_A3.fcs",
...                               conditions = {'Dox' : 10.0}),
...                     flow.Tube(file = "Plate01/CFP_Well_A4.fcs",
...                               conditions = {'Dox' : 1.0})]
>>> import_op.conditions = {'Dox' : 'float'}
>>> ex = import_op.apply()
```

Create and parameterize the operation.

```
>>> range_op = flow.RangeOp(name = 'Range',
...                         channel = 'Y2-A',
...                         low = 2000,
...                         high = 10000)
```

Plot a diagnostic view

```
>>> rv = range_op.default_view(scale = 'log')
>>> rv.plot(ex)
```



Note: If you want to use the interactive default view in a Jupyter notebook, make sure you say `%matplotlib notebook` in the first cell (instead of `%matplotlib inline` or similar). Then call `default_view()` with `interactive = True`:

```
rv = range_op.default_view(scale = 'log',
                           interactive = True)
rv.plot(ex)
```

Apply the gate, and show the result

```
>>> ex2 = range_op.apply(ex)
>>> ex2.data.groupby('Range').size()
Range
False    16042
True      3958
dtype: int64
```

apply(*experiment*)

Applies the range gate to an experiment.

Parameters **experiment** (*Experiment*) – the *Experiment* to which this op is applied

Returns a new *Experiment*, the same as old *Experiment* but with a new column of type `bool` with the same as the operation name. The bool is `True` if the event's measurement in *channel* is greater than `low` and less than `high`; it is `False` otherwise.

Return type *Experiment*

`default_view`(***kwargs*)

class `cytoflow.operations.range.RangeSelection`

Bases: `cytoflow.operations.base_op_views.Op1DView`, `cytoflow.views.histogram.HistogramView`

Plots, and lets the user interact with, a selection on the X axis.

`interactive`

is this view interactive? Ie, can the user set min and max with a mouse drag?

Type Bool

`channel`

The channel this view is viewing. If you created the view using `default_view`, this is already set.

Type Str

`scale`

The way to scale the x axes. If you created the view using `default_view`, this may be already set.

Type {'linear', 'log', 'logicle'}

`op`

The `IOperation` that this view is associated with. If you created the view using `default_view`, this is already set.

Type Instance(`IOperation`)

`subset`

An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

`xfacet`

Set to one of the `Experiment.conditions` in the `Experiment`, and a new column of subplots will be added for every unique value of that condition.

Type String

`yfacet`

Set to one of the `Experiment.conditions` in the `Experiment`, and a new row of subplots will be added for every unique value of that condition.

Type String

`huefacet`

Set to one of the `Experiment.conditions` in the in the `Experiment`, and a new color will be added to the plot for every unique value of that condition.

Type String

`huescale`

How should the color scale for `huefacet` be scaled?

Type {'linear', 'log', 'logicle'}

Examples

In an IPython notebook with `%matplotlib notebook`

```
>>> r = RangeOp(name = "RangeGate",
...             channel = 'Y2-A')
>>> rv = r.default_view()
>>> rv.interactive = True
>>> rv.plot(ex2)
>>> ### draw a range on the plot ###
>>> print r.low, r.high
```

`plot`(*experiment*, ***kwargs*)

Plot the underlying histogram and then plot the selection on top of it.

Parameters

- **experiment** (*Experiment*) – The `Experiment` to plot using this view.
- **title** (*str*) – Set the plot title
- **xlabel** (*str*) – Set the X axis label
- **ylabel** (*str*) – Set the Y axis label
- **huelabel** (*str*) – Set the label for the hue facet (in the legend)
- **legend** (*bool*) – Plot a legend for the color or hue facet? Defaults to `True`.
- **sharex** (*bool*) – If there are multiple subplots, should they share X axes? Defaults to `True`.
- **sharey** (*bool*) – If there are multiple subplots, should they share Y axes? Defaults to `True`.
- **row_order** (*list*) – Override the row facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **col_order** (*list*) – Override the column facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **hue_order** (*list*) – Override the hue facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **height** (*float*) – The height of *each row* in inches. Default = 3.0
- **aspect** (*float*) – The aspect ratio *of each subplot*. Default = 1.5
- **col_wrap** (*int*) – If `xfacet` is set and `yfacet` is not set, you can "wrap" the subplots around so that they form a multi-row grid by setting this to the number of columns you want.
- **sns_style** (*{"darkgrid", "whitegrid", "dark", "white", "ticks"}*) – Which `seaborn` style to apply to the plot? Default is `whitegrid`.
- **sns_context** (*{"paper", "notebook", "talk", "poster"}*) – Which `seaborn` context to use? Controls the scaling of plot elements such as tick labels and the legend. Default is `talk`.
- **palette** (*palette name, list, or dict*) – Colors to use for the different levels of the hue variable. Should be something that can be interpreted by `seaborn.color_palette`, or a dictionary mapping hue levels to matplotlib colors.
- **despine** (*Bool*) – Remove the top and right axes from the plot? Default is `True`.
- **min_quantile** (*float (>0.0 and <1.0, default = 0.001)*) – Clip data that is less than this quantile.

- **max_quantile** (*float (>0.0 and <1.0, default = 1.00)*) – Clip data that is greater than this quantile.
- **lim** (*(float, float)*) – Set the range of the plot's data axis.
- **orientation** (*{'vertical', 'horizontal'}*)
- **num_bins** (*int*) – The number of bins to plot in the histogram. Clipped to [100, 1000]
- **histtype** (*{'stepfilled', 'step', 'bar'}*) – The type of histogram to draw. `stepfilled` is the default, which is a line plot with a color filled under the curve.
- **density** (*bool*) – If `True`, re-scale the histogram to form a probability density function, so the area under the histogram is 1.
- **linewidth** (*float*) – The width of the histogram line (in points)
- **linestyle** (*['-' | '–' | '-.' | ':' | "None"]*) – The style of the line to plot
- **alpha** (*float (default = 0.5)*) – The alpha blending value, between 0 (transparent) and 1 (opaque).
- **line_props** (*Dict*) – The properties of the `matplotlib.lines.Line2D` that are drawn on top of the histogram. They're passed directly to the `matplotlib.lines.Line2D` constructor. Default: `{color : 'black', linewidth : 2}`

cytoflow.operations.range2d

Applies a 2D range gate (ie, a rectangle gate) to an *`Experiment`*. *`range2d`* has two classes:

`Range2DOp` – Applies the gate, given four thresholds

`ScatterplotRangeSelection2DView` – an *`IView`* that allows you to view the range and/or interactively set the thresholds on a scatterplot.

`DensityRangeSelection2DView` – an *`IView`* that allows you to view the range and/or interactively set the thresholds on a scatterplot.

`class cytoflow.operations.range2d.`**`Range2DOp`**

Bases: `traits.has_traits.HasStrictTraits`

Apply a 2D range gate to a cytometry experiment.

`name`

The operation name. Used to name the new metadata field in the experiment that's created by *`apply`*

Type Str

`xchannel`

The name of the first channel to apply the range gate.

Type Str

`xlow`

The lowest value in xchannel to include in this gate.

Type Float

`xhigh`

The highest value in xchannel to include in this gate.

Type Float

`ychannel`
The name of the second channel to apply the range gate.

Type Str

`ylow`
The lowest value in ychannel to include in this gate.

Type Float

`yhigh`
The highest value in ychannel to include in this gate.

Type Float

Examples

Make a little data set.

```
>>> import cytoflow as flow
>>> import_op = flow.ImportOp()
>>> import_op.tubes = [flow.Tube(file = "Plate01/RFP_Well_A3.fcs",
...                               conditions = {'Dox' : 10.0}),
...                     flow.Tube(file = "Plate01/CFP_Well_A4.fcs",
...                               conditions = {'Dox' : 1.0})]
>>> import_op.conditions = {'Dox' : 'float'}
>>> ex = import_op.apply()
```

Create and parameterize the operation.

```
>>> r = flow.Range2DOp(name = "Range2D",
...                     xchannel = "V2-A",
...                     xlow = 10,
...                     xhigh = 1000,
...                     ychannel = "Y2-A",
...                     ylow = 1000,
...                     yhigh = 20000)
```

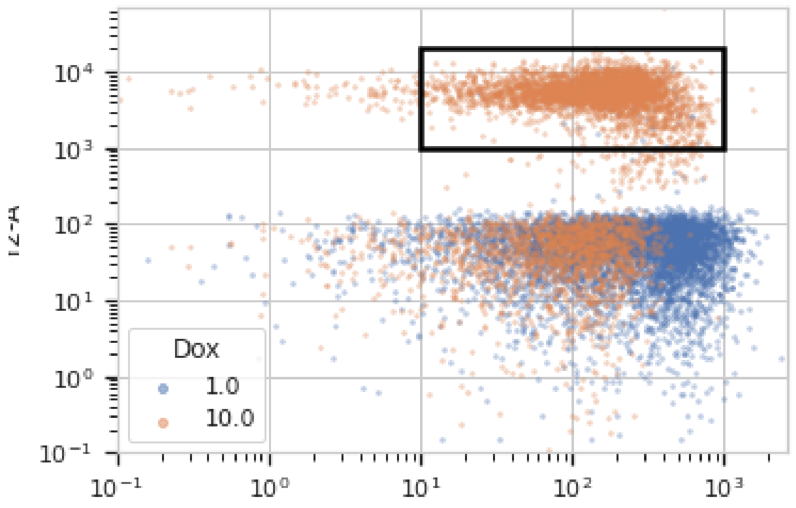
Show the default view.

```
>>> rv = r.default_view(huefacet = "Dox",
...                      xscale = 'log',
...                      yscale = 'log')
```

```
>>> rv.plot(ex)
```

Note: If you want to use the interactive default view in a Jupyter notebook, make sure you say `%matplotlib notebook` in the first cell (instead of `%matplotlib inline` or similar). Then call `default_view()` with `interactive = True`:

```
rv = r.default_view(huefacet = "Dox",
                    xscale = 'log',
                    yscale = 'log',
                    interactive = True)
rv.plot(ex)
```



Apply the gate, and show the result

```
>>> ex2 = r.apply(ex)
>>> ex2.data.groupby('Range2D').size()
Range2D
False    16405
True      3595
dtype: int64
```

apply(*experiment*)

Applies the threshold to an experiment.

Parameters **experiment** (*Experiment*) – the Experiment to which this op is applied

Returns a new Experiment, the same as the old experiment but with a new column with a data type of bool and the same as the operation name. The bool is True if the event's measurement in xchannel is greater than xlow and less than xhigh, and the event's measurement in ychannel is greater than ylow and less than yhigh; it is False otherwise.

Return type *Experiment*

default_view(***kwargs*)

Returns an IView that allows a user to view the selection or interactively draw it.

Parameters **density** (*bool, default = False*) – If True, return a density plot instead of a scatterplot.

class cytoflow.operations.range2d.**ScatterplotRangeSelection2DView**

Bases: cytoflow.operations.range2d._RangeSelection2D, *cytoflow.views.scatterplot.ScatterplotView*

Plots, and lets the user interact with, a 2D selection.

interactive

is this view interactive? Ie, can the user set min and max with a mouse drag?

Type Bool

`xchannel`
The channels to use for this view's X axis. If you created the view using `default_view`, this is already set.

Type Str

`ychannel`
The channels to use for this view's Y axis. If you created the view using `default_view`, this is already set.

Type Str

`xscale`
The way to scale the x axis. If you created the view using `default_view`, this may be already set.

Type {'linear', 'log', 'logicle'}

`yscale`
The way to scale the y axis. If you created the view using `default_view`, this may be already set.

Type {'linear', 'log', 'logicle'}

`op`
The `IOperation` that this view is associated with. If you created the view using `default_view`, this is already set.

Type Instance(`IOperation`)

`subset`
An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

`xfacet`
Set to one of the `Experiment.conditions` in the `Experiment`, and a new column of subplots will be added for every unique value of that condition.

Type String

`yfacet`
Set to one of the `Experiment.conditions` in the `Experiment`, and a new row of subplots will be added for every unique value of that condition.

Type String

`huefacet`
Set to one of the `Experiment.conditions` in the in the `Experiment`, and a new color will be added to the plot for every unique value of that condition.

Type String

`huescale`
How should the color scale for `huefacet` be scaled?

Type {'linear', 'log', 'logicle'}

Examples

In a Jupyter notebook with `%matplotlib notebook`

```
>>> r = flow.Range2DOp(name = "Range2D",
...                    xchannel = "V2-A",
...                    ychannel = "Y2-A"))
>>> rv = r.default_view()
>>> rv.interactive = True
>>> rv.plot(ex2)
```

class cytoflow.operations.range2d.**DensityRangeSelection2DView**

Bases: `cytoflow.operations.range2d._RangeSelection2D`, *`cytoflow.views.densityplot.DensityView`*

Plots, and lets the user interact with, a 2D selection.

interactive

is this view interactive? Ie, can the user set min and max with a mouse drag?

Type Bool

xchannel

The channels to use for this view's X axis. If you created the view using *`default_view`*, this is already set.

Type Str

ychannel

The channels to use for this view's Y axis. If you created the view using *`default_view`*, this is already set.

Type Str

xscale

The way to scale the x axis. If you created the view using *`default_view`*, this may be already set.

Type {'linear', 'log', 'logicle'}

yscale

The way to scale the y axis. If you created the view using *`default_view`*, this may be already set.

Type {'linear', 'log', 'logicle'}

op

The *`IOperation`* that this view is associated with. If you created the view using *`default_view`*, this is already set.

Type Instance(*`IOperation`*)

huefacet

You must leave the hue facet unset!

Type None

subset

An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

xfacet
Set to one of the *Experiment.conditions* in the *Experiment*, and a new column of subplots will be added for every unique value of that condition.

Type String

yfacet
Set to one of the *Experiment.conditions* in the *Experiment*, and a new row of subplots will be added for every unique value of that condition.

Type String

huescale
How should the color scale for *huefacet* be scaled?

Type {'linear', 'log', 'logicle'}

Examples

In a Jupyter notebook with `%matplotlib notebook`

```
>>> r = flow.Range2DOp(name = "Range2D",
...                    xchannel = "V2-A",
...                    ychannel = "Y2-A"))
>>> rv = r.default_view(density = True)
>>> rv.interactive = True
>>> rv.plot(ex2)
```

cytoflow.operations.ratio

Creates a new "channel" that is the ratio of the measurements in two other channels. *ratio* has one class:

RatioOp – applies the operation.

class `cytoflow.operations.ratio.`**`RatioOp`**
Bases: `traits.has_traits.HasStrictTraits`

Create a new "channel" from the ratio of two other channels.

name
The operation name. Also becomes the name of the new channel.

Type Str

numerator
The channel that is the numerator of the ratio.

Type Str

denominator
The channel that is the denominator of the ratio.

Type Str

Examples

```
>>> ratio_op = flow.RatioOp()
>>> ratio_op.numerator = "FITC-A"
>>> ex5 = ratio_op.apply(ex4)
```

`apply`(*experiment*)

Applies the ratio operation to an experiment

Parameters **experiment** (*Experiment*) – the *Experiment* to which this operation is applied

Returns

a new experiment with the new ratio channel

The new channel also has the following new metadata:

- **numerator** [Str] What was the numerator channel for the new one?
- **denominator** [Str] What was the denominator channel for the new one?

Return type *Experiment*

cytoflow.operations.threshold

Applies a threshold gate to an *Experiment*. *threshold* has two classes:

ThresholdOp – Applies the gate, given a threshold

ThresholdSelection – an *IView* that allows you to view and/or interactively set the threshold.

`class cytoflow.operations.threshold.ThresholdOp`

Bases: `traits.has_traits.HasStrictTraits`

Apply a threshold gate to a cytometry experiment.

`name`

The operation name. Used to name the new column in the experiment that's created by *apply*

Type Str

`channel`

The name of the channel to apply the threshold on.

Type Str

`threshold`

The value at which to threshold this channel.

Type Float

Examples

Make a little data set.

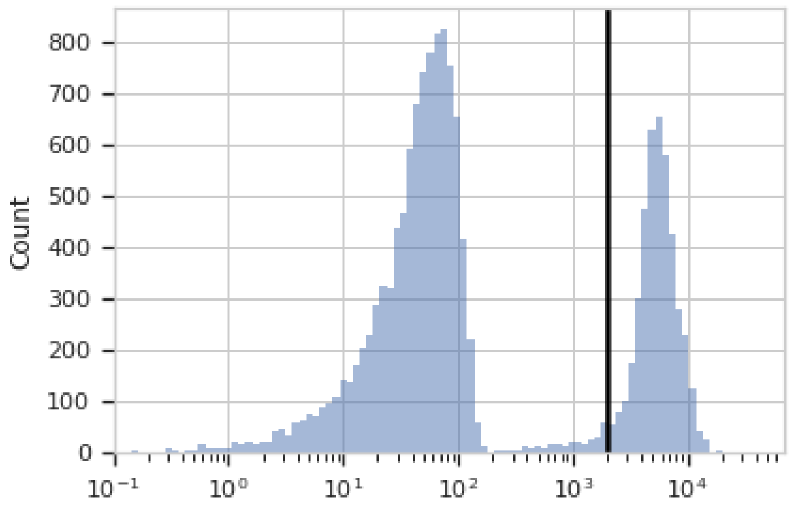
```
>>> import cytoflow as flow
>>> import_op = flow.ImportOp()
>>> import_op.tubes = [flow.Tube(file = "Plate01/RFP_Well_A3.fcs",
...                              conditions = {'Dox' : 10.0}),
...                     flow.Tube(file = "Plate01/CFP_Well_A4.fcs",
...                              conditions = {'Dox' : 1.0})]
>>> import_op.conditions = {'Dox' : 'float'}
>>> ex = import_op.apply()
```

Create and parameterize the operation.

```
>>> thresh_op = flow.ThresholdOp(name = 'Threshold',
...                              channel = 'Y2-A',
...                              threshold = 2000)
```

Plot a diagnostic view

```
>>> tv = thresh_op.default_view(scale = 'log')
>>> tv.plot(ex)
```



Note: If you want to use the interactive default view in a Jupyter notebook, make sure you say `%matplotlib notebook` in the first cell (instead of `%matplotlib inline` or similar). Then call `default_view()` with `interactive = True`:

```
tv = thresh_op.default_view(scale = 'log',
                            interactive = True)
tv.plot(ex)
```

Apply the gate, and show the result

```
>>> ex2 = thresh_op.apply(ex)
>>> ex2.data.groupby('Threshold').size()
Threshold
False    15786
True      4214
dtype: int64
```

apply(*experiment*)

Applies the threshold to an experiment.

Parameters **experiment** (*Experiment*) – the `Experiment` to which this operation is applied

Returns a new `Experiment`, the same as the old experiment but with a new column of type `bool` with the same name as the operation `name`. The new condition is `True` if the event's measurement in `channel` is greater than `threshold`; it is `False` otherwise.

Return type *Experiment*

default_view(***kwargs*)

class cytoflow.operations.threshold.**ThresholdSelection**

Bases: `cytoflow.operations.base_op_views.Op1DView`, `cytoflow.views.histogram.HistogramView`

Plots, and lets the user interact with, a threshold on the X axis.

interactive

is this view interactive?

Type Bool

channel

The channel this view is viewing. If you created the view using `default_view`, this is already set.

Type Str

scale

The way to scale the x axes. If you created the view using `default_view`, this may be already set.

Type {'linear', 'log', 'logicle'}

op

The `IOperation` that this view is associated with. If you created the view using `default_view`, this is already set.

Type Instance(`IOperation`)

subset

An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

xfacet

Set to one of the `Experiment.conditions` in the `Experiment`, and a new column of subplots will be added for every unique value of that condition.

Type String

yfacet

Set to one of the `Experiment.conditions` in the `Experiment`, and a new row of subplots will be added for every unique value of that condition.

Type String

`huefacet`
Set to one of the *`Experiment.conditions`* in the in the *`Experiment`*, and a new color will be added to the plot for every unique value of that condition.

Type String

`huescale`
How should the color scale for *`huefacet`* be scaled?

Type {'linear', 'log', 'logicle'}

Examples

In an Jupyter notebook with `%matplotlib notebook`

```
>>> t = flow.ThresholdOp(name = "Threshold",
...                      channel = "Y2-A")
>>> tv = t.default_view()
>>> tv.plot(ex2)
>>> tv.interactive = True
>>> # .... draw a threshold on the plot
>>> ex3 = thresh.apply(ex2)
```

`plot`(*experiment*, ***kwargs*)
Plot the histogram and then plot the threshold on top of it.

Parameters

- **experiment** (*Experiment*) – The *`Experiment`* to plot using this view.
- **title** (*str*) – Set the plot title
- **xlabel** (*str*) – Set the X axis label
- **ylabel** (*str*) – Set the Y axis label
- **huelabel** (*str*) – Set the label for the hue facet (in the legend)
- **legend** (*bool*) – Plot a legend for the color or hue facet? Defaults to `True`.
- **sharex** (*bool*) – If there are multiple subplots, should they share X axes? Defaults to `True`.
- **sharey** (*bool*) – If there are multiple subplots, should they share Y axes? Defaults to `True`.
- **row_order** (*list*) – Override the row facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **col_order** (*list*) – Override the column facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **hue_order** (*list*) – Override the hue facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **height** (*float*) – The height of *each row* in inches. Default = 3.0
- **aspect** (*float*) – The aspect ratio *of each subplot*. Default = 1.5
- **col_wrap** (*int*) – If *`xfacet`* is set and *`yfacet`* is not set, you can "wrap" the subplots around so that they form a multi-row grid by setting this to the number of columns you want.

- **sns_style** (*{"darkgrid", "whitegrid", "dark", "white", "ticks"}*) – Which `seaborn` style to apply to the plot? Default is `whitegrid`.
- **sns_context** (*{"paper", "notebook", "talk", "poster"}*) – Which `seaborn` context to use? Controls the scaling of plot elements such as tick labels and the legend. Default is `talk`.
- **palette** (*palette name, list, or dict*) – Colors to use for the different levels of the hue variable. Should be something that can be interpreted by `seaborn.color_palette`, or a dictionary mapping hue levels to matplotlib colors.
- **despine** (*Bool*) – Remove the top and right axes from the plot? Default is `True`.
- **min_quantile** (*float (>0.0 and <1.0, default = 0.001)*) – Clip data that is less than this quantile.
- **max_quantile** (*float (>0.0 and <1.0, default = 1.00)*) – Clip data that is greater than this quantile.
- **lim** (*(float, float)*) – Set the range of the plot's data axis.
- **orientation** (*{'vertical', 'horizontal'}*)
- **num_bins** (*int*) – The number of bins to plot in the histogram. Clipped to [100, 1000]
- **histtype** (*{'stepfilled', 'step', 'bar'}*) – The type of histogram to draw. `stepfilled` is the default, which is a line plot with a color filled under the curve.
- **density** (*bool*) – If `True`, re-scale the histogram to form a probability density function, so the area under the histogram is 1.
- **linewidth** (*float*) – The width of the histogram line (in points)
- **linestyle** (*['-' | '–' | '-.' | ':' | "None"]*) – The style of the line to plot
- **alpha** (*float (default = 0.5)*) – The alpha blending value, between 0 (transparent) and 1 (opaque).
- **line_props** (*Dict*) – The properties of the `matplotlib.lines.Line2D` that are drawn on top of the histogram. They're passed directly to the `matplotlib.lines.Line2D` constructor. Default: `{color : 'black', linewidth : 2}`

cytoflow.operations.xform_stat

Transforms a statistic. *`xform_stat`* has one class:

`TransformStatisticOp` – apply a function to a statistic, making a new statistic.

class `cytoflow.operations.xform_stat.`**`TransformStatisticOp`**

Bases: `traits.has_traits.HasStrictTraits`

Apply a function to a statistic, creating a new statistic. The function can be applied to the entire statistic, or it can be applied individually to groups of the statistic. The function should take a `pandas.Series` as its only argument. Return type is arbitrary, but a to be used with the rest of *`cytoflow`* it should probably be a numeric type or an iterable of numeric types.

As a special case, if the function returns a `pandas.Series` *with the same index that it was passed*, it is interpreted as a transformation. The resulting statistic will have the same length, index names and index levels as the original statistic.

`name`

The operation name. Becomes the first element in the *`Experiment.statistics`* key tuple.

Type Str

statistic
The statistic to apply the function to.

Type Tuple(Str, Str)

function
The function used to transform the statistic. *function* must take a pandas.Series as its only parameter. The return type is arbitrary, but to work with the rest of *cytoflow* it should probably be a numeric type or an iterable of numeric types.. If *statistic_name* is unset, the name of the function becomes the second in element in the *Experiment.statistics* key tuple.

Type Callable

statistic_name
The name of the function; if present, becomes the second element in the *Experiment.statistics* key tuple.

Type Str

by
A list of metadata attributes to aggregate the input statistic before applying the function. For example, if the statistic has two indices `Time` and `Dox`, setting `by = ["Time", "Dox"]` will apply *function* separately to each subset of the data with a unique combination of `Time` and `Dox`.

Type List(Str)

fill
Value to use in the statistic if a slice of the data is empty.

Type Any (default = 0)

Examples

```
>>> stats_op = ChannelStatisticOp(name = "Mean",
...                               channel = "Y2-A",
...                               function = np.mean,
...                               by = ["Dox"])
>>> ex2 = stats_op.apply(ex)
>>> log_op = TransformStatisticOp(name = "LogMean",
...                               statistic = ("Mean", "mean"),
...                               function = np.log)
>>> ex3 = log_op.apply(ex2)
```

apply(*experiment*)
Applies *function* to a statistic.

Parameters experiment (*Experiment*) – The *Experiment* to apply the operation to

Returns The same as the old experiment, but with a new statistic that results from applying *function* to the statistic specified in *statistic*.

Return type *Experiment*

cytoflow.scripts package

Submodules

cytoflow.scripts.channel_voltages

Returns the channel voltages ($PnV) for the given FCS file.

`cytoflow.scripts.channel_voltages.main()`

cytoflow.scripts.fcs_metadata

Return the FCS metadata (in key : value format) for a given FCS file.

`cytoflow.scripts.fcs_metadata.main()`

cytoflow.scripts.parse_beads

Parse beads.csv into a dict to go in cytoflow.operations.bead_calibration. (As new beads are added, please add them to cytoflow/operations/beads.csv, then run this and put the result in *cytoflow.operations.bead_calibration*)

`cytoflow.scripts.parse_beads.main()`

cytoflow.scripts.split_fcs

The FCS standard allows multiple datasets to be concatenated into one FCS file. This script splits them, optionally renaming the resulting files using the provided FCS metadata.

`cytoflow.scripts.split_fcs.main()`

cytoflow.utility package

cytoflow.utility

This package contains a bunch of utility functions to support the other modules in *cytoflow*. This includes numeric functions, algorithms, error handling, custom traits, custom `matplotlib` widgets, custom `matplotlib` scales, and docstring handling.

Submodules

cytoflow.utility.algorithms

Useful algorithms.

ci – determine a confidence interval by boostrapping.

percentiles – find percentiles in an array.

bootstrap – resample (with replacement) and store aggregate values.

`cytoflow.utility.algorithms.`**`ci`**`(`*data*, *func*, *which=95*, *boots=1000*`)`

Determine the confidence interval of a function applied to a data set by bootstrapping.

Parameters

- **data** (*pandas.DataFrame*) – The data to resample.
- **func** (*callable*) – A function that is called on a resampled `data`
- **which** (*int*) – The percentile to use for the confidence interval
- **boots** (*int (default = 1000):*) – How many times to bootstrap

Returns The confidence interval.

Return type (float, float)

`cytoflow.utility.algorithms.`**`percentiles`**`(`*a*, *pcts*, *axis=None*`)`

Like `scipy.stats.scoreatpercentile` but can take and return array of percentiles.

from seaborn: https://github.com/mwaskom/seaborn/blob/master/seaborn/utils.py

Parameters

- **a** (*array*) – data
- **pcts** (*sequence of percentile values*) – percentile or percentiles to find score at
- **axis** (*int or None*) – if not None, computes scores over this axis

Returns **scores** – array of scores at requested percentiles first dimension is length of object passed to `pcts`

Return type array

`cytoflow.utility.algorithms.`**`bootstrap`**`(`**args*, ***kwargs*`)`

Resample one or more arrays with replacement and store aggregate values. Positional arguments are a sequence of arrays to bootstrap along the first axis and pass to a summary function.

Parameters

- **n_boot** (*int, default 10000*) – Number of iterations
- **axis** (*int, default None*) – Will pass axis to `func` as a keyword argument.
- **units** (*array, default None*) – Array of sampling unit IDs. When used the bootstrap resamples units and then observations within units instead of individual datapoints.
- **smooth** (*bool, default False*) – If True, performs a smoothed bootstrap (draws samples from a kernel destiny estimate); only works for one-dimensional inputs and cannot be used *`units`* is present.
- **func** (*callable, default np.mean*) – Function to call on the args that are passed in.
- **random_seed** (*int | None, default None*) – Seed for the random number generator; useful if you want reproducible resamples.

Returns

- *array* – array of bootstrapped statistic values
- **from seaborn** (*https://github.com/mwaskom/seaborn/blob/master/seaborn/algorithms.py*)

`cytoflow.utility.algorithms.`**`is_inside_sm`**`(`*polygon*, *point*`)`

`cytoflow.utility.algorithms.`**`polygon_contains`**`(`*points*, *polygon*`)`

cytoflow.utility.custom_traits

Custom traits for `cytoflow`

`PositiveInt`, `PositiveFloat` – versions of `traits.trait_types.Int` and `traits.trait_types.Float` that must be positive (and optionally 0).

`PositiveCInt`, `PositiveCFloat` – versions of `traits.trait_types.CInt`, `traits.trait_types.CFloat` that must be positive (and optionally 0).

`IntOrNone`, `FloatOrNone` – versions of `traits.trait_types.Int` and `traits.trait_types.Float` that may also hold the value `None`.

`CIntOrNone`, `CFloatOrNone` – versions of `traits.trait_types.CInt` and `traits.trait_types.CFloat` that may also hold the value `None`.

`ScaleEnum` – an enumeration whose value is one of the registered scales.

`Removed` – a trait that was present in a previous version but was removed.

`Deprecated` – a trait that was present in a previous version but was renamed.

class `cytoflow.utility.custom_traits.`**`PositiveInt`**(*default_value=<traits.trait_type._NoDefaultSpecifiedType object>*, ***metadata*)

Bases: `traits.trait_types.BaseInt`

Defines a trait whose value must be a positive integer

`info_text = 'a positive integer'`

A description of the type of value this trait accepts:

`validate`(*obj*, *name*, *value*)

Validates that a specified value is valid for this trait.

class `cytoflow.utility.custom_traits.`**`PositiveCInt`**(*default_value=<traits.trait_type._NoDefaultSpecifiedType object>*, ***metadata*)

Bases: `traits.trait_types.BaseCInt`

Defines a trait whose converted value must be a positive integer

`info_text = 'a positive integer'`

A description of the type of value this trait accepts:

`validate`(*obj*, *name*, *value*)

Validates that a specified value is valid for this trait.

Note: The ‘fast validator’ version performs this check in C.

class `cytoflow.utility.custom_traits.`**`PositiveFloat`**(*default_value=<traits.trait_type._NoDefaultSpecifiedType object>*, ***metadata*)

Bases: `traits.trait_types.BaseFloat`

Defines a trait whose value must be a positive float

`info_text = 'a positive float'`

A description of the type of value this trait accepts:

`validate`(*obj*, *name*, *value*)

Validates that a specified value is valid for this trait.

Note: The ‘fast validator’ version performs this check in C.

class `cytoflow.utility.custom_traits.`**`PositiveCFloat`**(*default_value=<traits.trait_type._NoDefaultSpecifiedType object>*, ***metadata*)

Bases: `traits.trait_types.BaseCFloat`

Defines a trait whose converted value must be a positive float

info_text = 'a positive float'
A description of the type of value this trait accepts:

validate(*obj*, *name*, *value*)
Validates that a specified value is valid for this trait.

Note: The 'fast validator' version performs this check in C.

class cytoflow.utility.custom_traits.**FloatOrNone**(*default_value=<traits.trait_type._NoDefaultSpecifiedType object>*, ***metadata*)
Bases: `traits.trait_types.BaseFloat`

Defines a trait whose value must be a float or None

info_text = 'a float or None'
A description of the type of value this trait accepts:

validate(*obj*, *name*, *value*)
Validates that a specified value is valid for this trait.

Note: The 'fast validator' version performs this check in C.

class cytoflow.utility.custom_traits.**CFloatOrNone**(*default_value=<traits.trait_type._NoDefaultSpecifiedType object>*, ***metadata*)
Bases: `traits.trait_types.BaseCFloat`

Defines a trait whose converted value must be a float or None

info_text = 'a float or None'
A description of the type of value this trait accepts:

validate(*obj*, *name*, *value*)
Validates that a specified value is valid for this trait.

Note: The 'fast validator' version performs this check in C.

class cytoflow.utility.custom_traits.**IntOrNone**(*default_value=<traits.trait_type._NoDefaultSpecifiedType object>*, ***metadata*)
Bases: `traits.trait_types.BaseInt`

Defines a trait whose value must be an integer or None

info_text = 'an int or None'
A description of the type of value this trait accepts:

validate(*obj*, *name*, *value*)
Validates that a specified value is valid for this trait.

class cytoflow.utility.custom_traits.**CIntOrNone**(*default_value=<traits.trait_type._NoDefaultSpecifiedType object>*, ***metadata*)
Bases: `traits.trait_types.BaseCInt`

Defines a trait whose converted value must be an integer or None

info_text = 'an int or None'
A description of the type of value this trait accepts:

validate(*obj*, *name*, *value*)
Validates that a specified value is valid for this trait.

Note: The 'fast validator' version performs this check in C.

class `cytoflow.utility.custom_traits.`**`ScaleEnum`**`(`**args*, ***metadata*`)`

Bases: `traits.trait_types.BaseEnum`

Defines an enumeration that contains one of the registered data scales

`info_text = 'an enum containing one of the registered scales'`

`get_default_value``()`

Get information about the default value.

The default implementation analyzes the value of the trait's `default_value` attribute and determines an appropriate `default_value_type` for the `default_value`. If you need to override this method to provide a different result tuple, the following values are valid values for `default_value_type`:

- 0, 1: The `default_value` item of the tuple is the default value.
- 2: The object containing the trait is the default value.
- 3: A new copy of the list specified by `default_value` is the default value.
- 4: A new copy of the dictionary specified by `default_value` is the default value.
- 5: A new instance of TraitListObject constructed using the `default_value` list is the default value.
- 6: A new instance of TraitDictObject constructed using the `default_value` dictionary is the default value.
- 7: `default_value` is a tuple of the form: `(callable, args, kw)`, where `callable` is a callable, `args` is a tuple, and `kw` is either a dictionary or None. The default value is the result obtained by invoking `callable(\*args, \*\*kw)`.
- 8: `default_value` is a callable. The default value is the result obtained by invoking `default_value(object)`, where `object` is the object containing the trait. If the trait has a `validate()` method, the `validate()` method is also called to validate the result.
- 9: A new instance of `TraitSetObject` constructed using the `default_value` set is the default value.

Returns **default_value_type, default_value** – The default value information, consisting of an integer, giving the type of default value, and the corresponding default value as described above.

Return type int, any

class `cytoflow.utility.custom_traits.`**`Removed`**`(`***metadata*`)`

Bases: `traits.trait_type.TraitType`

Names a trait that was present in a previous version but was removed.

Trait metadata:

- **err_string** : the error string in the error
- **gui** : if `True`, don't return a backtrace (because it's very slow)
- **warning** [if `True`, raise a warning when the trait is referenced.] Otherwise, raise an exception.

`get``(`*obj*, *name*`)`

`set``(`*obj*, *name*, *value*`)`

class `cytoflow.utility.custom_traits.`**`Deprecated`**`(`***metadata*`)`

Bases: `traits.trait_type.TraitType`

Names a trait that was present in a previous version but was renamed. When the trait is accessed, a warning is raised, and the access is passed through to the new trait.

Trait metadata:

- **new** : the name of the new trait
- **err_string** : the error string in the error
- **gui** : if `True`, don't return a backtrace (because it's very slow)

get(*obj*, *name*)

set(*obj*, *name*, *value*)

cytoflow.utility.cytoflow_errors

Custom errors for *cytoflow*. Allows for custom handling in the GUI.

CytoflowError – a general error

CytoflowOpError – an error raised by an operation

CytoflowViewError – an error raised by a view

CytoflowWarning – a general warning

CytoflowOpWarning – a warning raised by an operation

CytoflowViewWarning – a warning raised by a view

exception `cytoflow.utility.cytoflow_errors.`**CytoflowError**

Bases: `RuntimeError`

A general error

exception `cytoflow.utility.cytoflow_errors.`**CytoflowOpError**

Bases: *cytoflow.utility.cytoflow_errors.CytoflowError*

An error raised by an operation.

Parameters

- **args[0]** (*string*) – The attribute or parameter whose bad value caused the error, or `None` if there isn't one.
- **args[1]** (*string*) – A more verbose error message.

exception `cytoflow.utility.cytoflow_errors.`**CytoflowViewError**

Bases: *cytoflow.utility.cytoflow_errors.CytoflowError*

An error raised by a view.

Parameters

- **args[0]** (*string*) – The attribute or parameter whose bad value caused the error, or `None` if there isn't one.
- **args[1]** (*string*) – A more verbose error message.

exception `cytoflow.utility.cytoflow_errors.`**CytoflowWarning**

Bases: `UserWarning`

A general warning.

exception `cytoflow.utility.cytoflow_errors.`**CytoflowOpWarning**

Bases: *cytoflow.utility.cytoflow_errors.CytoflowWarning*

A warning raised by an operation.

Parameters **args[0]** (*string*) – A verbose warning message

exception `cytoflow.utility.cytoflow_errors.`**`CytoflowViewWarning`**

Bases: *`cytoflow.utility.cytoflow_errors.CytoflowWarning`*

A warning raised by a view.

Parameters **args[0]** (*string*) – A verbose warning message

cytoflow.utility.docstring

Utility functions for operating on docstrings. They all assume that docstrings are formatted using the NumPy standard style:

https://numpydoc.readthedocs.io/en/latest/format.html

`expand_class_attributes` – take entries in the `Attributes` section of a class's ancestors' docstrings and adds them to the `Attributes` section of this class's docstring.

`expand_method_parameters` – expand the `Parameters` section of a method's docstring with `Parameters` from the overridden methods in the class's ancestors.

`find_section` – find a named section in a numpy-formatted docstring.

`get_class_attributes` – get the entries from the `Attributes` section of a class's docstring

`get_method_parameters` – get the entries from the `Parameters` section of a method's docstring

`cytoflow.utility.docstring.`**`expand_class_attributes`**(*cls*)

Takes entries in the `Attributes` section of a class's ancestors' docstrings and adds them to the `Attributes` section of this class's docstring.

All the classes must have docstrings formatted with the using the `numpy` docstring style.

Parameters **cls** (*class*) – The class whose docstring is to be expanded.

`cytoflow.utility.docstring.`**`expand_method_parameters`**(*cls*, *method*)

Expand the `Parameters` section of a method's docstring with `Parameters` from the overridden methods in the class's ancestors.

All the methods must have docstrings formatted with the using the `numpy` docstring style.

Parameters

- **cls** (*class*) – The class whose ancestors are to be parsed for more parameters.
- **method** (*callable*) – The method whose docstring is to be expanded.

`cytoflow.utility.docstring.`**`find_section`**(*section*, *lines*)

Find a named section in a `numpy`-formatted docstring.

Parameters

- **section** (*string*) – The name of the section to find
- **lines** (*array of string*) – The docstring, split into lines

Returns The indices of the first and last lines of the section.

Return type int, int

`cytoflow.utility.docstring.`**`get_class_attributes`**(*cls*)

Gets the entries from the `Attributes` section of a class's `numpy`-formated docstring.

Parameters **cls** (*class*) – The class whose docstring to parse

Returns

- name : the attribute's name
- type : the attribute's type
- body : the attribute's description body

Return type array of `(name, type, body)`

`cytoflow.utility.docstring.`**`get_method_parameters`**`(`*method*`)`

Gets the entries from the `Parameters` section of a method's `numpy`-formatted docstring.

Parameters **method** (*callable*) – The method whose docstring to parse.

Returns

- name : the attribute's name
- type : the attribute's type
- body : the attribute's description body

Return type array of `(name, type, body)`

cytoflow.utility.fcswrite

Write .fcs files for flow cytometry

Adapted from https://github.com/ZELLMECHANIK-DRESDEN/fcswrite

`cytoflow.utility.fcswrite.`**`write_fcs`**`(`*filename*, *chn_names*, *chn_ranges*, *data*, *compat_chn_names=True*, *compat_percent=True*, *compat_negative=True*, *compat_copy=True*, *verbose=0*, ***kws*`)`

Write numpy data to an .fcs file (FCS3.0 file format)

Parameters

- **filename** (*str*) – Path to the output .fcs file
- **chn_names** (*list of str, length C*) – Names of the output channels
- **chn_ranges** (*dictionary*) – Keys: channel names. Values: ranges
- **data** (*2d ndarray of shape (N,C)*) – The numpy array data to store as .fcs file format.
- **compat_chn_names** (*bool*) – Compatibility mode for 3rd party flow analysis software: The characters " ", "?", and "_" are removed in the output channel names.
- **compat_percent** (*bool*) – Compatibliity mode for 3rd party flow analysis software: If a column in `data` contains values only between 0 and 1, they are multiplied by 100.
- **compat_negative** (*bool*) – Compatibliity mode for 3rd party flow analysis software: Flip the sign of `data` if its mean is smaller than zero.
- **compat_copy** (*bool*) – Do not override the input array `data` when modified in compatibility mode.
- **kwargs** (*Str*) – Additional keyword arguments are written as keyword/value pairs in the TEXT segment of the FCS file.

Notes

These commonly used unicode characters are replaced: “µ”, “^{2}”

cytoflow.utility.hlog_scale

A scale that transforms the data using the `hyperlog` function. *`hlog_scale`* has several classes:

`HlogScale` – implements *`IScale`*, the *`cytoflow`* interface for the scale.

`MatplotlibHlogScale` – inherits `matplotlib.scale.ScaleBase`, implements the matplotlib interface

`HlogMajorLocator` – inherits `matplotlib.ticker.Locator`, lets matplotlib know where major tics are along a plot axis.

`HlogMinorLocator` – inherits `matplotlib.ticker.Locator`, lets matplotlib know where minor tics are along a plot axis

`hlog`, *`hlog_inv`* – the actual functions that perform the scale and inverse

class `cytoflow.utility.hlog_scale.`**`HlogScale`**(***kwargs*)

Bases: *`cytoflow.utility.scale.ScaleMixin`*

A scale that transforms the data using the `hyperlog` function.

This scaling method implements a “linear-like” region around 0, and a “log-like” region for large values, with a smooth transition between them.

The transformation has one parameter, *`b`*, which specifies the location of the transition from linear to log-like. The default, `500`, is good for 18-bit scales and not good for other scales.

`b`

the location of the transition from linear to log-like.

Type Float (default = 500)

References

[1] Hyperlog-a flexible log-like transform for negative, zero, and positive valued data. Bagwell CB. Cytometry A. 2005 Mar;64(1):34-42. PMID: 15700280 http://onlinelibrary.wiley.com/doi/10.1002/cyto.a.20114/abstract

`name = 'hlog'`

`inverse`(*data*)

Transforms ‘data’ using the inverse of this scale.

`clip`(*data*)

Clips data to the range of the scale function

`norm`()

A factory function that returns `matplotlib.colors.Normalize` instance, which normalizes values for a `matplotlib` color palette.

`get_mpl_params`()

Returns a dict with the traits needed to initialize an instance of *`MatplotlibHlogScale`*

class `cytoflow.utility.hlog_scale.`**`MatplotlibHlogScale`**(*axis*, ***kwargs*)

Bases: `traits.has_traits.HasTraits`, `matplotlib.scale.ScaleBase`

A class that inherits from `matplotlib.scale.ScaleBase`, which implements all the bits for `matplotlib` to use a new scale.

`name = 'hlog'`

`get_transform`()
Returns the matplotlib.transform instance that does the actual transformation

`set_default_locators_and_formatters`(*axis*)
Set the locators and formatters to reasonable defaults for linear scaling.

`class` **`HlogTransform`**(***kwargs*)
Bases: `traits.has_traits.HasTraits`, `matplotlib.transforms.Transform`

A class that implements the actual transformation

`input_dims = 1`
The number of input dimensions of this transform. Must be overridden (with integers) in the subclass.

`output_dims = 1`
The number of output dimensions of this transform. Must be overridden (with integers) in the subclass.

`is_separable = True`
True if this transform is separable in the x- and y- dimensions.

`has_inverse = True`
True if this transform has a corresponding inverse transform.

`transform_non_affine`(*values*)
Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Parameters **values** (*array*) – The input values as NumPy array of length *`input_dims`* or shape (N x *`input_dims`*).

Returns The output values as NumPy array of length *`input_dims`* or shape (N x *`output_dims`*), depending on the input.

Return type array

`inverted`()
Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

`class` **`InvertedLogicleTransform`**(***kwargs*)
Bases: `traits.has_traits.HasTraits`, `matplotlib.transforms.Transform`

A class that implements the inverse transformation

`input_dims = 1`
The number of input dimensions of this transform. Must be overridden (with integers) in the subclass.

`output_dims = 1`
The number of output dimensions of this transform. Must be overridden (with integers) in the subclass.

`is_separable = True`
True if this transform is separable in the x- and y- dimensions.

has_inverse = True
True if this transform has a corresponding inverse transform.

transform_non_affine(*values*)
Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Parameters **values** (*array*) – The input values as NumPy array of length *input_dims* or shape (N x *input_dims*).

Returns The output values as NumPy array of length *input_dims* or shape (N x *output_dims*), depending on the input.

Return type array

inverted()
Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

class cytoflow.utility.hlog_scale.**HlogMajorLocator**
Bases: `matplotlib.ticker.Locator`

Determine the tick locations for hlog axes. Based on matplotlib.LogLocator

set_params()
Empty

tick_values(*vmin*, *vmax*)
Every decade, including 0 and negative

view_limits(*data_min*, *data_max*)
Try to choose the view limits intelligently

class cytoflow.utility.hlog_scale.**HlogMinorLocator**
Bases: `matplotlib.ticker.Locator`

Determine the tick locations for logicle axes. Based on matplotlib.LogLocator

set_params()
Empty

tick_values(*vmin*, *vmax*)
Every tenth decade, including 0 and negative

cytoflow.utility.hlog_scale.**hlog_inv**(*y*, *b*, *r*, *d*)
Inverse of base 10 hyperlog transform.

cytoflow.utility.hlog_scale.**hlog**(*x*, *b*, *r*, *d*)
Base 10 hyperlog transform.

Parameters

- **x** (*num | num iterable*) – values to be transformed.
- **b** (*num*) – Parameter controling the location of the shift from linear to log transformation.
- **r** (*num (default = 10**4)*) – maximal transformed value.
- **d** (*num (default = log10(2**18))*) – log10 of maximal possible measured value. hlog_inv(r) = 10**d

Return type Array of transformed values.

cytoflow.utility.linear_scale

A scale that doesn't transform data at all – a "default" scale.

`LinearScale` – implements a no-op `IScale`

class `cytoflow.utility.linear_scale.`**`LinearScale`**`(**kwargs)`

Bases: `cytoflow.utility.scale.ScaleMixin`

A scale that doesn't transform the data at all.

`name = 'linear'`

`inverse``(data)`

`clip``(data)`

`norm``(vmin=None, vmax=None)`

`get_mpl_params``(ax)`

cytoflow.utility.log_scale

A scale that transforms data using a base-10 log.

`LogScale` – implements `IScale`, the `cytoflow` interface for the scale.

class `cytoflow.utility.log_scale.`**`LogScale`**`(**kwargs)`

Bases: `cytoflow.utility.scale.ScaleMixin`

`name = 'log'`

`get_mpl_params``(ax)`

Returns a dict with the traits needed to initialize an instance of `matplotlib.scale.ScaleBase`

`inverse``(data)`

Transforms 'data' using the inverse of this scale.

`clip``(data)`

Clips data to the range of the scale function

`norm``(vmin=None, vmax=None)`

A factory function that returns `matplotlib.colors.Normalize` instance, which normalizes values for a `matplotlib` color palette.

cytoflow.utility.logging

Utilities to help with logging.

`MplFilter` – a `logging.Filter` that removes nuisance warnings

class `cytoflow.utility.logging.`**`MplFilter`**`(name='')`

Bases: `logging.Filter`

A `logging.Filter` that removes nuisance warnings

filter(*record*)

Determine if the specified record is to be logged.

Returns True if the record should be logged, or False otherwise. If deemed appropriate, the record may be modified in-place.

cytoflow.utility.logicle_scale

A scale that transforms the data using the `logicle` function.

`LogicleScale` – implements *`IScale`*, the *`cytoflow`* interface for the scale.

`MatplotlibLogicleScale` – inherits `matplotlib.scale.ScaleBase`, implements the matplotlib interface

`LogicleMajorLocator` – inherits `matplotlib.ticker.Locator`, lets matplotlib know where major tics are along a plot axis.

`LogicleMinorLocator` – inherits `matplotlib.ticker.Locator`, lets matplotlib know where minor tics are along a plot axis

class `cytoflow.utility.logicle_scale.`**`LogicleScale`**

Bases: `traits.has_traits.HasStrictTraits`

A scale that transforms the data using the `logicle` function.

This scaling method implements a "linear-like" region around 0, and a "log-like" region for large values, with a very smooth transition between them. It's particularly good for compensated data, and data where you have "negative" events (events with a fluorescence of ~0.)

If you don't have any data around 0, you might be better of with a more traditional log scale.

The transformation has one parameter, *`W`*, which specifies the width of the "linear" range in log10 decades. By default, the optimal value is estimated from the data; but if you assign a value to *`W`* it will be used. `0.5` is usually a good start.

`experiment`

the *`Experiment`* used to estimate the scale parameters.

Type Instance(cytoflow.Experiment)

`channel`

If set, choose scale parameters from this channel in *`experiment`*. One of *`channel`*, *`condition`* or *`statistic`* must be set.

Type Str

`condition`

If set, choose scale parameters from this condition in *`experiment`*. One of *`channel`*, *`condition`* or *`statistic`* must be set.

Type Str

`statistic`

If set, choose scale parameters from this statistic in *`experiment`*. One of *`channel`*, *`condition`* or *`statistic`* must be set.

Type Str

`quantiles = Tuple(Float, Float) (default = (0.001, 0.999))`

If there are a few very large or very small values, this can throw off matplotlib's choice of default axis ranges. Set `quantiles` to choose what part of the data to consider when choosing axis ranges.

W

The width of the linear range, in log10 decades. can estimate from data, or use a fixed value like 0.5.

Type Float (default = estimated from data)

M

The width of the log portion of the display, in log10 decades.

Type Float (default = 4.5)

A

additional decades of negative data to include. the default display usually captures all the data, so 0 is fine to start.

Type Float (default = 0.0)

r

Quantile used to estimate *W*.

Type Float (default = 0.05)

References

[1] A new "Logicle" display method avoids deceptive effects of logarithmic scaling for low signals and compensated data. Parks DR, Roederer M, Moore WA. Cytometry A. 2006 Jun;69(6):541-51. PMID: 16604519 http://onlinelibrary.wiley.com/doi/10.1002/cyto.a.20258/full

[2] Update for the logicle data scale including operational code implementations. Moore WA, Parks DR. Cytometry A. 2012 Apr;81(4):273-7. doi: 10.1002/cyto.a.22030 PMID: 22411901 http://onlinelibrary.wiley.com/doi/10.1002/cyto.a.22030/full

name = 'logicle'

inverse(*data*)

Transforms 'data' using the inverse of this scale.

clip(*data*)

Clips data to the range of the scale function

norm(*vmin=None*, *vmax=None*)

A factory function that returns `matplotlib.colors.Normalize` instance, which normalizes values for a `matplotlib` color palette.

get_mpl_params(*ax*)

class cytoflow.utility.logicle_scale.**MatplotlibLogicleScale**(*axis*, ***kwargs*)

Bases: `traits.has_traits.HasTraits`, `matplotlib.scale.ScaleBase`

A class that inherits from `matplotlib.scale.ScaleBase`, which implements all the bits for `matplotlib` to use a new scale.

name = 'logicle'

get_transform()

Returns the matplotlib.transform instance that does the actual transformation

set_default_locators_and_formatters(*axis*)

Set the locators and formatters to reasonable defaults for linear scaling.

class **LogicleTransform**(***kwargs*)

Bases: `traits.has_traits.HasTraits`, `matplotlib.transforms.Transform`

input_dims = 1
The number of input dimensions of this transform. Must be overridden (with integers) in the subclass.

output_dims = 1
The number of output dimensions of this transform. Must be overridden (with integers) in the subclass.

is_separable = True
True if this transform is separable in the x- and y- dimensions.

has_inverse = True
True if this transform has a corresponding inverse transform.

transform_non_affine(*values*)
Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Parameters **values** (*array*) – The input values as NumPy array of length `input_dims` or shape (N x `input_dims`).

Returns The output values as NumPy array of length `input_dims` or shape (N x `output_dims`), depending on the input.

Return type array

inverted()
Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

class **InvertedLogicleTransform**(***kwargs*)
Bases: `traits.has_traits.HasTraits`, `matplotlib.transforms.Transform`

input_dims = 1
The number of input dimensions of this transform. Must be overridden (with integers) in the subclass.

output_dims = 1
The number of output dimensions of this transform. Must be overridden (with integers) in the subclass.

is_separable = True
True if this transform is separable in the x- and y- dimensions.

has_inverse = True
True if this transform has a corresponding inverse transform.

transform_non_affine(*values*)
Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Parameters **values** (*array*) – The input values as NumPy array of length `input_dims` or shape (N x `input_dims`).

Returns The output values as NumPy array of length `input_dims` or shape (N x `output_dims`), depending on the input.

Return type array

inverted()
Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

class `cytoflow.utility.logicle_scale.`**`LogicleMajorLocator`**

Bases: `matplotlib.ticker.Locator`

Determine the tick locations for logicle axes. Based on matplotlib.LogLocator

`set_params`(***kwargs*)
Empty

`tick_values`(*vmin*, *vmax*)
Every decade, including 0 and negative

`view_limits`(*data_min*, *data_max*)
Try to choose the view limits intelligently

class `cytoflow.utility.logicle_scale.`**`LogicleMinorLocator`**

Bases: `matplotlib.ticker.Locator`

Determine the tick locations for logicle axes. Based on matplotlib.LogLocator

`set_params`()
Empty

`tick_values`(*vmin*, *vmax*)
Every tenth decade, including 0 and negative

`view_limits`(*data_min*, *data_max*)
Try to choose the view limits intelligently

cytoflow.utility.scale

Base classes and functions for *`cytoflow`* scales.

`IScale` – the `traits.has_traits.Interface` that scales must implement

`ScaleMixin` – provides useful functionality that scales can inherit

`scale_factory` – make a new instance of a scale

`register_scale` – register a new type of scale

`set_default_scale`, *`get_default_scale`* – sets and gets the default scale

class `cytoflow.utility.scale.`**`IScale`**(*adaptee*, *default=<class 'traits.adaptation.adaptation_error.AdaptationError'>*)

Bases: `traits.has_traits.Interface`

An interface for various ways we could rescale flow data.

`name`
The name of this view (for serialization, UI, etc.)

Type Str

`experiment`
The experiment this scale is to be applied to. Needed because some scales have parameters estimated from data.

Type Instance(*Experiment*)

channel

Which channel to scale. Needed because some scales have parameters estimated from data.

Type Str

condition

What condition to scale. Needed because some scales have parameters estimated from the a condition. Must be a numeric condition; else instantiating the scale should fail.

Type Str

statistic

What statistic to scale. Needed because some scales have parameters estimated from a statistic. The statistic must be numeric or an iterable of numerics; else instantiating the scale should fail.

Type Tuple(Str, Str)

data

What raw data to scale.

Type array_like

inverse(*data*)

Transforms 'data' using the inverse of this scale. Must know how to handle int, float, and list, tuple, numpy.ndarray and pandas.Series of int or float. Returns the same type as passed.

clip(*data*)

Clips the data to the scale's domain.

norm(*vmin=None*, *vmax=None*)

Return an instance of matplotlib.colors.Normalize, which normalizes this scale to [0, 1]. Among other things, this is used to correctly scale a color bar.

class cytoflow.utility.scale.**ScaleMixin**(***kwargs*)

Bases: `traits.has_traits.HasStrictTraits`

Provides useful functionality that scales can inherit.

cytoflow.utility.scale.**scale_factory**(*scale*, *experiment*, ***scale_params*)

Make a new instance of a named scale.

Parameters

- **scale** (*string*) – The name of the scale to build
- **experiment** (*Experiment*) – The experiment to use to parameterize the new scale.
- ****scale_params** (*kwargs*) – Other parameters to pass to the scale constructor

cytoflow.utility.scale.**register_scale**(*scale_class*)

Register a new scale for the `scale_factory` and `ScaleEnum`.

cytoflow.utility.scale.**set_default_scale**(*scale*)

Set a default scale for `ScaleEnum`

cytoflow.utility.scale.**get_default_scale**()

Get the defaults scale set with `set_default_scale`

cytoflow.utility.util_functions

Useful (mostly numeric) utility functions.

`iqr` – calculate the interquartile range for an array of numberes

`num_hist_bins` – calculate the number of histogram bins using Freedman-Diaconis

`geom_mean` – compute the geometric mean

`geom_sd` – compute the geometric standard deviation

`geom_sd_range` – compute [geom_mean / geom_sd, geom_mean * geom_sd]

`geom_sem` – compute the geometric standard error of the mean

`geom_sem_range` – compute [geom_mean / geom_sem, geom_mean * geom_sem]

`cartesian` – generate the cartesian product of input arrays

`sanitize_identifier` – makes a string a valid Python identifier by replacing all non-safe characters with '_'

`random_string` – Makes a random string of ascii digits and lowercase letters

`is_numeric` – determine if a `pandas.Series` or `numpy.ndarray` is numeric from its dtype

`cov2corr` – compute the correlation matrix from the covariance matric

`cytoflow.utility.util_functions.`**`iqr`**`(`*a*`)`

Calculate the inter-quartile range for an array of numbers.

Parameters **a** (*array_like*) – The array of numbers to compute the IQR for.

Returns The IQR of the data.

Return type float

`cytoflow.utility.util_functions.`**`num_hist_bins`**`(`*a*`)`

Calculate number of histogram bins using Freedman-Diaconis rule.

From http://stats.stackexchange.com/questions/798/

Parameters **a** (*array_like*) – The data to make a histogram of.

Returns The number of bins in the histogram

Return type int

`cytoflow.utility.util_functions.`**`geom_mean`**`(`*a*`)`

Compute the geometric mean for an "arbitrary" data set, ie one that contains zeros and negative numbers.

Parameters **a** (*array-like*) – A numpy.ndarray, or something that can be converted to an ndarray

Return type The geometric mean of the input array

Notes

The traditional geometric mean can not be computed on a mixture of positive and negative numbers. The approach here, validated rigorously in the cited paper[1], is to compute the geometric mean of the absolute value of the negative numbers separately, and then take a weighted arithmetic mean of that and the geometric mean of the positive numbers. We're going to discard 0 values, operating under the assumption that in this context there are going to be few or no observations with a value of exactly 0.

References

[1] Geometric mean for negative and zero values Elsayed A. E. Habib International Journal of Research and Reviews in Applied Sciences 11:419 (2012) http://www.arpapress.com/Volumes/Vol11Issue3/IJRRAS_11_3_08.pdf

`cytoflow.utility.util_functions.`**`geom_sd`**`(`*a*`)`

Compute the geometric standard deviation for an "abitrary" data set, ie one that contains zeros and negative numbers. Since we're in log space, this gives a *dimensionless scaling factor*, not a measure. If you want traditional "error bars", don't plot `[geom_mean - geom_sd, geom_mean + geom_sd]`; rather, plot `[geom_mean / geom_sd, geom_mean * geom_sd]`.

Parameters **a** (*array-like*) – A numpy.ndarray, or something that can be converted to an ndarray

Return type The geometric standard deviation of the distribution.

Notes

As with *`geom_mean`*, non-positive numbers pose a problem. The approach here, though less rigorously validated than the one above, is to replace negative numbers with their absolute value plus 2 * geometric mean, then go about our business as per the Wikipedia page for geometric sd[1].

References

[1] https://en.wikipedia.org/wiki/Geometric_standard_deviation

`cytoflow.utility.util_functions.`**`geom_sd_range`**`(`*a*`)`

A convenience function to compute [geom_mean / geom_sd, geom_mean * geom_sd].

Parameters **a** (*array-like*) – A numpy.ndarray, or something that can be converted to an ndarray

Return type A tuple, with `(geom_mean / geom_sd, geom_mean * geom_sd)`

`cytoflow.utility.util_functions.`**`geom_sem`**`(`*a*`)`

Compute the geometric standard error of the mean for an "arbirary" data set, ie one that contains zeros and negative numbers.

Parameters **a** (*array-like*) – A numpy.ndarray, or something that can be converted to an ndarray

Return type The geometric mean of the distribution.

Notes

As with *geom_mean*, non-positive numbers pose a problem. The approach here, though less rigorously validated than the one above, is to replace negative numbers with their absolute value plus 2 * geometric mean. The geometric SEM is computed as in [1].

References

[1] The Standard Errors of the Geometric and Harmonic Means and Their Application to Index Numbers
Nilan Norris The Annals of Mathematical Statistics Vol. 11, No. 4 (Dec., 1940), pp. 445-448

http://www.jstor.org/stable/2235723?seq=1#page_scan_tab_contents

`cytoflow.utility.util_functions.`**`geom_sem_range`**`(`*a*`)`
A convenience function to compute [geom_mean / geom_sem, geom_mean * geom_sem].

Parameters a (*array-like*) – A numpy.ndarray, or something that can be converted to an ndarray

Return type A tuple, with `(geom_mean / geom_sem, geom_mean * geom_sem)`

`cytoflow.utility.util_functions.`**`cartesian`**`(`*arrays*, *out=None*`)`
Generate a cartesian product of input arrays.

Parameters

- **arrays** (*list of array-like*) – 1-D arrays to form the cartesian product of.
- **out** (*ndarray*) – Array to place the cartesian product in.

Returns out – 2-D array of shape (M, len(arrays)) containing cartesian products formed of input arrays.

Return type ndarray

Examples

```
>>> cartesian(([1, 2, 3], [4, 5], [6, 7]))
array([[1, 4, 6],
       [1, 4, 7],
       [1, 5, 6],
       [1, 5, 7],
       [2, 4, 6],
       [2, 4, 7],
       [2, 5, 6],
       [2, 5, 7],
       [3, 4, 6],
       [3, 4, 7],
       [3, 5, 6],
       [3, 5, 7]])
```

References

Originally from http://stackoverflow.com/a/1235363/4755587

`cytoflow.utility.util_functions.`**`sanitize_identifier`**`(`*name*`)`
Makes name a Python identifier by replacing all nonsafe characters with '_'

`cytoflow.utility.util_functions.`**`random_string`**`(`*n*`)`
Makes a random string of ascii digits and lowercase letters of length `n`

from http://stackoverflow.com/questions/2257441/random-string-generation-with-upper-case-letters-and-digits-in-python

`cytoflow.utility.util_functions.`**`is_numeric`**`(`*s*`)`
Determine if a `pandas.Series` or `numpy.ndarray` is numeric from its dtype.

`cytoflow.utility.util_functions.`**`cov2corr`**`(`*covariance*`)`
Compute the correlation matrix from the covariance matrix.

From https://github.com/AndreaCensi/procgraph/blob/master/src/procgraph_statistics/cov2corr.py

cytoflow.views package

cytoflow.views

This package contains all *`cytoflow`* views – classes implementing *`IView`* whose `plot()` function plots an experiment.

Submodules

cytoflow.views.bar_chart

Plot a bar chart from a statistic.

`BarChartView` – the *`IView`* class that makes the plot.

`class cytoflow.views.bar_chart.`**`BarChartView`**
Bases: *`cytoflow.views.base_views.Base1DStatisticsView`*

Plots a bar chart of some summary statistic

`statistic`
The name of the statistic to plot. Must be a key in the *`Experiment.statistics`* attribute of the *`Experiment`* being plotted.

Type (str, str)

`error_statistic`
The name of the statistic used to plot error bars. Must be a key in the *`Experiment.statistics`* attribute of the *`Experiment`* being plotted.

Type (str, str)

`scale`
The scale applied to the data before plotting it.

Type {'linear', 'log', 'logicle'}

`variable`
The condition that varies when plotting this statistic: used for the x axis of line plots, the bar groups in bar plots, etc.

Type str

subset
An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

xfacet
Set to one of the *Experiment.conditions* in the *Experiment*, and a new column of subplots will be added for every unique value of that condition.

Type String

yfacet
Set to one of the *Experiment.conditions* in the *Experiment*, and a new row of subplots will be added for every unique value of that condition.

Type String

huefacet
Set to one of the *Experiment.conditions* in the in the *Experiment*, and a new color will be added to the plot for every unique value of that condition.

Type String

huescale
How should the color scale for *huefacet* be scaled?

Type {'linear', 'log', 'logicle'}

Examples

Make a little data set.

```
>>> import cytoflow as flow
>>> import_op = flow.ImportOp()
>>> import_op.tubes = [flow.Tube(file = "Plate01/RFP_Well_A3.fcs",
...                               conditions = {'Dox' : 10.0}),
...                     flow.Tube(file = "Plate01/CFP_Well_A4.fcs",
...                               conditions = {'Dox' : 1.0})]
>>> import_op.conditions = {'Dox' : 'float'}
>>> ex = import_op.apply()
```

Add a threshold gate

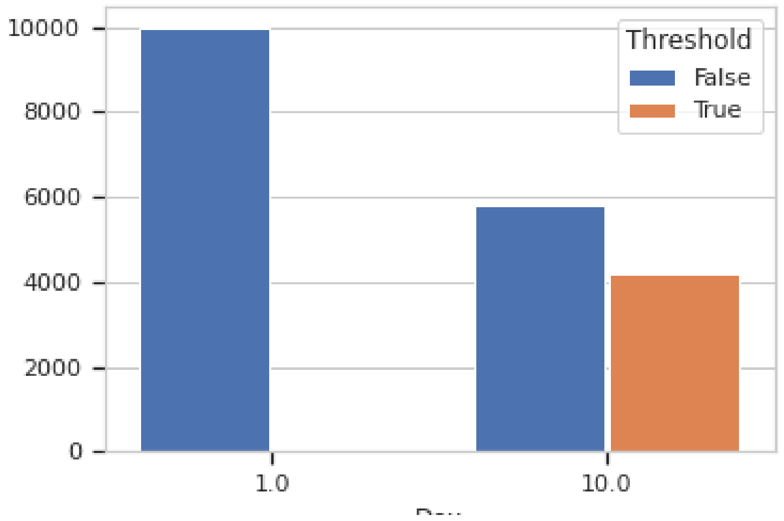
```
>>> ex2 = flow.ThresholdOp(name = 'Threshold',
...                        channel = 'Y2-A',
...                        threshold = 2000).apply(ex)
```

Add a statistic

```
>>> ex3 = flow.ChannelStatisticOp(name = "ByDox",
...                               channel = "Y2-A",
...                               by = ['Dox', 'Threshold'],
...                               function = len).apply(ex2)
```

Plot the bar chart

```
>>> flow.BarChartView(statistic = ("ByDox", "len"),
...                   variable = "Dox",
...                   huefacet = "Threshold").plot(ex3)
```



enum_plots(*experiment*)
Returns an iterator over the possible plots that this View can produce. The values returned can be passed to "plot".

plot(*experiment*, *plot_name=None*, ***kwargs*)
Plot a bar chart

Parameters

- **experiment** (*Experiment*) – The `Experiment` to plot using this view.
- **title** (*str*) – Set the plot title
- **xlabel** (*str*) – Set the X axis label
- **ylabel** (*str*) – Set the Y axis label
- **huelabel** (*str*) – Set the label for the hue facet (in the legend)
- **legend** (*bool*) – Plot a legend for the color or hue facet? Defaults to `True`.
- **sharex** (*bool*) – If there are multiple subplots, should they share X axes? Defaults to `True`.
- **sharey** (*bool*) – If there are multiple subplots, should they share Y axes? Defaults to `True`.
- **row_order** (*list*) – Override the row facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **col_order** (*list*) – Override the column facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **hue_order** (*list*) – Override the hue facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **height** (*float*) – The height of *each row* in inches. Default = 3.0
- **aspect** (*float*) – The aspect ratio *of each subplot*. Default = 1.5

- **col_wrap** (*int*) – If `xfacet` is set and `yfacet` is not set, you can "wrap" the subplots around so that they form a multi-row grid by setting this to the number of columns you want.
- **sns_style** (*{"darkgrid", "whitegrid", "dark", "white", "ticks"}*) – Which `seaborn` style to apply to the plot? Default is `whitegrid`.
- **sns_context** (*{"paper", "notebook", "talk", "poster"}*) – Which `seaborn` context to use? Controls the scaling of plot elements such as tick labels and the legend. Default is `talk`.
- **palette** (*palette name, list, or dict*) – Colors to use for the different levels of the hue variable. Should be something that can be interpreted by `seaborn.color_palette`, or a dictionary mapping hue levels to matplotlib colors.
- **despine** (*Bool*) – Remove the top and right axes from the plot? Default is `True`.
- **plot_name** (*str*) – If this `IView` can make multiple plots, `plot_name` is the name of the plot to make. Must be one of the values retrieved from `enum_plots`.
- **orientation** (*{'vertical', 'horizontal'}*)
- **lim** (*(float, float)*) – Set the range of the plot's axis.
- **color** (*a matplotlib color*) – Sets the colors of all the bars, even if there is a hue facet
- **errwidth** (*scalar*) – The width of the error bars, in points
- **errcolor** (*a matplotlib color*) – The color of the error bars
- **capsize** (*scalar*) – The size of the error bar caps, in points

Notes

Other `kwargs` are passed to matplotlib.axes.Axes.bar

cytoflow.views.base_views

Base classes for views.

I found, as I wrote a bunch of views, that I was also writing a bunch of shared boiler-plate code. This led to more bugs and a harder-to-maintain codebase. So, I extracted the copied code in a short hierarchy of reusable base classes:

`BaseView` – implements a view with row, column and hue facets. After setting up the facet grid, it calls the derived class's `_grid_plot` to actually do the plotting. `BaseView.plot` also has parameters to set the plot style, legend, axis labels, etc.

`BaseDataView` – implements a view that plots an `Experiment`'s data (as opposed to a statistic.) Includes functionality for subsetting the data before plotting, and determining axis limits and scales.

`Base1DView` – implements a 1-dimensional data view. See `HistogramView` for an example.

`Base2DView` – implements a 2-dimensional data view. See `ScatterplotView` for an example.

`BaseNDView` – implements an N-dimensional data view. See `RadvizView` for an example.

`BaseStatisticsView` – implements a view that plots a statistic from an `Experiment` (as opposed to the underlying data.) These views have a "primary" `BaseStatisticsView.variable`, and can be subset as well.

`Base1DStatisticsView` – implements a view that plots one dimension of a statistic. See `BarChartView` for an example.

Base2DStatisticsView – implements a view that plots two dimensions of a statistic. See *Stats2DView* for an example.

class `cytoflow.views.base_views.`**`BaseView`**

Bases: `traits.has_traits.HasStrictTraits`

The base class for facetted plotting.

`xfacet`

Set to one of the *Experiment.conditions* in the *Experiment*, and a new column of subplots will be added for every unique value of that condition.

Type String

`yfacet`

Set to one of the *Experiment.conditions* in the *Experiment*, and a new row of subplots will be added for every unique value of that condition.

Type String

`huefacet`

Set to one of the *Experiment.conditions* in the in the *Experiment*, and a new color will be added to the plot for every unique value of that condition.

Type String

`huescale`

How should the color scale for *huefacet* be scaled?

Type {'linear', 'log', 'logicle'}

`plot`(*experiment*, *data*, ***kwargs*)

Base function for facetted plotting

Parameters

- **experiment** (*Experiment*) – The *Experiment* to plot using this view.
- **title** (*str*) – Set the plot title
- **xlabel** (*str*) – Set the X axis label
- **ylabel** (*str*) – Set the Y axis label
- **huelabel** (*str*) – Set the label for the hue facet (in the legend)
- **legend** (*bool*) – Plot a legend for the color or hue facet? Defaults to `True`.
- **sharex** (*bool*) – If there are multiple subplots, should they share X axes? Defaults to `True`.
- **sharey** (*bool*) – If there are multiple subplots, should they share Y axes? Defaults to `True`.
- **row_order** (*list*) – Override the row facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **col_order** (*list*) – Override the column facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **hue_order** (*list*) – Override the hue facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **height** (*float*) – The height of *each row* in inches. Default = 3.0
- **aspect** (*float*) – The aspect ratio *of each subplot*. Default = 1.5

- **col_wrap** (*int*) – If `xfacet` is set and `yfacet` is not set, you can "wrap" the subplots around so that they form a multi-row grid by setting this to the number of columns you want.
- **sns_style** (*{"darkgrid", "whitegrid", "dark", "white", "ticks"}*) – Which `seaborn` style to apply to the plot? Default is `whitegrid`.
- **sns_context** (*{"paper", "notebook", "talk", "poster"}*) – Which `seaborn` context to use? Controls the scaling of plot elements such as tick labels and the legend. Default is `talk`.
- **palette** (*palette name, list, or dict*) – Colors to use for the different levels of the hue variable. Should be something that can be interpreted by `seaborn.color_palette`, or a dictionary mapping hue levels to matplotlib colors.
- **despine** (*Bool*) – Remove the top and right axes from the plot? Default is `True`.

Other Parameters

- **cmap** (*matplotlib colormap*) – If plotting a huefacet with many values, use this color map instead of the default.
- **norm** (*matplotlib.colors.Normalize*) – If plotting a huefacet with many values, use this object for color scale normalization.

`class cytoflow.views.base_views.`**`BaseDataView`**

Bases: `cytoflow.views.base_views.BaseView`

The base class for data views (as opposed to statistics views).

`subset`

An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

`plot`(*experiment*, ***kwargs*)

Plot some data from an experiment. This function takes care of checking for facet name validity and subsetting, then passes the underlying dataframe to `BaseView.plot`

Parameters

- **min_quantile** (*float (>0.0 and <1.0, default = 0.001)*) – Clip data that is less than this quantile.
- **max_quantile** (*float (>0.0 and <1.0, default = 1.00)*) – Clip data that is greater than this quantile.

Other Parameters

- **lim** (*Dict(Str : (float, float))*) – Set the range of each channel's axis. If unspecified, assume that the limits are the minimum and maximum of the clipped data. Required.
- **scale** (*Dict(Str : IScale)*) – Scale the data on each axis. Required.

`class cytoflow.views.base_views.`**`Base1DView`**

Bases: `cytoflow.views.base_views.BaseDataView`

A data view that plots data from a single channel.

`channel`

The channel to view

Type Str

`scale`
The scale applied to the data before plotting it.

Type {'linear', 'log', 'logicle'}

`plot`(*experiment*, ***kwargs*)

Parameters

- **lim** (*(float, float)*) – Set the range of the plot's data axis.
- **orientation** (*{'vertical', 'horizontal'}*)

class `cytoflow.views.base_views.Base2DView`
Bases: *cytoflow.views.base_views.BaseDataView*

A data view that plots data from two channels.

`xchannel`
The channel to view on the X axis

Type Str

`ychannel`
The channel to view on the Y axis

Type Str

`xscale`
The scales applied to the *xchannel* data before plotting it.

Type {'linear', 'log', 'logicle'} (default = 'linear')

`yscale`
The scales applied to the *ychannel* data before plotting it.

Type {'linear', 'log', 'logicle'} (default = 'linear')

`plot`(*experiment*, ***kwargs*)

Parameters

- **xlim** (*(float, float)*) – Set the range of the plot's X axis.
- **ylim** (*(float, float)*) – Set the range of the plot's Y axis.

class `cytoflow.views.base_views.BaseNDView`
Bases: *cytoflow.views.base_views.BaseDataView*

A data view that plots data from one or more channels.

`channels`
The channels to view

Type List(Str)

`scale`
Re-scale the data in the specified channels before plotting. If a channel isn't specified, assume that the scale is linear.

Type Dict(Str : {"linear", "logicle", "log"})

`plot`(*experiment*, ***kwargs*)

Parameters **lim** (*Dict(Str : (float, float))*) – Set the range of each channel's axis. If unspecified, assume that the limits are the minimum and maximum of the clipped data

`class cytoflow.views.base_views.`**`BaseStatisticsView`**

Bases: *`cytoflow.views.base_views.BaseView`*

The base class for statistics views (as opposed to data views).

`variable`

The condition that varies when plotting this statistic: used for the x axis of line plots, the bar groups in bar plots, etc.

Type str

`subset`

An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

`enum_plots``(`*experiment*, *data*`)`

Enumerate the named plots we can make from this set of statistics.

Returns An iterator across the possible plot names. The iterator ALSO has an instance attribute called `by`, which holds a list of the facets that are not yet set (and thus need to be specified in the plot name.)

Return type iterator

`plot``(`*experiment*, *data*, *plot_name=None*, ***kwargs*`)`

Plot some data from a statistic.

This function takes care of checking for facet name validity and subsetting, then passes the dataframe to *`BaseView.plot`*

Parameters **plot_name** (*str*) – If this *`IView`* can make multiple plots, `plot_name` is the name of the plot to make. Must be one of the values retrieved from *`enum_plots`*.

`class cytoflow.views.base_views.`**`Base1DStatisticsView`**

Bases: *`cytoflow.views.base_views.BaseStatisticsView`*

The base class for 1-dimensional statistic views – ie, the *`variable`* attribute is on the x axis, and the statistic value is on the y axis.

`statistic`

The name of the statistic to plot. Must be a key in the *`Experiment.statistics`* attribute of the *`Experiment`* being plotted.

Type (str, str)

`error_statistic`

The name of the statistic used to plot error bars. Must be a key in the *`Experiment.statistics`* attribute of the *`Experiment`* being plotted.

Type (str, str)

`scale`

The scale applied to the data before plotting it.

Type {'linear', 'log', 'logicle'}

`enum_plots``(`*experiment*`)`

Enumerate the named plots we can make from this set of statistics.

Returns An iterator across the possible plot names. The iterator ALSO has an instance attribute called by, which holds a list of the facets that are not yet set (and thus need to be specified in the plot name.)

Return type iterator

`plot`(*experiment*, *plot_name=None*, ***kwargs*)

Parameters

- **orientation** (*{'vertical', 'horizontal'}*)
- **lim** (*(float, float)*) – Set the range of the plot's axis.

class `cytoflow.views.base_views.Base2DStatisticsView`

Bases: `cytoflow.views.base_views.BaseStatisticsView`

The base class for 2-dimensional statistic views – ie, the `variable` attribute varies independently, and the corresponding values from the x and y statistics are plotted on the x and y axes.

`xstatistic`

The name of the statistic to plot on the X axis. Must be a key in the `Experiment.statistics` attribute of the `Experiment` being plotted.

Type (str, str)

`ystatistic`

The name of the statistic to plot on the Y axis. Must be a key in the `Experiment.statistics` attribute of the `Experiment` being plotted.

Type (str, str)

`x_error_statistic`

The name of the statistic used to plot error bars on the X axis. Must be a key in the `Experiment.statistics` attribute of the `Experiment` being plotted.

Type (str, str)

`y_error_statistic`

The name of the statistic used to plot error bars on the Y axis. Must be a key in the `Experiment.statistics` attribute of the `Experiment` being plotted.

Type (str, str)

`xscale`

The scale applied to `xstatistic` before plotting it.

Type {'linear', 'log', 'logicle'}

`yscale`

The scale applied to `ystatistic` before plotting it.

Type {'linear', 'log', 'logicle'}

`enum_plots`(*experiment*)

Enumerate the named plots we can make from this set of statistics.

Returns An iterator across the possible plot names. The iterator ALSO has an instance attribute called by, which holds a list of the facets that are not yet set (and thus need to be specified in the plot name.)

Return type iterator

`plot`(*experiment*, *plot_name=None*, ***kwargs*)

Parameters

- **xlim** (*(float, float)*) – Set the range of the plot's X axis.
- **ylim** (*(float, float)*) – Set the range of the plot's Y axis.

cytoflow.views.densityplot

Plot a 2D density plot.

`DensityView` – the *`IView`* class that makes the plot.

`class cytoflow.views.densityplot.`**`DensityView`**

Bases: *`cytoflow.views.base_views.Base2DView`*

Plots a 2-d density plot.

`huefacet`

You must leave the hue facet unset!

Type None

`xchannel`

The channel to view on the X axis

Type Str

`ychannel`

The channel to view on the Y axis

Type Str

`xscale`

The scales applied to the *`xchannel`* data before plotting it.

Type {'linear', 'log', 'logicle'} (default = 'linear')

`yscale`

The scales applied to the *`ychannel`* data before plotting it.

Type {'linear', 'log', 'logicle'} (default = 'linear')

`subset`

An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

`xfacet`

Set to one of the *`Experiment.conditions`* in the *`Experiment`*, and a new column of subplots will be added for every unique value of that condition.

Type String

`yfacet`

Set to one of the *`Experiment.conditions`* in the *`Experiment`*, and a new row of subplots will be added for every unique value of that condition.

Type String

`huescale`

How should the color scale for *`huefacet`* be scaled?

Type {'linear', 'log', 'logicle'}

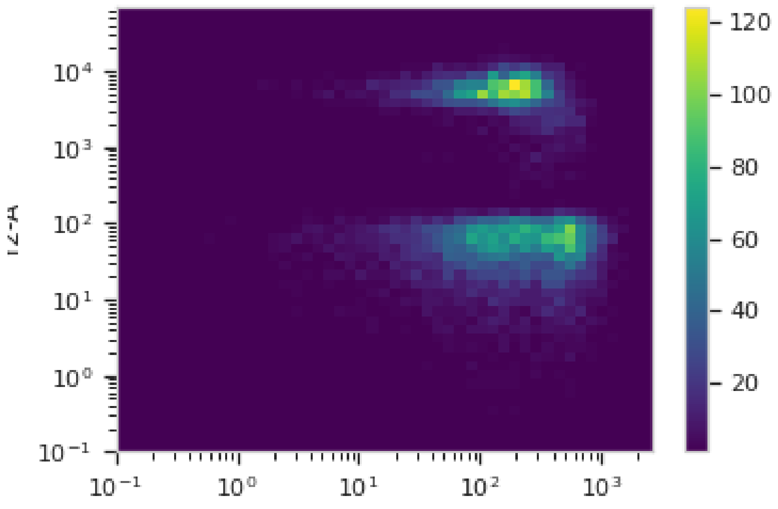
Examples

Make a little data set.

```
>>> import cytoflow as flow
>>> import_op = flow.ImportOp()
>>> import_op.tubes = [flow.Tube(file = "Plate01/RFP_Well_A3.fcs",
...                              conditions = {'Dox' : 10.0}),
...                     flow.Tube(file = "Plate01/CFP_Well_A4.fcs",
...                              conditions = {'Dox' : 1.0})]
>>> import_op.conditions = {'Dox' : 'float'}
>>> ex = import_op.apply()
```

Plot a density plot

```
>>> flow.DensityView(xchannel = 'V2-A',
...                  xscale = 'log',
...                  ychannel = 'Y2-A',
...                  yscale = 'log').plot(ex)
```



The same plot, smoothed, with a log color scale. *Note - you can change the hue scale, even if you don't have control over the hue facet!*

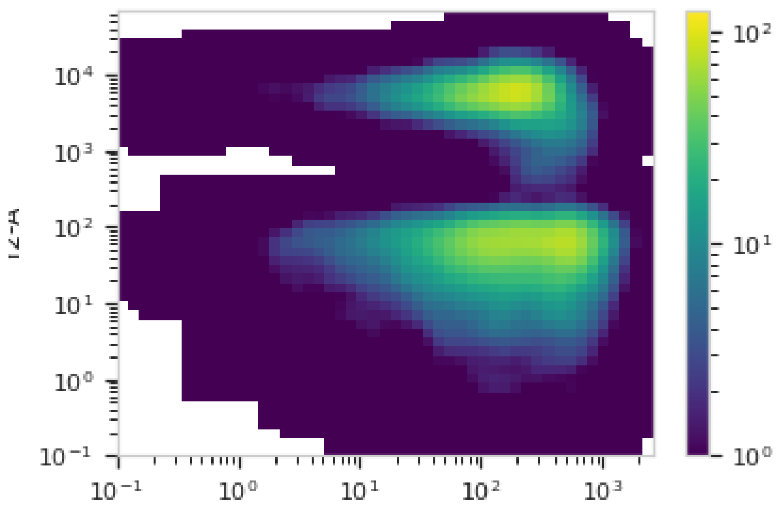
```
>>> flow.DensityView(xchannel = 'V2-A',
...                   xscale = 'log',
...                   ychannel = 'Y2-A',
...                   yscale = 'log',
...                   huescale = 'log').plot(ex, smoothed = True)
```

plot(*experiment*, ***kwargs*)

Plot a faceted density plot view of a channel

Parameters

- **experiment** (*Experiment*) – The `Experiment` to plot using this view.
- **title** (*str*) – Set the plot title



- **xlabel** (*str*) – Set the X axis label
- **ylabel** (*str*) – Set the Y axis label
- **huelabel** (*str*) – Set the label for the hue facet (in the legend)
- **legend** (*bool*) – Plot a legend for the color or hue facet? Defaults to `True`.
- **sharex** (*bool*) – If there are multiple subplots, should they share X axes? Defaults to `True`.
- **sharey** (*bool*) – If there are multiple subplots, should they share Y axes? Defaults to `True`.
- **row_order** (*list*) – Override the row facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **col_order** (*list*) – Override the column facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **hue_order** (*list*) – Override the hue facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **height** (*float*) – The height of *each row* in inches. Default = 3.0
- **aspect** (*float*) – The aspect ratio *of each subplot*. Default = 1.5
- **col_wrap** (*int*) – If `xfacet` is set and `yfacet` is not set, you can "wrap" the subplots around so that they form a multi-row grid by setting this to the number of columns you want.
- **sns_style** (*{"darkgrid", "whitegrid", "dark", "white", "ticks"}*) – Which `seaborn` style to apply to the plot? Default is `whitegrid`.
- **sns_context** (*{"paper", "notebook", "talk", "poster"}*) – Which `seaborn` context to use? Controls the scaling of plot elements such as tick labels and the legend. Default is `talk`.
- **palette** (*palette name, list, or dict*) – Colors to use for the different levels of the hue variable. Should be something that can be interpreted by `seaborn.color_palette`, or a dictionary mapping hue levels to matplotlib colors.
- **despine** (*Bool*) – Remove the top and right axes from the plot? Default is `True`.

- **min_quantile** (*float (>0.0 and <1.0, default = 0.001)*) – Clip data that is less than this quantile.
- **max_quantile** (*float (>0.0 and <1.0, default = 1.00)*) – Clip data that is greater than this quantile.
- **xlim** (*(float, float)*) – Set the range of the plot's X axis.
- **ylim** (*(float, float)*) – Set the range of the plot's Y axis.
- **gridsize** (*int*) – The size of the grid on each axis. Default = 50
- **smoothed** (*bool*) – Should the resulting mesh be smoothed?
- **smoothed_sigma** (*int*) – The standard deviation of the smoothing kernel. default = 1.
- **cmap** (*cmap*) – An instance of matplotlib.colors.Colormap. By default, the `viridis` colormap is used
- **under_color** (*matplotlib color*) – Sets the color to be used for low out-of-range values.
- **bad_color** (*matplotlib color*) – Set the color to be used for masked values.

Notes

Other `kwargs` are passed to `matplotlib.axes.Axes.pcolormesh`

cytoflow.views.export_fcs

A "view" that exports events as FCS files.

`ExportFCS` – the *`IView`* class that does the exporting.

class `cytoflow.views.export_fcs.`**`ExportFCS`**

Bases: `traits.has_traits.HasStrictTraits`

Exports events as FCS files.

This isn't a traditional view, in that it doesn't implement *`plot`*. Instead, use *`enum_files`* to figure out which files will be created from a particular experiment, and *`export`* to export the FCS files.

The Cytoflow attributes will be encoded in keywords in the FCS TEXT segment, starting with the characters `CF_`. Any FCS keywords that are the same across all the input files will also be included.

`base`

The prefix of the FCS filenames

Type Str

`path`

The directory to export to.

Type Directory

`by`

A list of conditions from *`Experiment.conditions`*; each unique combination of conditions will be exported to an FCS file.

Type List(Str)

`keywords`

If you want to add more keywords to the FCS files' TEXT segment, specify them here.

Type Dict(Str, Str)

subset
A Python expression used to select a subset of the data

Type str

Examples

Make a little data set.

```
>>> import cytoflow as flow
>>> import_op = flow.ImportOp()
>>> import_op.tubes = [flow.Tube(file = "Plate01/RFP_Well_A3.fcs",
...                              conditions = {'Dox' : 10.0}),
...                    flow.Tube(file = "Plate01/CFP_Well_A4.fcs",
...                              conditions = {'Dox' : 1.0})]
>>> import_op.conditions = {'Dox' : 'float'}
>>> ex = import_op.apply()
```

Export the data

```
>>> import tempfile
>>> flow.ExportFCS(path = 'export/',
...                by = ["Dox"],
...                subset = "Dox == 10.0").export(ex)
```

enum_files(*experiment*)
Return an iterator over the file names that this export module will produce from a given experiment.

Parameters **experiment** (*Experiment*) – The *Experiment* to export

export(*experiment*)
Export FCS files from an experiment.

Parameters **experiment** (*Experiment*) – The *Experiment* to export

cytoflow.views.histogram

Plots a histogram.

HistogramView – the *IView* class that makes the plot.

class cytoflow.views.histogram.**HistogramView**
Bases: *cytoflow.views.base_views.Base1DView*

Plots a one-channel histogram

channel
The channel to view

Type Str

scale
The scale applied to the data before plotting it.

Type {‘linear’, ‘log’, ‘logicle’}

subset
An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

xfacet
Set to one of the *Experiment.conditions* in the *Experiment*, and a new column of subplots will be added for every unique value of that condition.

Type String

yfacet
Set to one of the *Experiment.conditions* in the *Experiment*, and a new row of subplots will be added for every unique value of that condition.

Type String

huefacet
Set to one of the *Experiment.conditions* in the in the *Experiment*, and a new color will be added to the plot for every unique value of that condition.

Type String

huescale
How should the color scale for *huefacet* be scaled?

Type {'linear', 'log', 'logicle'}

Examples

Make a little data set.

```
>>> import cytoflow as flow
>>> import_op = flow.ImportOp()
>>> import_op.tubes = [flow.Tube(file = "Plate01/RFP_Well_A3.fcs",
...                               conditions = {'Dox' : 10.0}),
...                     flow.Tube(file = "Plate01/CFP_Well_A4.fcs",
...                               conditions = {'Dox' : 1.0})]
>>> import_op.conditions = {'Dox' : 'float'}
>>> ex = import_op.apply()
```

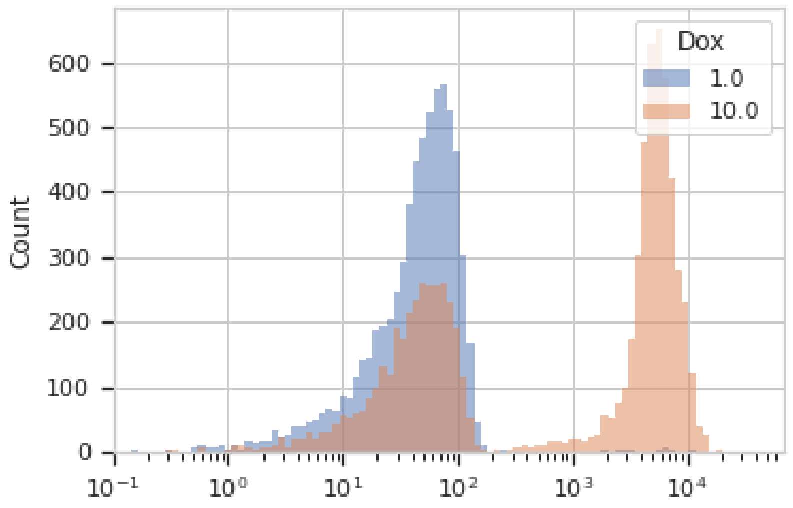
Plot a histogram

```
>>> flow.HistogramView(channel = 'Y2-A',
...                     scale = 'log',
...                     huefacet = 'Dox').plot(ex)
```

plot(*experiment*, ***kwargs*)
Plot a faceted histogram view of a channel

Parameters

- **experiment** (*Experiment*) – The *Experiment* to plot using this view.
- **title** (*str*) – Set the plot title
- **xlabel** (*str*) – Set the X axis label
- **ylabel** (*str*) – Set the Y axis label



- **huelabel** (*str*) – Set the label for the hue facet (in the legend)
- **legend** (*bool*) – Plot a legend for the color or hue facet? Defaults to `True`.
- **sharex** (*bool*) – If there are multiple subplots, should they share X axes? Defaults to `True`.
- **sharey** (*bool*) – If there are multiple subplots, should they share Y axes? Defaults to `True`.
- **row_order** (*list*) – Override the row facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **col_order** (*list*) – Override the column facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **hue_order** (*list*) – Override the hue facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **height** (*float*) – The height of *each row* in inches. Default = 3.0
- **aspect** (*float*) – The aspect ratio *of each subplot*. Default = 1.5
- **col_wrap** (*int*) – If `xfacet` is set and `yfacet` is not set, you can "wrap" the subplots around so that they form a multi-row grid by setting this to the number of columns you want.
- **sns_style** (*{"darkgrid", "whitegrid", "dark", "white", "ticks"}*) – Which `seaborn` style to apply to the plot? Default is `whitegrid`.
- **sns_context** (*{"paper", "notebook", "talk", "poster"}*) – Which `seaborn` context to use? Controls the scaling of plot elements such as tick labels and the legend. Default is `talk`.
- **palette** (*palette name, list, or dict*) – Colors to use for the different levels of the hue variable. Should be something that can be interpreted by `seaborn.color_palette`, or a dictionary mapping hue levels to matplotlib colors.
- **despine** (*Bool*) – Remove the top and right axes from the plot? Default is `True`.
- **min_quantile** (*float (>0.0 and <1.0, default = 0.001)*) – Clip data that is less than this quantile.
- **max_quantile** (*float (>0.0 and <1.0, default = 1.00)*) – Clip data that is greater than this quantile.

- **lim** (*(float, float)*) – Set the range of the plot's data axis.
- **orientation** (*{'vertical', 'horizontal'}*)
- **num_bins** (*int*) – The number of bins to plot in the histogram. Clipped to [100, 1000]
- **histtype** (*{'stepfilled', 'step', 'bar'}*) – The type of histogram to draw. `stepfilled` is the default, which is a line plot with a color filled under the curve.
- **density** (*bool*) – If `True`, re-scale the histogram to form a probability density function, so the area under the histogram is 1.
- **linewidth** (*float*) – The width of the histogram line (in points)
- **linestyle** (*['-' | '–' | '-.' | ':' | "None"]*) – The style of the line to plot
- **alpha** (*float (default = 0.5)*) – The alpha blending value, between 0 (transparent) and 1 (opaque).

Notes

Other `kwargs` are passed to matplotlib.pyplot.hist

cytoflow.views.histogram_2d

Plot a 2D histogram.

`Histogram2DView` – the *`IView`* class that makes the plot.

`class cytoflow.views.histogram_2d.Histogram2DView`

Bases: *`cytoflow.views.base_views.Base2DView`*

Plots a 2-d histogram. Similar to a density plot, but the number of events in a bin change the bin's opacity, so you can use different colors.

`xchannel`

The channel to view on the X axis

Type Str

`ychannel`

The channel to view on the Y axis

Type Str

`xscale`

The scales applied to the *`xchannel`* data before plotting it.

Type {'linear', 'log', 'logicle'} (default = 'linear')

`yscale`

The scales applied to the *`ychannel`* data before plotting it.

Type {'linear', 'log', 'logicle'} (default = 'linear')

`subset`

An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

xfacet
Set to one of the *Experiment.conditions* in the *Experiment*, and a new column of subplots will be added for every unique value of that condition.

Type String

yfacet
Set to one of the *Experiment.conditions* in the *Experiment*, and a new row of subplots will be added for every unique value of that condition.

Type String

huefacet
Set to one of the *Experiment.conditions* in the in the *Experiment*, and a new color will be added to the plot for every unique value of that condition.

Type String

huescale
How should the color scale for *huefacet* be scaled?

Type {'linear', 'log', 'logicle'}

Examples

Make a little data set.

```
>>> import cytoflow as flow
>>> import_op = flow.ImportOp()
>>> import_op.tubes = [flow.Tube(file = "Plate01/RFP_Well_A3.fcs",
...                              conditions = {'Dox' : 10.0}),
...                    flow.Tube(file = "Plate01/CFP_Well_A4.fcs",
...                              conditions = {'Dox' : 1.0})]
>>> import_op.conditions = {'Dox' : 'float'}
>>> ex = import_op.apply()
```

Plot a density plot

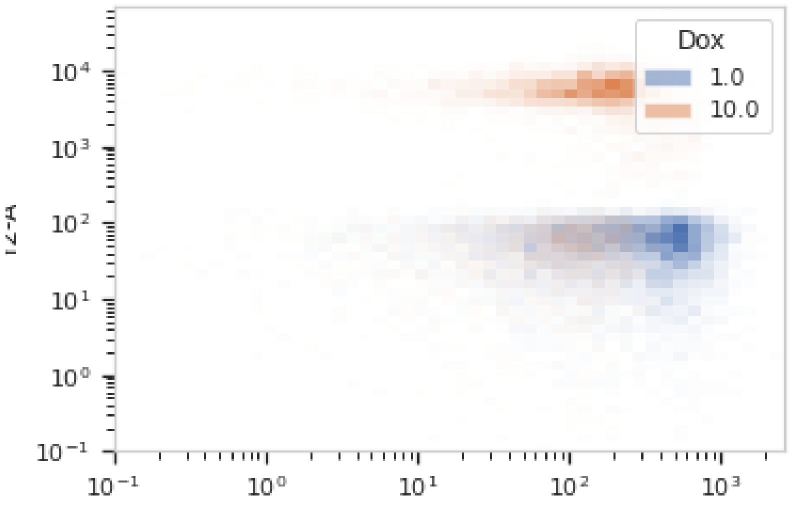
```
>>> flow.Histogram2DView(xchannel = 'V2-A',
...                      xscale = 'log',
...                      ychannel = 'Y2-A',
...                      yscale = 'log',
...                      huefacet = 'Dox').plot(ex)
```

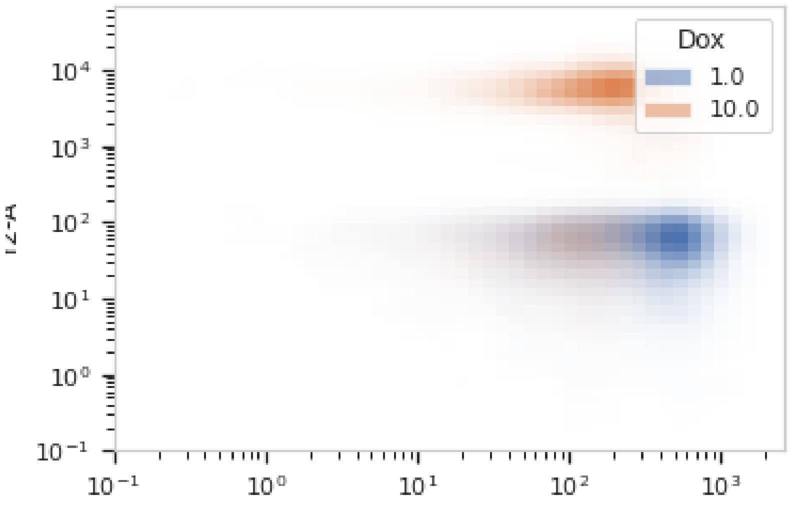
The same plot, smoothed, with a log color scale.

```
>>> flow.Histogram2DView(xchannel = 'V2-A',
...                      xscale = 'log',
...                      ychannel = 'Y2-A',
...                      yscale = 'log',
...                      huefacet = 'Dox',
...                      huescale = 'log').plot(ex, smoothed = True)
```

id = 'edu.mit.synbio.cytoflow.view.histogram2d'

friend_id = '2D Histogram'


Dox
1.0
10.0
Y2-A
10⁻¹
10⁰
10¹
10²
10³
10⁴


Dox
1.0
10.0
Y2-A
10⁻¹
10⁰
10¹
10²
10³
10⁴

`plot`(*experiment*, ***kwargs*)

Plot a faceted density plot view of a channel

Parameters

- **experiment** (*Experiment*) – The `Experiment` to plot using this view.
- **title** (*str*) – Set the plot title
- **xlabel** (*str*) – Set the X axis label
- **ylabel** (*str*) – Set the Y axis label
- **huelabel** (*str*) – Set the label for the hue facet (in the legend)
- **legend** (*bool*) – Plot a legend for the color or hue facet? Defaults to `True`.
- **sharex** (*bool*) – If there are multiple subplots, should they share X axes? Defaults to `True`.
- **sharey** (*bool*) – If there are multiple subplots, should they share Y axes? Defaults to `True`.
- **row_order** (*list*) – Override the row facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **col_order** (*list*) – Override the column facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **hue_order** (*list*) – Override the hue facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **height** (*float*) – The height of *each row* in inches. Default = 3.0
- **aspect** (*float*) – The aspect ratio *of each subplot*. Default = 1.5
- **col_wrap** (*int*) – If `xfacet` is set and `yfacet` is not set, you can "wrap" the subplots around so that they form a multi-row grid by setting this to the number of columns you want.
- **sns_style** (*{"darkgrid", "whitegrid", "dark", "white", "ticks"}*) – Which `seaborn` style to apply to the plot? Default is `whitegrid`.
- **sns_context** (*{"paper", "notebook", "talk", "poster"}*) – Which `seaborn` context to use? Controls the scaling of plot elements such as tick labels and the legend. Default is `talk`.
- **palette** (*palette name, list, or dict*) – Colors to use for the different levels of the hue variable. Should be something that can be interpreted by `seaborn.color_palette`, or a dictionary mapping hue levels to matplotlib colors.
- **despine** (*Bool*) – Remove the top and right axes from the plot? Default is `True`.
- **min_quantile** (*float (>0.0 and <1.0, default = 0.001)*) – Clip data that is less than this quantile.
- **max_quantile** (*float (>0.0 and <1.0, default = 1.00)*) – Clip data that is greater than this quantile.
- **xlim** (*(float, float)*) – Set the range of the plot's X axis.
- **ylim** (*(float, float)*) – Set the range of the plot's Y axis.
- **gridsize** (*int*) – The number of bins on the X and Y axis.
- **smoothed** (*bool*) – Should the mesh be smoothed?
- **smoothed_sigma** (*int*) – The standard deviation of the smoothing kernel. default = 1.

Notes

Other `kwargs` are passed to `matplotlib.axes.Axes.pcolormesh`

class `cytoflow.views.histogram_2d.`**`AlphaColormap`**(*name*, *color*, *min_alpha=0.0*, *max_alpha=1.0*, *N=256*)

Bases: `matplotlib.colors.Colormap`

cytoflow.views.i_selectionview

`ISelectionView` – a decorator that lets you add selections to an *`IView`*.

class `cytoflow.views.i_selectionview.`**`ISelectionView`**(*adaptee*, *default=<class 'traits.adaptation.adaptation_error.AdaptationError'>*)

Bases: *`cytoflow.views.i_view.IView`*

A decorator that lets you add (possibly interactive) selections to an IView.

Note that this is a Decorator *design pattern*, not a Python `@decorator`.

`interactive`

Is this view's interactivity turned on?

Type Bool

cytoflow.views.i_view

`IView` – an interface for the visualization of flow data.

class `cytoflow.views.i_view.`**`IView`**(*adaptee*, *default=<class 'traits.adaptation.adaptation_error.AdaptationError'>*)

Bases: `traits.has_traits.Interface`

An interface for a visualization of flow data.

Could be a histogram, a density plot, a scatter plot, a statistical visualization like a bar chart of population means; even a textual representation like a table.

`id`

A unique id for this view. Prefix: "edu.mit.cytoflow.views"

Type Constant

`friendly_id`

The human-readable id of this view: eg, "Histogram"

Type Constant

`plot`(*experiment*, ***kwargs*)

Plot a visualization of flow data using the pyplot stateful interface

Parameters

- **experiment** (*Experiment*) – the Experiment containing the data to plot
- **kwargs** (*dict*) – additional arguments to pass to the underlying plotting function.

cytoflow.views.kde_1d

A one-channel kernel density estimate (like a smoothed histogram).

`Kde1DView` – the `IView` class that makes the plot.

class `cytoflow.views.kde_1d.Kde1DView`

Bases: `cytoflow.views.base_views.Base1DView`

Plots a one-channel kernel density estimate, which is like a smoothed histogram.

channel
The channel to view

Type Str

scale
The scale applied to the data before plotting it.

Type {'linear', 'log', 'logicle'}

subset
An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

xfacet
Set to one of the `Experiment.conditions` in the `Experiment`, and a new column of subplots will be added for every unique value of that condition.

Type String

yfacet
Set to one of the `Experiment.conditions` in the `Experiment`, and a new row of subplots will be added for every unique value of that condition.

Type String

huefacet
Set to one of the `Experiment.conditions` in the in the `Experiment`, and a new color will be added to the plot for every unique value of that condition.

Type String

huescale
How should the color scale for `huefacet` be scaled?

Type {'linear', 'log', 'logicle'}

Examples

Make a little data set.

```
>>> import cytoflow as flow
>>> import_op = flow.ImportOp()
>>> import_op.tubes = [flow.Tube(file = "Plate01/RFP_Well_A3.fcs",
...                              conditions = {'Dox' : 10.0}),
...                    flow.Tube(file = "Plate01/CFP_Well_A4.fcs",
...                              conditions = {'Dox' : 1.0})]
```

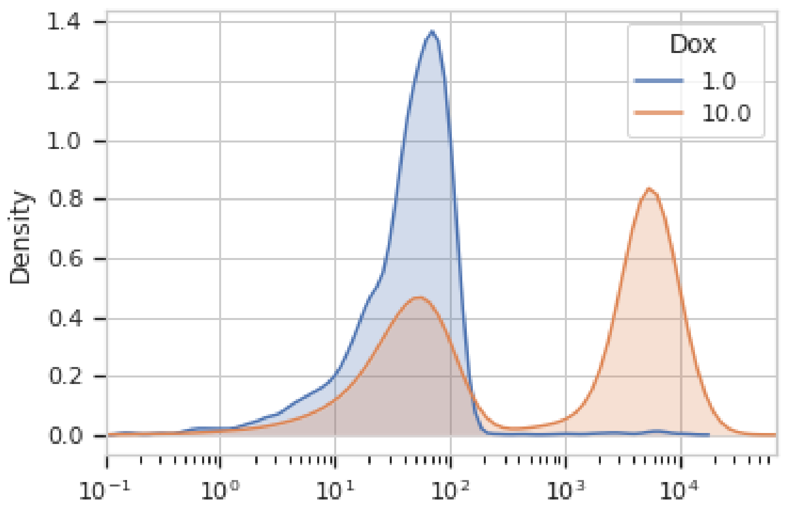
(continues on next page)

(continued from previous page)

```
>>> import_op.conditions = {'Dox' : 'float'}
>>> ex = import_op.apply()
```

Plot a histogram

```
>>> flow.Kde1DView(channel = 'Y2-A',
...                scale = 'log',
...                huefacet = 'Dox').plot(ex)
```



plot(*experiment*, ***kwargs*)

Plot a smoothed histogram view of a channel

Parameters

- **experiment** (*Experiment*) – The `Experiment` to plot using this view.
- **title** (*str*) – Set the plot title
- **xlabel** (*str*) – Set the X axis label
- **ylabel** (*str*) – Set the Y axis label
- **huelabel** (*str*) – Set the label for the hue facet (in the legend)
- **legend** (*bool*) – Plot a legend for the color or hue facet? Defaults to `True`.
- **sharex** (*bool*) – If there are multiple subplots, should they share X axes? Defaults to `True`.
- **sharey** (*bool*) – If there are multiple subplots, should they share Y axes? Defaults to `True`.
- **row_order** (*list*) – Override the row facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **col_order** (*list*) – Override the column facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **hue_order** (*list*) – Override the hue facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **height** (*float*) – The height of *each row* in inches. Default = 3.0

- **aspect** (*float*) – The aspect ratio *of each subplot*. Default = 1.5
- **col_wrap** (*int*) – If `xfacet` is set and `yfacet` is not set, you can "wrap" the subplots around so that they form a multi-row grid by setting this to the number of columns you want.
- **sns_style** (*{"darkgrid", "whitegrid", "dark", "white", "ticks"}*) – Which `seaborn` style to apply to the plot? Default is `whitegrid`.
- **sns_context** (*{"paper", "notebook", "talk", "poster"}*) – Which `seaborn` context to use? Controls the scaling of plot elements such as tick labels and the legend. Default is `talk`.
- **palette** (*palette name, list, or dict*) – Colors to use for the different levels of the hue variable. Should be something that can be interpreted by `seaborn.color_palette`, or a dictionary mapping hue levels to matplotlib colors.
- **despine** (*Bool*) – Remove the top and right axes from the plot? Default is `True`.
- **min_quantile** (*float (>0.0 and <1.0, default = 0.001)*) – Clip data that is less than this quantile.
- **max_quantile** (*float (>0.0 and <1.0, default = 1.00)*) – Clip data that is greater than this quantile.
- **lim** (*(float, float)*) – Set the range of the plot's data axis.
- **orientation** (*{'vertical', 'horizontal'}*)
- **shade** (*bool*) – If `True` (the default), shade the area under the plot.
- **alpha** (*float, >=0 and <= 1*) – The transparency of the shading. 1 is opaque, 0 is transparent. Default = 0.25
- **kernel** (*str*) – The kernel to use for the kernel density estimate. Choices are:
- **bw** (*str or float*) – The bandwidth for the kernel, controls how lumpy or smooth the kernel estimate is. Choices are:
- **gridsize** (*int (default = 100)*) – How many times to compute the kernel?

Notes

Other `kwargs` are passed to matplotlib.pyplot.plot

cytoflow.views.kde_2d

A two-dimensional kernel density estimate – kind of like a data "topo" map.

`Kde2DView` – the *`IView`* class that makes the plot.

class `cytoflow.views.kde_2d.`**`Kde2DView`**

Bases: *`cytoflow.views.base_views.Base2DView`*

Plots a 2-d kernel-density estimate. Sort of like a smoothed histogram. The density is visualized with a set of isolines.

`xchannel`

The channel to view on the X axis

Type Str

ychannel
The channel to view on the Y axis

Type Str

xscale
The scales applied to the *xchannel* data before plotting it.

Type {'linear', 'log', 'logicle'} (default = 'linear')

yscale
The scales applied to the *ychannel* data before plotting it.

Type {'linear', 'log', 'logicle'} (default = 'linear')

subset
An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

xfacet
Set to one of the *Experiment.conditions* in the *Experiment*, and a new column of subplots will be added for every unique value of that condition.

Type String

yfacet
Set to one of the *Experiment.conditions* in the *Experiment*, and a new row of subplots will be added for every unique value of that condition.

Type String

huefacet
Set to one of the *Experiment.conditions* in the in the *Experiment*, and a new color will be added to the plot for every unique value of that condition.

Type String

huescale
How should the color scale for *huefacet* be scaled?

Type {'linear', 'log', 'logicle'}

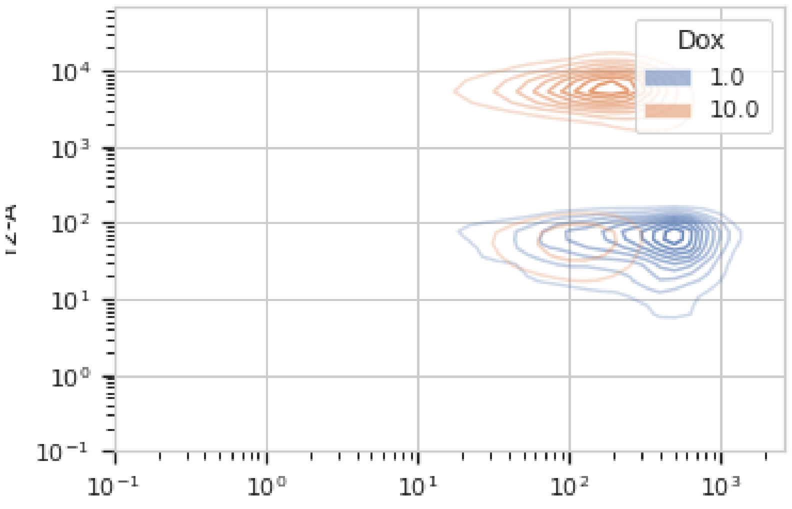
Examples

Make a little data set.

```
>>> import cytoflow as flow
>>> import_op = flow.ImportOp()
>>> import_op.tubes = [flow.Tube(file = "Plate01/RFP_Well_A3.fcs",
...                              conditions = {'Dox' : 10.0}),
...                    flow.Tube(file = "Plate01/CFP_Well_A4.fcs",
...                              conditions = {'Dox' : 1.0})]
>>> import_op.conditions = {'Dox' : 'float'}
>>> ex = import_op.apply()
```

Plot a density plot

```
>>> flow.Kde2DView(xchannel = 'V2-A',
...                xscale = 'log',
...                ychannel = 'Y2-A',
...                yscale = 'log',
...                huefacet = 'Dox').plot(ex)
```



plot(*experiment*, ***kwargs*)

Plot a faceted 2d kernel density estimate

Parameters

- **experiment** (*Experiment*) – The `Experiment` to plot using this view.
- **title** (*str*) – Set the plot title
- **xlabel** (*str*) – Set the X axis label
- **ylabel** (*str*) – Set the Y axis label
- **huelabel** (*str*) – Set the label for the hue facet (in the legend)
- **legend** (*bool*) – Plot a legend for the color or hue facet? Defaults to `True`.
- **sharex** (*bool*) – If there are multiple subplots, should they share X axes? Defaults to `True`.
- **sharey** (*bool*) – If there are multiple subplots, should they share Y axes? Defaults to `True`.
- **row_order** (*list*) – Override the row facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **col_order** (*list*) – Override the column facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **hue_order** (*list*) – Override the hue facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **height** (*float*) – The height of *each row* in inches. Default = 3.0
- **aspect** (*float*) – The aspect ratio *of each subplot*. Default = 1.5

- **col_wrap** (*int*) – If `xfacet` is set and `yfacet` is not set, you can "wrap" the subplots around so that they form a multi-row grid by setting this to the number of columns you want.
- **sns_style** (*{"darkgrid", "whitegrid", "dark", "white", "ticks"}*) – Which `seaborn` style to apply to the plot? Default is `whitegrid`.
- **sns_context** (*{"paper", "notebook", "talk", "poster"}*) – Which `seaborn` context to use? Controls the scaling of plot elements such as tick labels and the legend. Default is `talk`.
- **palette** (*palette name, list, or dict*) – Colors to use for the different levels of the hue variable. Should be something that can be interpreted by `seaborn.color_palette`, or a dictionary mapping hue levels to matplotlib colors.
- **despine** (*Bool*) – Remove the top and right axes from the plot? Default is `True`.
- **min_quantile** (*float (>0.0 and <1.0, default = 0.001)*) – Clip data that is less than this quantile.
- **max_quantile** (*float (>0.0 and <1.0, default = 1.00)*) – Clip data that is greater than this quantile.
- **xlim** (*(float, float)*) – Set the range of the plot's X axis.
- **ylim** (*(float, float)*) – Set the range of the plot's Y axis.
- **shade** (*bool*) – Shade the interior of the isoplot? (default = `False`)
- **min_alpha, max_alpha** (*float*) – The minimum and maximum alpha blending values of the isolines, between 0 (transparent) and 1 (opaque).
- **n_levels** (*int*) – How many isolines to draw? (default = 10)
- **bw** (*str or float*) – The bandwidth for the gaussian kernel, controls how lumpy or smooth the kernel estimate is. Choices are:
- **gridsize** (*int*) – How many times to compute the kernel on each axis? (default: 100)

Notes

Other `kwargs` are passed to matplotlib.axes.Axes.contour

cytoflow.views.parallel_coords

A parallel-coordinates plot.

`ParallelCoordinatesView` – the *`IView`* class that makes the plot.

`class cytoflow.views.parallel_coords.`**`ParallelCoordinatesView`**

Bases: *`cytoflow.views.base_views.BaseNDView`*

Plots a parallel coordinates plot. PC plots are good for multivariate data; each vertical line represents one attribute, and one set of connected line segments represents one data point.

`channels`

The channels to view

Type List(Str)

`scale`

Re-scale the data in the specified channels before plotting. If a channel isn't specified, assume that the scale is linear.

Type Dict(Str : {“linear”, “logicle”, “log”})

subset
An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

xfacet
Set to one of the *Experiment.conditions* in the *Experiment*, and a new column of subplots will be added for every unique value of that condition.

Type String

yfacet
Set to one of the *Experiment.conditions* in the *Experiment*, and a new row of subplots will be added for every unique value of that condition.

Type String

huefacet
Set to one of the *Experiment.conditions* in the in the *Experiment*, and a new color will be added to the plot for every unique value of that condition.

Type String

huescale
How should the color scale for *huefacet* be scaled?

Type {‘linear’, ‘log’, ‘logicle’}

Examples

Make a little data set.

```
>>> import cytoflow as flow
>>> import_op = flow.ImportOp()
>>> import_op.tubes = [flow.Tube(file = "Plate01/RFP_Well_A3.fcs",
...                              conditions = {'Dox' : 10.0}),
...                    flow.Tube(file = "Plate01/CFP_Well_A4.fcs",
...                              conditions = {'Dox' : 1.0})]
>>> import_op.conditions = {'Dox' : 'float'}
>>> ex = import_op.apply()
```

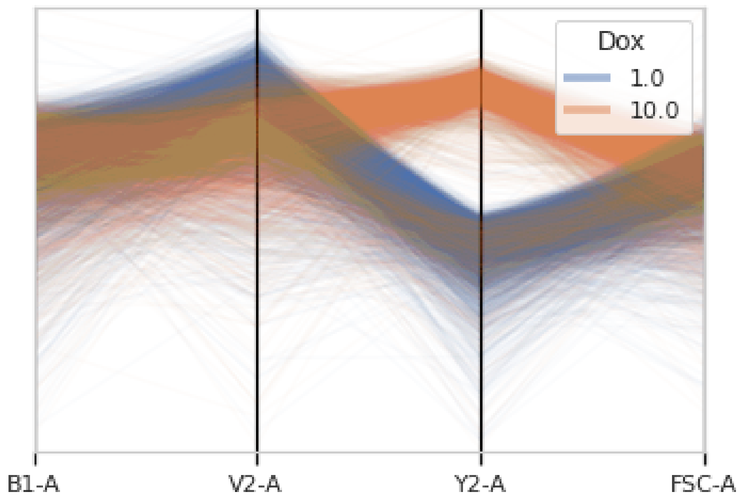
Plot the PC plot.

```
>>> flow.ParallelCoordinatesView(channels = ['B1-A', 'V2-A', 'Y2-A', 'FSC-A'],
...                              scale = {'Y2-A' : 'log',
...                                       'V2-A' : 'log',
...                                       'B1-A' : 'log',
...                                       'FSC-A' : 'log'},
...                              huefacet = 'Dox').plot(ex)
```

plot(*experiment*, ***kwargs*)
Plot a faceted parallel coordinates plot

Parameters

- **experiment** (*Experiment*) – The *Experiment* to plot using this view.



- **title** (*str*) – Set the plot title
- **xlabel** (*str*) – Set the X axis label
- **ylabel** (*str*) – Set the Y axis label
- **huelabel** (*str*) – Set the label for the hue facet (in the legend)
- **legend** (*bool*) – Plot a legend for the color or hue facet? Defaults to `True`.
- **sharex** (*bool*) – If there are multiple subplots, should they share X axes? Defaults to `True`.
- **sharey** (*bool*) – If there are multiple subplots, should they share Y axes? Defaults to `True`.
- **row_order** (*list*) – Override the row facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **col_order** (*list*) – Override the column facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **hue_order** (*list*) – Override the hue facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **height** (*float*) – The height of *each row* in inches. Default = 3.0
- **aspect** (*float*) – The aspect ratio *of each subplot*. Default = 1.5
- **col_wrap** (*int*) – If `xfacet` is set and `yfacet` is not set, you can "wrap" the subplots around so that they form a multi-row grid by setting this to the number of columns you want.
- **sns_style** (*{"darkgrid", "whitegrid", "dark", "white", "ticks"}*) – Which `seaborn` style to apply to the plot? Default is `whitegrid`.
- **sns_context** (*{"paper", "notebook", "talk", "poster"}*) – Which `seaborn` context to use? Controls the scaling of plot elements such as tick labels and the legend. Default is `talk`.
- **palette** (*palette name, list, or dict*) – Colors to use for the different levels of the hue variable. Should be something that can be interpreted by `seaborn.color_palette`, or a dictionary mapping hue levels to matplotlib colors.
- **despine** (*Bool*) – Remove the top and right axes from the plot? Default is `True`.

- **min_quantile** (*float (>0.0 and <1.0, default = 0.001)*) – Clip data that is less than this quantile.
- **max_quantile** (*float (>0.0 and <1.0, default = 1.00)*) – Clip data that is greater than this quantile.
- **lim** (*Dict(Str : (float, float))*) – Set the range of each channel's axis. If unspecified, assume that the limits are the minimum and maximum of the clipped data
- **alpha** (*float (default = 0.02)*) – The alpha blending value, between 0 (transparent) and 1 (opaque).
- **axvlines_kwds** (*dict*) – A dictionary of parameters to pass to ax.axvline

Notes

This uses a low-level API for speed; there are far fewer visual options that with other views.

cytoflow.views.radviz

A "Radviz" plot projects multivariate plots into two dimensions.

`RadvizView` – the `IView` class that makes the plot.

class `cytoflow.views.radviz.`**`RadvizView`**

Bases: `cytoflow.views.base_views.BaseNDView`

Plots a Radviz plot. Radviz plots project multivariate plots into two dimensions. Good for looking for clusters.

`channels`

The channels to view

Type List(Str)

`scale`

Re-scale the data in the specified channels before plotting. If a channel isn't specified, assume that the scale is linear.

Type Dict(Str : {"linear", "logicle", "log"})

`subset`

An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

`xfacet`

Set to one of the `Experiment.conditions` in the `Experiment`, and a new column of subplots will be added for every unique value of that condition.

Type String

`yfacet`

Set to one of the `Experiment.conditions` in the `Experiment`, and a new row of subplots will be added for every unique value of that condition.

Type String

`huefacet`

Set to one of the `Experiment.conditions` in the in the `Experiment`, and a new color will be added to the plot for every unique value of that condition.

Type String

huescale
How should the color scale for *huefacet* be scaled?

Type {'linear', 'log', 'logicle'}

Notes

The Radviz plot is based on a method of "dimensional anchors"[1]. The variables are conceived as points equidistant around a unit circle, and each data point connected to each anchor by a spring whose stiffness corresponds to the value of that data point. The location of the data point is the location where springs' tensions are minimized. Fortunately, there is fast matrix math to do this.

As per[2], the order of the anchors can make a huge difference. I've adapted the code from the R `radviz` package[3] to compute the cosine similarity of all possible circular permutations ("necklaces"). For a moderate number of anchors such as is likely to be encountered here, computing them all is completely feasible.

References

Examples

Make a little data set.

```
>>> import cytoflow as flow
>>> import_op = flow.ImportOp()
>>> import_op.tubes = [flow.Tube(file = "Plate01/RFP_Well_A3.fcs",
...                              conditions = {'Dox' : 10.0}),
...                    flow.Tube(file = "Plate01/CFP_Well_A4.fcs",
...                              conditions = {'Dox' : 1.0})]
>>> import_op.conditions = {'Dox' : 'float'}
>>> ex = import_op.apply()
```

Plot the radviz.

```
>>> flow.RadvizView(channels = ['B1-A', 'V2-A', 'Y2-A'],
...                 scale = {'Y2-A' : 'log',
...                          'V2-A' : 'log',
...                          'B1-A' : 'log'},
...                 huefacet = 'Dox').plot(ex)
```

plot(*experiment*, ***kwargs*)
Plot a faceted Radviz plot

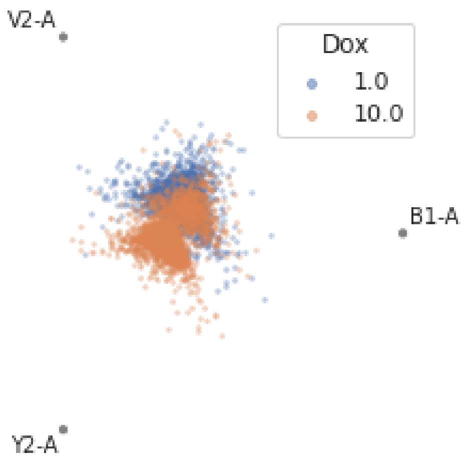
Parameters

- **experiment** (*Experiment*) – The *Experiment* to plot using this view.
- **title** (*str*) – Set the plot title
- **xlabel** (*str*) – Set the X axis label

[1] Hoffman P, Grinstein G, Pinkney D. Dimensional anchors: a graphic primitive for multidimensional multivariate information visualizations. Proceedings of the 1999 workshop on new paradigms in information visualization and manipulation in conjunction with the eighth ACM internation conference on Information and knowledge management. 1999 Nov 1 (pp. 9-16). ACM.

[2] Di Caro L, Frias-Martinez V, Frias-Martinez E. Analyzing the role of dimension arrangement for data visualization in radviz. Advances in Knowledge Discovery and Data Mining. 2010:125-32.

[3] https://github.com/yannabraham/Radviz



- **ylabel** (*str*) – Set the Y axis label
- **huelabel** (*str*) – Set the label for the hue facet (in the legend)
- **legend** (*bool*) – Plot a legend for the color or hue facet? Defaults to `True`.
- **sharex** (*bool*) – If there are multiple subplots, should they share X axes? Defaults to `True`.
- **sharey** (*bool*) – If there are multiple subplots, should they share Y axes? Defaults to `True`.
- **row_order** (*list*) – Override the row facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **col_order** (*list*) – Override the column facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **hue_order** (*list*) – Override the hue facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **height** (*float*) – The height of *each row* in inches. Default = 3.0
- **aspect** (*float*) – The aspect ratio *of each subplot*. Default = 1.5
- **col_wrap** (*int*) – If `xfacet` is set and `yfacet` is not set, you can "wrap" the subplots around so that they form a multi-row grid by setting this to the number of columns you want.
- **sns_style** (*{"darkgrid", "whitegrid", "dark", "white", "ticks"}*) – Which `seaborn` style to apply to the plot? Default is `whitegrid`.
- **sns_context** (*{"paper", "notebook", "talk", "poster"}*) – Which `seaborn` context to use? Controls the scaling of plot elements such as tick labels and the legend. Default is `talk`.
- **palette** (*palette name, list, or dict*) – Colors to use for the different levels of the hue variable. Should be something that can be interpreted by `seaborn.color_palette`, or a dictionary mapping hue levels to matplotlib colors.
- **despine** (*Bool*) – Remove the top and right axes from the plot? Default is `True`.
- **min_quantile** (*float (>0.0 and <1.0, default = 0.001)*) – Clip data that is less than this quantile.

- **max_quantile** (*float (>0.0 and <1.0, default = 1.00)*) – Clip data that is greater than this quantile.
- **lim** (*Dict(Str : (float, float))*) – Set the range of each channel's axis. If unspecified, assume that the limits are the minimum and maximum of the clipped data
- **alpha** (*float (default = 0.25)*) – The alpha blending value, between 0 (transparent) and 1 (opaque).
- **s** (*int (default = 2)*) – The size in points^2.
- **marker** (*a matplotlib marker style, usually a string*) – Specfies the glyph to draw for each point on the scatterplot. See matplotlib.markers for examples. Default: 'o'

Notes

Other `kwargs` are passed to matplotlib.pyplot.scatter

cytoflow.views.scatterplot

A 2-d scatterplot.

`ScatterplotView` – the *`IView`* class that makes the plot.

class `cytoflow.views.scatterplot.`**`ScatterplotView`**

Bases: *`cytoflow.views.base_views.Base2DView`*

Plots a 2-d scatterplot.

`xchannel`

The channel to view on the X axis

Type Str

`ychannel`

The channel to view on the Y axis

Type Str

`xscale`

The scales applied to the *`xchannel`* data before plotting it.

Type {'linear', 'log', 'logicle'} (default = 'linear')

`yscale`

The scales applied to the *`ychannel`* data before plotting it.

Type {'linear', 'log', 'logicle'} (default = 'linear')

`subset`

An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

`xfacet`

Set to one of the *`Experiment.conditions`* in the *`Experiment`*, and a new column of subplots will be added for every unique value of that condition.

Type String

yfacet
Set to one of the *Experiment.conditions* in the *Experiment*, and a new row of subplots will be added for every unique value of that condition.

Type String

huefacet
Set to one of the *Experiment.conditions* in the in the *Experiment*, and a new color will be added to the plot for every unique value of that condition.

Type String

huescale
How should the color scale for *huefacet* be scaled?

Type {'linear', 'log', 'logicle'}

Examples

Make a little data set.

```
>>> import cytoflow as flow
>>> import_op = flow.ImportOp()
>>> import_op.tubes = [flow.Tube(file = "Plate01/RFP_Well_A3.fcs",
...                              conditions = {'Dox' : 10.0}),
...                    flow.Tube(file = "Plate01/CFP_Well_A4.fcs",
...                              conditions = {'Dox' : 1.0})]
>>> import_op.conditions = {'Dox' : 'float'}
>>> ex = import_op.apply()
```

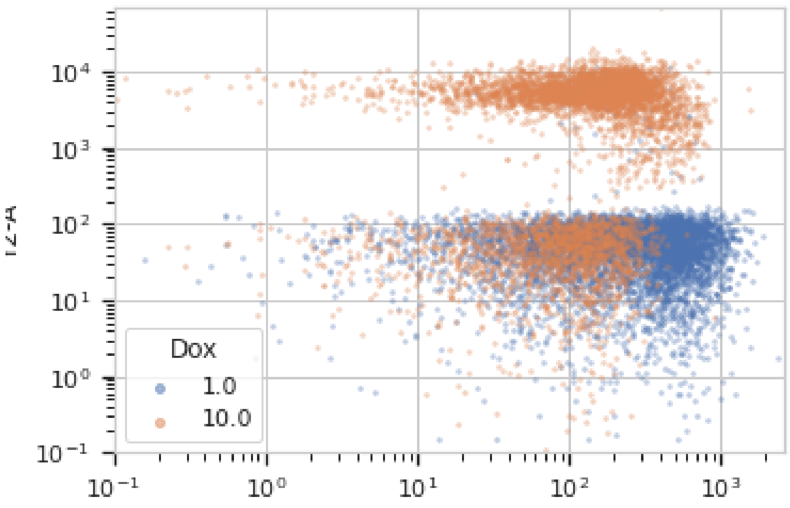
Plot a density plot

```
>>> flow.ScatterplotView(xchannel = 'V2-A',
...                      xscale = 'log',
...                      ychannel = 'Y2-A',
...                      yscale = 'log',
...                      huefacet = 'Dox').plot(ex)
```

plot(*experiment*, ***kwargs*)
Plot a faceted scatter plot view of a channel

Parameters

- **experiment** (*Experiment*) – The *Experiment* to plot using this view.
- **title** (*str*) – Set the plot title
- **xlabel** (*str*) – Set the X axis label
- **ylabel** (*str*) – Set the Y axis label
- **huelabel** (*str*) – Set the label for the hue facet (in the legend)
- **legend** (*bool*) – Plot a legend for the color or hue facet? Defaults to `True`.
- **sharex** (*bool*) – If there are multiple subplots, should they share X axes? Defaults to `True`.
- **sharey** (*bool*) – If there are multiple subplots, should they share Y axes? Defaults to `True`.
- **row_order** (*list*) – Override the row facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.



- **col_order** (*list*) – Override the column facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **hue_order** (*list*) – Override the hue facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **height** (*float*) – The height of *each row* in inches. Default = 3.0
- **aspect** (*float*) – The aspect ratio *of each subplot*. Default = 1.5
- **col_wrap** (*int*) – If `xfacet` is set and `yfacet` is not set, you can "wrap" the subplots around so that they form a multi-row grid by setting this to the number of columns you want.
- **sns_style** (*{"darkgrid", "whitegrid", "dark", "white", "ticks"}*) – Which `seaborn` style to apply to the plot? Default is `whitegrid`.
- **sns_context** (*{"paper", "notebook", "talk", "poster"}*) – Which `seaborn` context to use? Controls the scaling of plot elements such as tick labels and the legend. Default is `talk`.
- **palette** (*palette name, list, or dict*) – Colors to use for the different levels of the hue variable. Should be something that can be interpreted by `seaborn.color_palette`, or a dictionary mapping hue levels to matplotlib colors.
- **despine** (*Bool*) – Remove the top and right axes from the plot? Default is `True`.
- **min_quantile** (*float (>0.0 and <1.0, default = 0.001)*) – Clip data that is less than this quantile.
- **max_quantile** (*float (>0.0 and <1.0, default = 1.00)*) – Clip data that is greater than this quantile.
- **xlim** (*(float, float)*) – Set the range of the plot's X axis.
- **ylim** (*(float, float)*) – Set the range of the plot's Y axis.
- **alpha** (*float (default = 0.25)*) – The alpha blending value, between 0 (transparent) and 1 (opaque).
- **s** (*int (default = 2)*) – The size in points^2.

- **marker** (*a matplotlib marker style, usually a string*) – Specfies the glyph to draw for each point on the scatterplot. See matplotlib.markers for examples. Default: 'o'

Notes

Other `kwargs` are passed to matplotlib.pyplot.scatter

cytoflow.views.stats_1d

Plot a statistic with a numeric variable on the X axis.

`Stats1DView` – the *`IView`* class that makes the plot.

class `cytoflow.views.stats_1d.`**`Stats1DView`**

Bases: *`cytoflow.views.base_views.Base1DStatisticsView`*

Plot a statistic. The value of the statistic will be plotted on the Y axis; a numeric conditioning variable must be chosen for the X axis. Every variable in the statistic must be specified as either the *`variable`* or one of the plot facets.

`variable_scale`

The scale applied to the variable (on the X axis)

Type {'linear', 'log', 'logicle'}

`statistic`

The name of the statistic to plot. Must be a key in the *`Experiment.statistics`* attribute of the *`Experiment`* being plotted.

Type (str, str)

`error_statistic`

The name of the statistic used to plot error bars. Must be a key in the *`Experiment.statistics`* attribute of the *`Experiment`* being plotted.

Type (str, str)

`scale`

The scale applied to the data before plotting it.

Type {'linear', 'log', 'logicle'}

`variable`

The condition that varies when plotting this statistic: used for the x axis of line plots, the bar groups in bar plots, etc.

Type str

`subset`

An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

`xfacet`

Set to one of the *`Experiment.conditions`* in the *`Experiment`*, and a new column of subplots will be added for every unique value of that condition.

Type String

yfacet
Set to one of the *Experiment.conditions* in the *Experiment*, and a new row of subplots will be added for every unique value of that condition.

Type String

huefacet
Set to one of the *Experiment.conditions* in the in the *Experiment*, and a new color will be added to the plot for every unique value of that condition.

Type String

huescale
How should the color scale for *huefacet* be scaled?

Type {'linear', 'log', 'logicle'}

Examples

Make a little data set.

```
>>> import cytoflow as flow
>>> import_op = flow.ImportOp()
>>> import_op.tubes = [flow.Tube(file = "Plate01/RFP_Well_A3.fcs",
...                              conditions = {'Dox' : 10.0}),
...                    flow.Tube(file = "Plate01/CFP_Well_A4.fcs",
...                              conditions = {'Dox' : 1.0})]
>>> import_op.conditions = {'Dox' : 'float'}
>>> ex = import_op.apply()
```

Create and a new statistic.

```
>>> ch_op = flow.ChannelStatisticOp(name = 'MeanByDox',
...                     channel = 'Y2-A',
...                     function = flow.geom_mean,
...                     by = ['Dox'])
>>> ex2 = ch_op.apply(ex)
```

View the new statistic

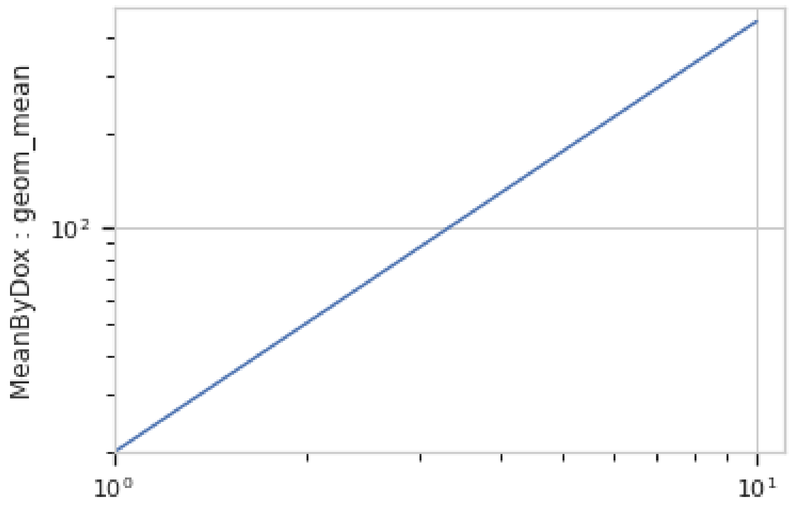
```
>>> flow.Stats1DView(variable = 'Dox',
...                   statistic = ('MeanByDox', 'geom_mean'),
...                   variable_scale = 'log',
...                   scale = 'log').plot(ex2)
```

enum_plots(*experiment*)
Returns an iterator over the possible plots that this View can produce. The values returned can be passed to *plot*.

plot(*experiment*, *plot_name=None*, ***kwargs*)
Plot a chart of a variable's values against a statistic.

Parameters

- **experiment** (*Experiment*) – The *Experiment* to plot using this view.
- **title** (*str*) – Set the plot title
- **xlabel** (*str*) – Set the X axis label



- **ylabel** (*str*) – Set the Y axis label
- **huelabel** (*str*) – Set the label for the hue facet (in the legend)
- **legend** (*bool*) – Plot a legend for the color or hue facet? Defaults to `True`.
- **sharex** (*bool*) – If there are multiple subplots, should they share X axes? Defaults to `True`.
- **sharey** (*bool*) – If there are multiple subplots, should they share Y axes? Defaults to `True`.
- **row_order** (*list*) – Override the row facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **col_order** (*list*) – Override the column facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **hue_order** (*list*) – Override the hue facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **height** (*float*) – The height of *each row* in inches. Default = 3.0
- **aspect** (*float*) – The aspect ratio *of each subplot*. Default = 1.5
- **col_wrap** (*int*) – If `xfacet` is set and `yfacet` is not set, you can "wrap" the subplots around so that they form a multi-row grid by setting this to the number of columns you want.
- **sns_style** (*{"darkgrid", "whitegrid", "dark", "white", "ticks"}*) – Which `seaborn` style to apply to the plot? Default is `whitegrid`.
- **sns_context** (*{"paper", "notebook", "talk", "poster"}*) – Which `seaborn` context to use? Controls the scaling of plot elements such as tick labels and the legend. Default is `talk`.
- **palette** (*palette name, list, or dict*) – Colors to use for the different levels of the hue variable. Should be something that can be interpreted by `seaborn.color_palette`, or a dictionary mapping hue levels to matplotlib colors.
- **despine** (*Bool*) – Remove the top and right axes from the plot? Default is `True`.
- **plot_name** (*str*) – If this `IView` can make multiple plots, `plot_name` is the name of the plot to make. Must be one of the values retrieved from `enum_plots`.

- **orientation** (*{'vertical', 'horizontal'}*)
- **lim** (*(float, float)*) – Set the range of the plot's axis.
- **variable_lim** (*(float, float)*) – The limits on the variable axis
- **color** (*a matplotlib color*) – The color to plot with. Overridden if `huefacet` is not `None`
- **linewidth** (*float*) – The width of the line, in points
- **linestyle** (*['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | '-' | '–' | '-.' | ':' | 'None' | ' ' | '']*)
- **marker** (*a matplotlib marker style*) – See http://matplotlib.org/api/markers_api.html#module-matplotlib.markers
- **markersize** (*int*) – The marker size in points
- **markerfacecolor** (*a matplotlib color*) – The color to make the markers. Overridden (?) if `huefacet` is not `None`
- **alpha** (*the alpha blending value, from 0.0 (transparent) to 1.0 (opaque)*)
- **capsize** (*scalar*) – The size of the error bar caps, in points
- **shade_error** (*bool*) – If `False` (the default), plot the error statistic as traditional "error bars." If `True`, plot error statistic as a filled, shaded region.
- **shade_alpha** (*float*) – The transparency of the shaded error region, from 0.0 (transparent) to 1.0 (opaque.) Default is 0.2.

Notes

Other `kwargs` are passed to `matplotlib.pyplot.plot`

cytoflow.views.stats_2d

Plots two statistics on a scatter plot.

`Stats2DView` – the `IView` class that makes the plot.

class `cytoflow.views.stats_2d.`**`Stats2DView`**

Bases: `cytoflow.views.base_views.Base2DStatisticsView`

Plot two statistics on a scatter plot. A point (X,Y) is drawn for every pair of elements with the same value of `variable`; the X value is from `xstatistic` and the Y value is from `ystatistic`.

`xstatistic`

The name of the statistic to plot on the X axis. Must be a key in the `Experiment.statistics` attribute of the `Experiment` being plotted.

Type (str, str)

`ystatistic`

The name of the statistic to plot on the Y axis. Must be a key in the `Experiment.statistics` attribute of the `Experiment` being plotted.

Type (str, str)

`x_error_statistic`

The name of the statistic used to plot error bars on the X axis. Must be a key in the `Experiment.statistics` attribute of the `Experiment` being plotted.

Type (str, str)

`y_error_statistic`
The name of the statistic used to plot error bars on the Y axis. Must be a key in the `Experiment.statistics` attribute of the `Experiment` being plotted.

Type (str, str)

`xscale`
The scale applied to `xstatistic` before plotting it.

Type {'linear', 'log', 'logicle'}

`yscale`
The scale applied to `ystatistic` before plotting it.

Type {'linear', 'log', 'logicle'}

`variable`
The condition that varies when plotting this statistic: used for the x axis of line plots, the bar groups in bar plots, etc.

Type str

`subset`
An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

`xfacet`
Set to one of the `Experiment.conditions` in the `Experiment`, and a new column of subplots will be added for every unique value of that condition.

Type String

`yfacet`
Set to one of the `Experiment.conditions` in the `Experiment`, and a new row of subplots will be added for every unique value of that condition.

Type String

`huefacet`
Set to one of the `Experiment.conditions` in the in the `Experiment`, and a new color will be added to the plot for every unique value of that condition.

Type String

`huescale`
How should the color scale for `huefacet` be scaled?

Type {'linear', 'log', 'logicle'}

Examples

Make a little data set.

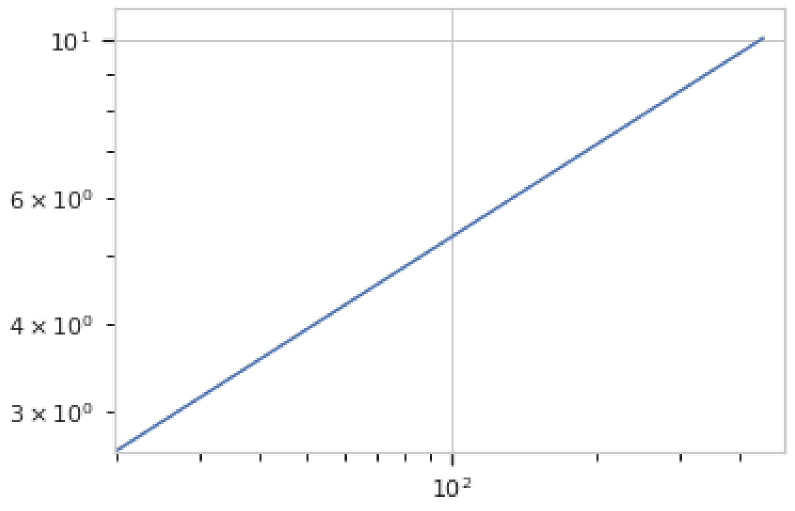
```
>>> import cytoflow as flow
>>> import_op = flow.ImportOp()
>>> import_op.tubes = [flow.Tube(file = "Plate01/RFP_Well_A3.fcs",
...                              conditions = {'Dox' : 10.0}),
...                    flow.Tube(file = "Plate01/CFP_Well_A4.fcs",
...                              conditions = {'Dox' : 1.0})]
>>> import_op.conditions = {'Dox' : 'float'}
>>> ex = import_op.apply()
```

Create two new statistics

```
>>> ch_op = flow.ChannelStatisticOp(name = 'MeanByDox',
...                     channel = 'Y2-A',
...                     function = flow.geom_mean,
...                     by = ['Dox'])
>>> ex2 = ch_op.apply(ex)
>>> ch_op_2 = flow.ChannelStatisticOp(name = 'SdByDox',
...                       channel = 'Y2-A',
...                       function = flow.geom_sd,
...                       by = ['Dox'])
>>> ex3 = ch_op_2.apply(ex2)
```

Plot the statistics

```
>>> flow.Stats2DView(variable = 'Dox',
...                  xstatistic = ('MeanByDox', 'geom_mean'),
...                  xscale = 'log',
...                  ystatistic = ('SdByDox', 'geom_sd'),
...                  yscale = 'log').plot(ex3)
```



`enum_plots`(*experiment*)
Returns an iterator over the possible plots that this View can produce. The values returned can be passed to `plot`.

`plot`(*experiment*, *plot_name=None*, ***kwargs*)
Plot a chart of two statistics' values as a common variable changes.

Parameters

- **experiment** (*Experiment*) – The `Experiment` to plot using this view.
- **title** (*str*) – Set the plot title
- **xlabel** (*str*) – Set the X axis label
- **ylabel** (*str*) – Set the Y axis label
- **huelabel** (*str*) – Set the label for the hue facet (in the legend)
- **legend** (*bool*) – Plot a legend for the color or hue facet? Defaults to `True`.
- **sharex** (*bool*) – If there are multiple subplots, should they share X axes? Defaults to `True`.
- **sharey** (*bool*) – If there are multiple subplots, should they share Y axes? Defaults to `True`.
- **row_order** (*list*) – Override the row facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **col_order** (*list*) – Override the column facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **hue_order** (*list*) – Override the hue facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **height** (*float*) – The height of *each row* in inches. Default = 3.0
- **aspect** (*float*) – The aspect ratio *of each subplot*. Default = 1.5
- **col_wrap** (*int*) – If `xfacet` is set and `yfacet` is not set, you can "wrap" the subplots around so that they form a multi-row grid by setting this to the number of columns you want.
- **sns_style** (*{"darkgrid", "whitegrid", "dark", "white", "ticks"}*) – Which `seaborn` style to apply to the plot? Default is `whitegrid`.
- **sns_context** (*{"paper", "notebook", "talk", "poster"}*) – Which `seaborn` context to use? Controls the scaling of plot elements such as tick labels and the legend. Default is `talk`.
- **palette** (*palette name, list, or dict*) – Colors to use for the different levels of the hue variable. Should be something that can be interpreted by `seaborn.color_palette`, or a dictionary mapping hue levels to matplotlib colors.
- **despine** (*Bool*) – Remove the top and right axes from the plot? Default is `True`.
- **plot_name** (*str*) – If this `IView` can make multiple plots, `plot_name` is the name of the plot to make. Must be one of the values retrieved from `enum_plots`.
- **xlim** (*(float, float)*) – Set the range of the plot's X axis.
- **ylim** (*(float, float)*) – Set the range of the plot's Y axis.
- **color** (*a matplotlib color*) – The color to plot with. Overridden if `huefacet` is not `None`
- **linestyle** (*{'solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | '-' | '–' | '-.' | ':' | 'None' | ' ' | ''}*)

- **marker** (*a matplotlib marker style*) – See http://matplotlib.org/api/markers_api.html#module-matplotlib.markers
- **markersize** (*int*) – The marker size in points
- **markerfacecolor** (*a matplotlib color*) – The color to make the markers. Overridden (?) if `huefacet` is not `None`
- **alpha** (*the alpha blending value, from 0.0 (transparent) to 1.0 (opaque)*)

Notes

Other `kwargs` are passed to matplotlib.pyplot.plot

cytoflow.views.table

"Plot" a tabular view of a statistic.

`TableView` – the `IView` class that makes the plot.

class `cytoflow.views.table.`**`TableView`**

Bases: `traits.has_traits.HasStrictTraits`

"Plot" a tabular view of a statistic. Mostly useful for GUIs. Each level of the statistic's index must be used in `row_facet`, `column_facet`, `subrow_facet`, or `subcolumn_facet`. This module can't "plot" a statistic with more than four index levels unless `subset` is set and that results in extra levels being dropped.

`statistic`

The name of the statistic to plot. Must be a key in the `Experiment.statistics` attribute of the `Experiment` being plotted. Each level of the statistic's index must be used in `row_facet`, `column_facet`, `subrow_facet`, or `subcolumn_facet`.

Type (str, str)

`row_facet`

The statistic facet to be used as row headers.

Type str

`column_facet`

The statistic facet to be used as column headers.

Type str

`subrow_facet`

The statistic facet to be used as subrow headers.

Type str

`subcolumn_facet`

The statistic facet to be used as subcolumn headers.

Type str

`subset`

A Python expression used to select a subset of the statistic to plot.

Type str

Examples

Make a little data set.

```
>>> import cytoflow as flow
>>> import_op = flow.ImportOp()
>>> import_op.tubes = [flow.Tube(file = "Plate01/RFP_Well_A3.fcs",
...                               conditions = {'Dox' : 10.0}),
...                     flow.Tube(file = "Plate01/CFP_Well_A4.fcs",
...                               conditions = {'Dox' : 1.0})]
>>> import_op.conditions = {'Dox' : 'float'}
>>> ex = import_op.apply()
```

Add a threshold gate

```
>>> ex2 = flow.ThresholdOp(name = 'Threshold',
...                         channel = 'Y2-A',
...                         threshold = 2000).apply(ex)
```

Add a statistic

```
>>> ex3 = flow.ChannelStatisticOp(name = "ByDox",
...                                channel = "Y2-A",
...                                by = ['Dox', 'Threshold'],
...                                function = len).apply(ex2)
```

“Plot” the table

```
>>> flow.TableView(statistic = ("ByDox", "len"),
...                row_facet = "Dox",
...                column_facet = "Threshold").plot(ex3)
```

plot(*experiment*, *plot_name=None*, ***kwargs*)
Plot a table

export(*experiment*, *filename*)
Export the table to a file.

cytoflow.views.violin

A violin plot is a facetted set of kernel density estimates.

ViolinPlotView – the *IView* class that makes the plot.

class cytoflow.views.violin.**ViolinPlotView**
Bases: *cytoflow.views.base_views.Base1DView*

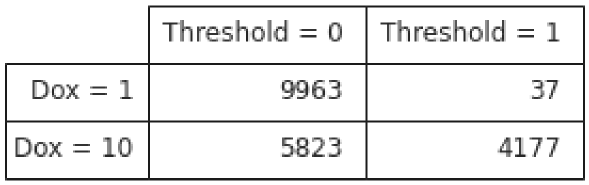
Plots a violin plot – a facetted set of kernel density estimates.

variable
the main variable by which we’re faceting

Type Str

channel
The channel to view

Type Str



	Threshold = 0	Threshold = 1
Dox = 1	9963	37
Dox = 10	5823	4177

scale
The scale applied to the data before plotting it.

Type {'linear', 'log', 'logicle'}

subset
An expression that specifies the subset of the statistic to plot. Passed unmodified to `pandas.DataFrame.query`.

Type str

xfacet
Set to one of the *Experiment.conditions* in the *Experiment*, and a new column of subplots will be added for every unique value of that condition.

Type String

yfacet
Set to one of the *Experiment.conditions* in the *Experiment*, and a new row of subplots will be added for every unique value of that condition.

Type String

huefacet
Set to one of the *Experiment.conditions* in the in the *Experiment*, and a new color will be added to the plot for every unique value of that condition.

Type String

huescale
How should the color scale for *huefacet* be scaled?

Type {'linear', 'log', 'logicle'}

Examples

Make a little data set.

```
>>> import cytoflow as flow
>>> import_op = flow.ImportOp()
>>> import_op.tubes = [flow.Tube(file = "Plate01/RFP_Well_A3.fcs",
...                               conditions = {'Dox' : 10.0}),
...                     flow.Tube(file = "Plate01/CFP_Well_A4.fcs",
...                               conditions = {'Dox' : 1.0})]
>>> import_op.conditions = {'Dox' : 'float'}
>>> ex = import_op.apply()
```

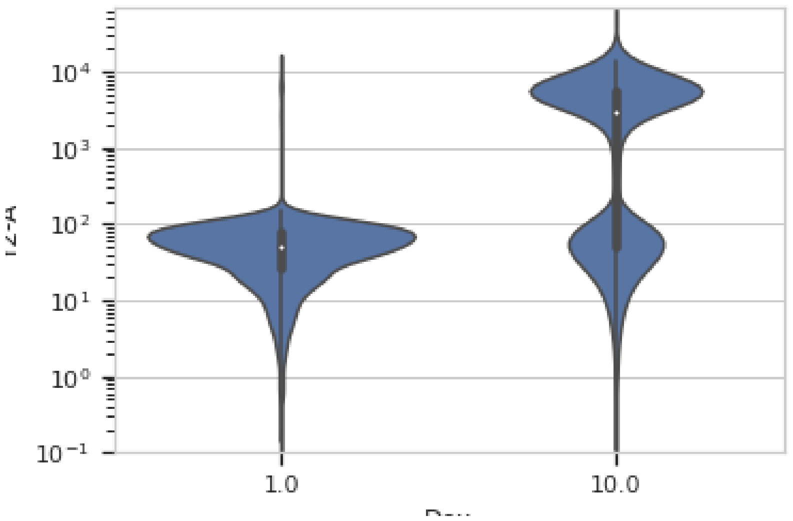
Plot a violin plot

```
>>> flow.ViolinPlotView(channel = 'Y2-A',
...                      scale = 'log',
...                      variable = 'Dox').plot(ex)
```

plot(*experiment*, ***kwargs*)
Plot a violin plot of a variable

Parameters

- **experiment** (*Experiment*) – The *Experiment* to plot using this view.
- **title** (*str*) – Set the plot title



- **xlabel** (*str*) – Set the X axis label
- **ylabel** (*str*) – Set the Y axis label
- **huelabel** (*str*) – Set the label for the hue facet (in the legend)
- **legend** (*bool*) – Plot a legend for the color or hue facet? Defaults to `True`.
- **sharex** (*bool*) – If there are multiple subplots, should they share X axes? Defaults to `True`.
- **sharey** (*bool*) – If there are multiple subplots, should they share Y axes? Defaults to `True`.
- **row_order** (*list*) – Override the row facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **col_order** (*list*) – Override the column facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **hue_order** (*list*) – Override the hue facet value order with the given list. If a value is not given in the ordering, it is not plotted. Defaults to a "natural ordering" of all the values.
- **height** (*float*) – The height of *each row* in inches. Default = 3.0
- **aspect** (*float*) – The aspect ratio *of each subplot*. Default = 1.5
- **col_wrap** (*int*) – If `xfacet` is set and `yfacet` is not set, you can "wrap" the subplots around so that they form a multi-row grid by setting this to the number of columns you want.
- **sns_style** (*{"darkgrid", "whitegrid", "dark", "white", "ticks"}*) – Which `seaborn` style to apply to the plot? Default is `whitegrid`.
- **sns_context** (*{"paper", "notebook", "talk", "poster"}*) – Which `seaborn` context to use? Controls the scaling of plot elements such as tick labels and the legend. Default is `talk`.
- **palette** (*palette name, list, or dict*) – Colors to use for the different levels of the hue variable. Should be something that can be interpreted by `seaborn.color_palette`, or a dictionary mapping hue levels to matplotlib colors.
- **despine** (*Bool*) – Remove the top and right axes from the plot? Default is `True`.

- **min_quantile** (*float (>0.0 and <1.0, default = 0.001)*) – Clip data that is less than this quantile.
- **max_quantile** (*float (>0.0 and <1.0, default = 1.00)*) – Clip data that is greater than this quantile.
- **lim** (*(float, float)*) – Set the range of the plot's data axis.
- **orientation** (*{'vertical', 'horizontal'}*)
- **bw** (*{{'scott', 'silverman', float}}, optional*) – Either the name of a reference rule or the scale factor to use when computing the kernel bandwidth. The actual kernel size will be determined by multiplying the scale factor by the standard deviation of the data within each bin.
- **scale_plot** (*{{"area", "count", "width"}}, optional*) – The method used to scale the width of each violin. If `area`, each violin will have the same area. If `count`, the width of the violins will be scaled by the number of observations in that bin. If `width`, each violin will have the same width.
- **scale_hue** (*bool, optional*) – When nesting violins using a `hue` variable, this parameter determines whether the scaling is computed within each level of the major grouping variable (`scale_hue=True`) or across all the violins on the plot (`scale_hue=False`).
- **gridsize** (*int, optional*) – Number of points in the discrete grid used to compute the kernel density estimate.
- **inner** (*{{"box", "quartile", None}}, optional*) – Representation of the datapoints in the violin interior. If `box`, draw a miniature boxplot. If `quartiles`, draw the quartiles of the distribution. If `point` or `stick`, show each underlying datapoint. Using `None` will draw unadorned violins.
- **split** (*bool, optional*) – When using hue nesting with a variable that takes two levels, setting `split` to True will draw half of a violin for each level. This can make it easier to directly compare the distributions.

Submodules

cytoflow.experiment

Defines `Experiment`, `cytoflow`'s main data structure.

`Experiment` – manages the data and metadata for a flow experiment.

`class cytoflow.experiment.`**`Experiment`**

Bases: `traits.has_traits.HasStrictTraits`

An Experiment manages all the data and metadata for a flow experiment.

An `Experiment` is the central data structure in `cytoflow`: it wraps a `pandas.DataFrame` containing all the data from a flow experiment. Each row in the table is an event. Each column is either a measurement from one of the detectors (or a "derived" measurement such as a transformed value or a ratio), or a piece of metadata associated with that event: which tube it came from, what the experimental conditions for that tube were, gate membership, etc. The `Experiment` object lets you:

- Add additional metadata to define subpopulations
- Get events that match a particular metadata signature.

Additionally, the `Experiment` object manages channel- and experiment-level metadata in the `metadata` attribute, which is a dictionary. This allows the rest of the modules in `cytoflow` to track and enforce other constraints that are important in doing quantitative flow cytometry: for example, every tube must be collected with the same channel parameters (such as PMT voltage.)

Note: `Experiment` is not responsible for enforcing the constraints; `ImportOp` and the other modules are.

`data`

All the events and metadata represented by this experiment. Each event is a row; each column is either a measured channel (eg. a fluorescence measurement), a derived channel (eg. the ratio between two channels), or a piece of metadata. Metadata can be either experimental conditions (eg. induction level, timepoint) or added by operations (eg. gate membership).

Type pandas.DataFrame

`metadata`

Each column in `data` has an entry in `metadata` whose key is the column name and whose value is a dict of column-specific metadata. Metadata is added by operations, and is occasionally useful if modules are expected to work together. See individual operations' documentation for a list of the metadata that operation adds. The only "required" metadata is `type`, which can be `channel` (if the column is a measured channel, or derived from one) or `condition` (if the column is an experimental condition, gate membership, etc.)

Warning: There may also be experiment-wide entries in `metadata` that are *not* columns in `data`!

Type Dict(Str : Dict(Str : Any)

`history`

The `IOperation` operations that have been applied to the raw data to result in this `Experiment`.

Type List(*IOperation*)

`statistics`

The statistics and parameters computed by models that were fit to the data. The key is an `(Str, Str)` tuple, where the first `Str` is the name of the operation that supplied the statistic, and the second `Str` is the name of the statistic. The value is a multi-indexed `pandas.Series`: each level of the index is a facet, and each combination of indices is a subset for which the statistic was computed. The values of the series, of course, are the values of the computed parameters or statistics for each subset.

Type Dict((Str, Str) : pandas.Series)

`channels`

The channels that this experiment tracks (read-only).

Type List(String)

`conditions`

The experimental conditions and analysis groups (gate membership, etc) that this experiment tracks. The key is the name of the condition, and the value is a `pandas.Series` with that condition's possible values.

Type Dict(String : pandas.Series)

Notes

The OOP programmer in me desperately wanted to subclass `pandas.DataFrame`, add some flow-specific stuff, and move on with my life. (I may still, with something like https://github.com/dalejung/pandas-composition). A few things get in the way of directly subclassing `pandas.DataFrame`:

- First, to enable some of the delicious syntactic sugar for accessing its contents, `pandas.DataFrame` redefines `__getattribute__` and `__setattribute__`, and making it recognize (and maintain across copies) additional attributes is an unsupported (non-public) API feature and introduces other subclassing weirdness.
- Second, many of the operations (like appending!) don't happen in-place; they return copies instead. It's cleaner to simply manage that copying ourselves instead of making the client deal with it. We can pretend to operate on the data in-place.

To maintain the ease of use, we'll override `__getitem__` and pass it to the wrapped `pandas.DataFrame`. We'll do the same with some of the more useful `pandas.DataFrame` API pieces (like *`query`*); and of course, you can just get the data frame itself with *`Experiment.data`*.

Examples

```
>>> import cytoflow as flow
>>> tube1 = flow.Tube(file = 'cytoflow/tests/data/Plate01/RFP_Well_A3.fcs',
...                   conditions = {"Dox" : 10.0})
>>> tube2 = flow.Tube(file='cytoflow/tests/data/Plate01/CFP_Well_A4.fcs',
...                   conditions = {"Dox" : 1.0})
>>>
>>> import_op = flow.ImportOp(conditions = {"Dox" : "float"},
...                           tubes = [tube1, tube2])
>>>
>>> ex = import_op.apply()
>>> ex.data.shape
(20000, 17)
>>> ex.data.groupby(['Dox']).size()
Dox
1      10000
10     10000
dtype: int64
```

subset(*conditions*, *values*)

Returns a subset of this experiment including only the events where each condition in `condition` equals the corresponding value in `values`.

Parameters

- **conditions** (*Str or List(Str)*) – A condition or list of conditions
- **values** (*Any or Tuple(Any)*) – The value(s) of the condition(s)

Returns A new *`Experiment`* containing only the events specified in `conditions` and `values`.

Return type *Experiment*

Note: This is a wrapper around `pandas.DataFrame.groupby` and `pandas.core.groupby.GroupBy.get_group`. That means you can pass other things in *`conditions`* – see the `pandas.DataFrame.groupby` documentation for details.

`query`(*expr*, ***kwargs*)

Return an experiment whose data is a subset of this one where `expr` evaluates to `True`.

This method "sanitizes" column names first, replacing characters that are not valid in a Python identifier with an underscore `_`. So, the column name `a column` becomes `a_column`, and can be queried with an `a_column == True` or such.

Parameters

- **expr** (*string*) – The expression to pass to `pandas.DataFrame.query`. Must be a valid Python expression, something you could pass to `eval`.
- ****kwargs** (*dict*) – Other named parameters to pass to `pandas.DataFrame.query`.

Returns A new `Experiment`, a clone of this one with the data returned by `pandas.DataFrame.query`

Return type *Experiment*

`clone`(*deep=True*)

Create a copy of this `Experiment`. `metadata`, `statistics` and `history` are deep copies; whether or not `data` is a deep copy depends on the value of the `deep` parameter.

Warning: The intent is that `deep` is set to `False` by operations that are only adding columns to the underlying `pandas.DataFrame`. This will improve memory performance. However, the resulting `Experiment` **CANNOT BE MODIFIED IN-PLACE**, because doing so will affect the other `Experiment` s that are clones of the one being modified.

`add_condition`(*name*, *dtype*, *data=None*)

Add a new column of per-event metadata to this `Experiment`.

Note: `add_condition` operates **in place.**

There are two places to call `add_condition`.

- As you're setting up a new `Experiment`, call `add_condition` with `data` set to `None` to specify the conditions the new events will have.
- If you compute some new per-event metadata on an existing `Experiment`, call `add_condition` to add it.

Parameters

- **name** (*String*) – The name of the new column in `data`. Must be a valid Python identifier: must start with `[A-Za-z_]` and contain only the characters `[A-Za-z0-9_]`.
- **dtype** (*String*) – The type of the new column in `data`. Must be a string that `pandas.Series` recognizes as a `dtype`: common types are `category`, `float`, `int`, and `bool`.
- **data** (*pandas.Series (default = None)*) – The `pandas.Series` to add to `data`. Must be the same length as `data`, and it must be convertable to a `pandas.Series` of type `dtype`. If `None`, will add an empty column to the `Experiment` … but the `Experiment` must be empty to do so!

Raises `.CytoflowError` – If the `pandas.Series` passed in `data` isn't the same length as `data`, or isn't convertable to type `dtype`.

Examples

```
>>> import cytoflow as flow
>>> ex = flow.Experiment()
>>> ex.add_condition("Time", "float")
>>> ex.add_condition("Strain", "category")
```

add_channel(*name*, *data=None*)

Add a new column of per-event data (as opposed to metadata) to this *Experiment*: ie, something that was measured per cell, or derived from per-cell measurements.

Note: *add_channel* operates *in place*.

Parameters

- **name** (*String*) – The name of the new column to be added to *data*.
- **data** (*pandas.Series*) – The pandas.Series to add to *data*. Must be the same length as *data*, and it must be convertable to a dtype of float64. If None, will add an empty column to the *Experiment* … but the *Experiment* must be empty to do so!

Raises **.CytoflowError** – If the pandas.Series passed in data isn't the same length as *data*, or isn't convertable to a dtype float64.

Examples

```
>>> ex.add_channel("FSC_over_2", ex.data["FSC-A"] / 2.0)
```

add_events(*data*, *conditions*)

Add new events to this *Experiment*. *add_events* operates **in place**, modifying the *Experiment* object that it's called on.

Each new event in data is appended to *data*, and its per-event metadata columns will be set with the values specified in conditions. Thus, it is particularly useful for adding tubes of data to new experiments, before additional per-event metadata is added by gates, etc.

Note: *Every* column in *data* must be accounted for. Each column of type channel must appear in data; each column of metadata must have a key:value pair in conditions.

Parameters

- **tube** (*pandas.DataFrame*) – A single tube or well's worth of data. Must be a DataFrame with the same columns as *channels*
- **conditions** (*Dict(Str, Any)*) – A dictionary of the tube's metadata. The keys must match *conditions*, and the values must be coercable to the relevant numpy dtype.

Raises **.CytoflowError** – *add_events* pukes if: * there are columns in data that aren't channels in the experiment, or vice versa. * there are keys in conditions that aren't conditions in the experiment, or vice versa. * there is metadata specified in conditions that can't be converted to the corresponding metadata dtype.

Examples

```
>>> import cytoflow as flow
>>> from fcsparser import fcsparser
>>> ex = flow.Experiment()
>>> ex.add_condition("Time", "float")
>>> ex.add_condition("Strain", "category")
>>> tube1, _ = fcparser.parse('CFP_Well_A4.fcs')
>>> tube2, _ = fcparser.parse('RFP_Well_A3.fcs')
>>> ex.add_events(tube1, {"Time" : 1, "Strain" : "BL21"})
>>> ex.add_events(tube2, {"Time" : 1, "Strain" : "Top10G"})
```

cytoflowgui package

cytoflowgui

The *cytoflowgui* package contains the Qt GUI for *cytoflow*.

Subpackages

cytoflowgui.editors package

cytoflowgui.editors

Custom Qt editors

Submodules

cytoflowgui.editors.color_text_editor

A `traitsui.editors.text_editor.TextEditor` that allows you to set the foreground and background color of the text.

class `cytoflowgui.editors.color_text_editor.`**`ColorTextEditor`**(**args: Any*, ***kwargs: Any*)
Bases: `traitsui.api.BasicEditorFactory`

Editor factory for a color text editor

`klass`
alias of `cytoflowgui.editors.color_text_editor._ColorTextEditor`

`foreground_color`
The foreground color

alias of `traits.trait_types.Str`

`background_color`
The background color

alias of `traits.trait_types.Str`

cytoflowgui.editors.ext_enum_editor

A `traitsui.editors.enum_editor.EnumEditor` that allows **_names** to be extended with **extra_items** in the factory.

`class cytoflowgui.editors.ext_enum_editor.`**`ExtendableEnumEditor`**`(`**args: Any*, ***kwargs: Any*`)`

Bases: `traitsui.editors.enum_editor.EnumEditor`

`extra_items`

The extra items for the enum, beyond what's in **names**

alias of `traits.trait_types.Dict`

cytoflowgui.editors.instance_handler_editor

A `traitsui.editors.instance_editor.InstanceEditor` that allows the handler to be created at runtime, using a factory.

`class cytoflowgui.editors.instance_handler_editor.`**`InstanceHandlerEditor`**`(`**args: Any*, ***kwargs: Any*`)`

Bases: `traitsui.api.InstanceEditor`

`custom_editor_class`

alias of `cytoflowgui.editors.instance_handler_editor._InstanceHandlerEditor`

`handler_factory`

The factory to create this editor's handler

alias of `traits.trait_types.Callable`

`custom_editor``(`*ui*, *object*, *name*, *description*, *parent*`)`

Generates an editor using the "custom" style.

cytoflowgui.editors.range_slider

A `QtGui.QSlider` with two carets – allows a range to be specified.

`class cytoflowgui.editors.range_slider.`**`RangeSlider`**`(`**args: Any*, ***kwargs: Any*`)`

Bases: `pyface.qt.QtGui.QSlider`

A slider for ranges.

This class provides a dual-slider for ranges, where there is a defined maximum and minimum, as is a normal slider, but instead of having a single slider value, there are 2 slider values.

This class emits the same signals as the `QtGui.QSlider` base class, with the exception of `valueChanged`

`low``()`

`setLow``(`*low*`)`

`high``()`

`setHigh``(`*high*`)`

`sliderOrderForPaint``()`

`sliderOrderForMove``()`

`paintEvent``(`*event*`)`

`mousePressEvent``(`*event*`)`

`mouseMoveEvent`(*event*)

`mouseReleaseEvent`(*event*)

cytoflowgui.editors.subset_list_editor

An editor for lists of *ISubset*. For each entry in the list, creates an appropriate instance editor, each set up with appropriate ranges, values, etc. from the experiment conditions and metadata.

class `cytoflowgui.editors.subset_list_editor.`**SubsetListEditor**(**args: Any*, ***kwargs: Any*)

Bases: *cytoflowgui.editors.vertical_list_editor.VerticalListEditor*

conditions

The name of the trait containing the names –> values dict

alias of `traits.trait_types.Str`

metadata

The name of the trait containing the metadata dictionary

alias of `traits.trait_types.Str`

when

A string to evaluate on the metadata to see if we include this condition in the editor

alias of `traits.trait_types.Str`

mutable = <traits.trait_types.Bool object>

cytoflowgui.editors.tab_list_editor

A `traitsui.editors.enum_editor.EnumEditor` whose widget is a tab bar.

class `cytoflowgui.editors.tab_list_editor.`**TabListEditor**(**args: Any*, ***kwargs: Any*)

Bases: `traitsui.editor_factory.EditorWithListFactory`

cytoflowgui.editors.toggle_button

A button that can be toggled off and on.

class `cytoflowgui.editors.toggle_button.`**ToggleButtonEditor**(**args: Any*, ***kwargs: Any*)

Bases: `traitsui.basic_editor_factory.BasicEditorFactory`

Editor factory for toggle buttons.

klass

alias of `cytoflowgui.editors.toggle_button._ToggleButton`

value = <traits.trait_factory.TraitFactory object>

Value to set when the button is clicked

label

Optional label for the button

alias of `traits.trait_types.Str`

label_value

The name of the external object trait that the button label is synced to

alias of `traits.trait_types.Str`

cytoflowgui.editors.value_bounds_editor

A `traitsui.editors.range_editor.RangeEditor` that allows the user to select a range of values from a list (specified in **values**, naturally).

Uses *RangeSlider* for the widget.

class cytoflowgui.editors.value_bounds_editor.**ValuesBoundsEditor**(**args: Any*, ***kwargs: Any*)

Bases: `traitsui.editor_factory.EditorWithListFactory`, `traitsui.editors.api.RangeEditor`

A `traitsui.editors.range_editor.RangeEditor` that uses a list of values instead of low & high range.

cytoflowgui.editors.vertical_list_editor

A vertical editor for lists, derived from `traitsui.editors.list_editor.ListEditor`, with the same API.

Note: The difference between this class and the underlying **ListEditor** is that this class doesn't use a scroll area. Instead, as items are added, it expands. To enable this behavior, make sure you ask for the 'simple' editor style, NOT 'custom'!

class cytoflowgui.editors.vertical_list_editor.**VerticalListEditor**(**args: Any*, ***kwargs: Any*)

Bases: `traitsui.api.ListEditor`

cytoflowgui.editors.vertical_notebook

The model for *cytoflowgui.editors.vertical_notebook_editor.VerticalNotebookEditor*.

class cytoflowgui.editors.vertical_notebook.**VerticalNotebookPage**

Bases: `traits.has_traits.HasPrivateTraits`

A class representing a vertical page within a notebook.

name
The name of the page (displayed on its 'tab')

description
The description of the page (displayed in smaller text in the button)

ui
The Traits UI associated with this page

data
Optional client data associated with the page

name_object
The HasTraits object whose trait we look at to set the page name

name_object_trait
The name of the *name_object* trait that signals a page name change

description_object
The HasTraits object whose trait we look at to set the page description

description_object_trait
The name of the *description_object* trait that signals a page description change

icon
The icon for the page button – a Str that is the name of an ImageResource

`icon_object`
: The HasTraits object whose trait we look at to set the page icon

`icon_object_trait`
: The name of the *icon_object* trait that signals an icon change

`deletable`
: If the notebook has "delete" buttons, can this page be deleted?

`deletable_object`
: The HasTraits object whose trait we look at to set the delete button enabled or disabled

`deletable_object_trait`
: The name of the *deletable_object* trait that signals a deletable change

`parent`
: The parent window for the client page

`is_open`
: The current open status of the notebook page

`min_size`
: The minimum size for the page

`dispose()`
: Removes this notebook page.

`register_name_listener`(*model*, *trait*)
: Registers a listener on the specified object trait for a page name change.

`register_description_listener`(*model*, *trait*)
: Registers a listener on the specified object trait for a page description change

`register_icon_listener`(*model*, *trait*)
: Registers a listener on the specified object trait for a page icon change

`register_deletable_listener`(*model*, *trait*)
: Registers a listener on the specified object trait for the delete button enable/disable

class `cytoflowgui.editors.vertical_notebook.VerticalNotebook`
: Bases: `traits.has_traits.HasPrivateTraits`

Defines a ThemedVerticalNotebook class for displaying a series of pages organized vertically, as opposed to horizontally like a standard notebook.

`multiple_open`
: Allow multiple open pages at once?

`delete`
: can the editor delete list items?

`pages`
: The pages contained in the notebook

`editor`
: The traits UI editor this notebook is associated with (if any)

`create_control`(*parent*)
: Creates the underlying Qt window used for the notebook.

`create_page()`
: Creates a new **VerticalNotebook** object representing a notebook page and returns it as the result.

open(*page*)
Handles opening a specified notebook page.

close(*page*)
Handles closing a specified notebook page.

cytoflowgui.editors.vertical_notebook_editor

A `cytoflow`-specific re-implementation of the `traitsui.editors.list_editor.ListEditor` in *notebook* mode, but vertically instead of horizontally

class `cytoflowgui.editors.vertical_notebook_editor.`**VerticalNotebookEditor**(**args: Any*, ***kwargs: Any*)

Bases: `traitsui.basic_editor_factory.BasicEditorFactory`

klass
alias of `cytoflowgui.editors.vertical_notebook_editor._VerticalNotebookEditor`

multiple_open = <traits.trait_types.Bool object>
Allow multiple open pages at once?

delete = <traits.trait_types.Bool object>
Include a “delete” button?

page_name
List member trait to read the notebook page name from

alias of `traits.trait_types.Str`

page_description
List member trait to read the notebook page name from

alias of `traits.trait_types.Str`

page_icon
List member trait to read the notebook page icon from If None, then use right-arrow for “closed” and down-arrow for “open” The type of this trait is toolkit-specific; for example, the `pyface.qt` type is a `QtGui.QStyle.StandardPixmap`

alias of `traits.trait_types.Str`

page_deletable
List member trait to specify whether the page is deletable If the “delete” trait, above, is True, then this list member trait will enable or disable the “delete” button

alias of `traits.trait_types.Str`

view
Name of the view to use for each page

handler_factory
A factory to produce a handler from an object

alias of `traits.trait_types.Callable`

selected
Name of the trait to synchronize notebook page selection with

alias of `traits.trait_types.Str`

cytoflowgui.editors.zoomable_html_editor

An HTML "editor" that is high-DPI aware.

Derived from `traitsui.editors.html_editor`

Adapted from: https://github.com/enthought/traitsui/blob/master/traitsui/editors/html_editor.py https://github.com/enthought/traitsui/blob/master/traitsui/qt4/html_editor.py

class `cytoflowgui.editors.zoomable_html_editor.`**`ZoomableHTMLEditor`**(**args: Any*, ***kwargs: Any*)

Bases: `traitsui.editors.html_editor.ToolkitEditorFactory`

Editor factory for zoomable HTML editors.

`klass`

alias of `cytoflowgui.editors.zoomable_html_editor._ZoomableHTMLEditor`

cytoflowgui.op_plugins package

cytoflowgui.op_plugins

`envisage.plugin.Plugin` classes and GUI `traitsui.handler.Controller` classes to adapt modules from *`cytoflowgui.workflow.operations`* to the Qt / `traitsui` GUI.

The module docstrings are also the ones that are used for the GUI help panel.

Submodules

cytoflowgui.op_plugins.autofluorescence

Apply autofluorescence correction to a set of fluorescence channels.

This module estimates the arithmetic median fluorescence from cells that are not fluorescent, then subtracts the median from the experimental data.

Check the diagnostic plot to make sure that the sample is actually non-fluorescent, and that the module found the population median.

`Channels`

The channels to correct

`Blank file`

The FCS file containing measurements of blank cells.

Note: You cannot have any operations before this one which estimate model parameters based on experimental conditions. (Eg, you can't use a **Density Gate** to choose morphological parameters and set *by* to an experimental condition.) If you need this functionality, you can access it using the Python module interface.

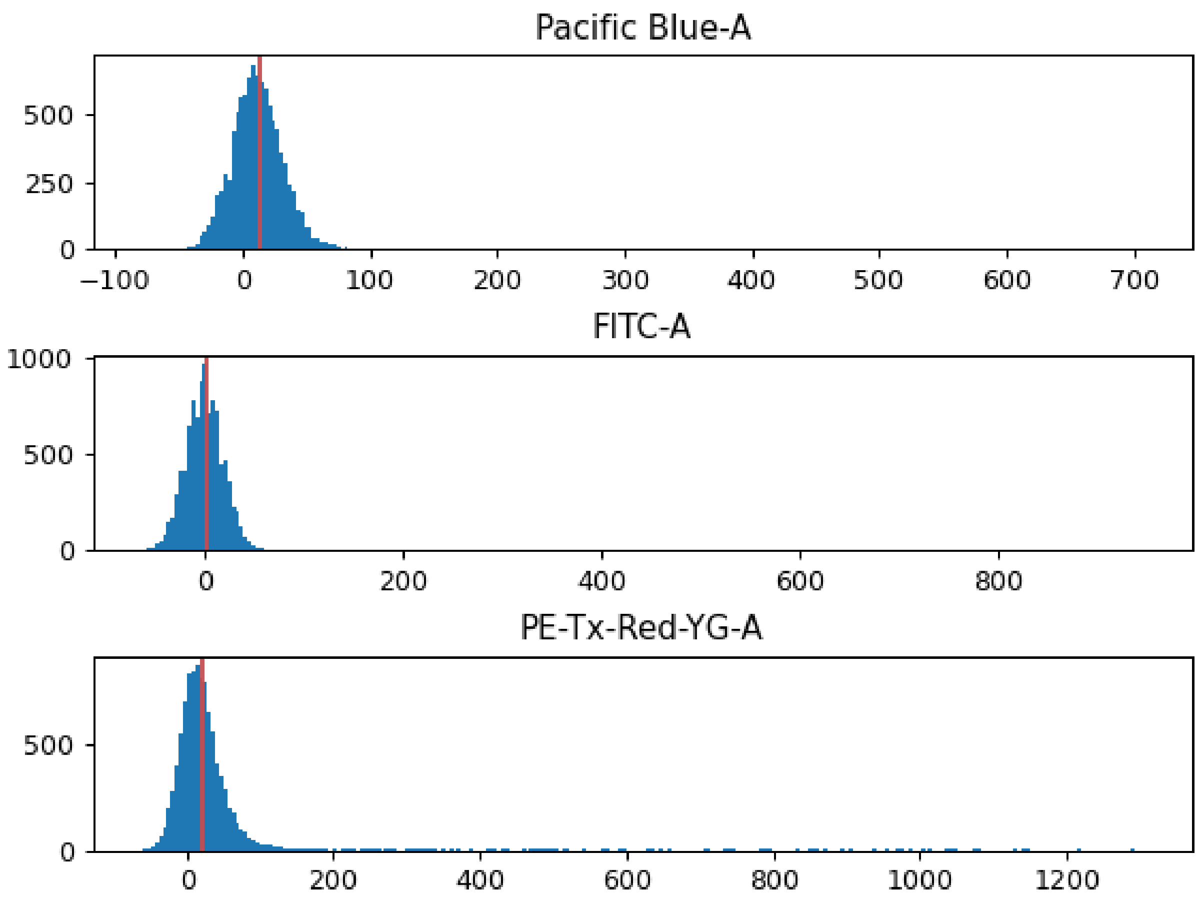
---

class `cytoflowgui.op_plugins.autofluorescence.`**`AutofluorescenceHandler`**(**args: Any*, ***kwargs: Any*)

Bases: *`cytoflowgui.op_plugins.op_plugin_base.OpHandler`*

class `cytoflowgui.op_plugins.autofluorescence.`**`AutofluorescenceViewHandler`**(**args: Any*, ***kwargs: Any*)

Bases: *`cytoflowgui.view_plugins.view_plugin_base.ViewHandler`*


Pacific Blue-A
0
250
500
−100
0
100
200
300
400
500
600
700
FITC-A
0
500
1000
0
200
400
600
800
PE-Tx-Red-YG-A
0
500
0
200
400
600
800
1000
1200

`class cytoflowgui.op_plugins.autofluorescence.`**`AutofluorescencePlugin`**

Bases: `envisage.plugin.Plugin`, *`cytoflowgui.op_plugins.op_plugin_base.PluginHelpMixin`*

`operation_id = 'edu.mit.synbio.cytoflow.operations.autofluorescence'`

`view_id = 'edu.mit.synbio.cytoflow.view.autofluorescencediagnosticview'`

`short_name = 'Autofluorescence correction'`

`menu_group = 'Calibration'`

`get_operation``()`

`get_handler``(`*model*, *context*`)`

`get_icon``()`

cytoflowgui.op_plugins.bead_calibration

Calibrate arbitrary channels to molecules-of-fluorophore using fluorescent beads (eg, the **Spherotech RCP-30-5A** rainbow beads.)

Computes a log-linear calibration function that maps arbitrary fluorescence units to physical units (ie molecules equivalent fluorophore, or *MEF*).

To use, set **Beads** to the beads you calibrated with (check the lot!) and **Beads File** to an FCS file containing events collected *using the same cytometer settings as the data you're calibrating*. Then, click **Add a channel** to add the channels to calibrate, and set both the channel name and the units you want calibrate to. Click **Estimate**, and *make sure you check the diagnostic plot to see that the correct peaks were found.*

If it didn't find all the peaks (or found too many), try tweaking **Peak Quantile**, **Peak Threshold** and **Peak Cutoff**. If you can't make the peak finding work by tweaking , please submit a bug report!

`Beads`
: The beads you're calibrating with. Make sure to check the lot number!

`Beads file`
: A file containing the FCS events from the beads.

`Channels`
: A list of the channels you want calibrated and the units you want them calibrated in.

`Peak Quantile`
: Peaks must be at least this quantile high to be considered. Default = 80.

`Peak Threshold`
: Don't search for peaks below this brightness. Default = 100.

`class cytoflowgui.op_plugins.bead_calibration.`**`UnitHandler`**`(`**args: Any*, ***kwargs: Any*`)`

Bases: `traitsui.api.Controller`

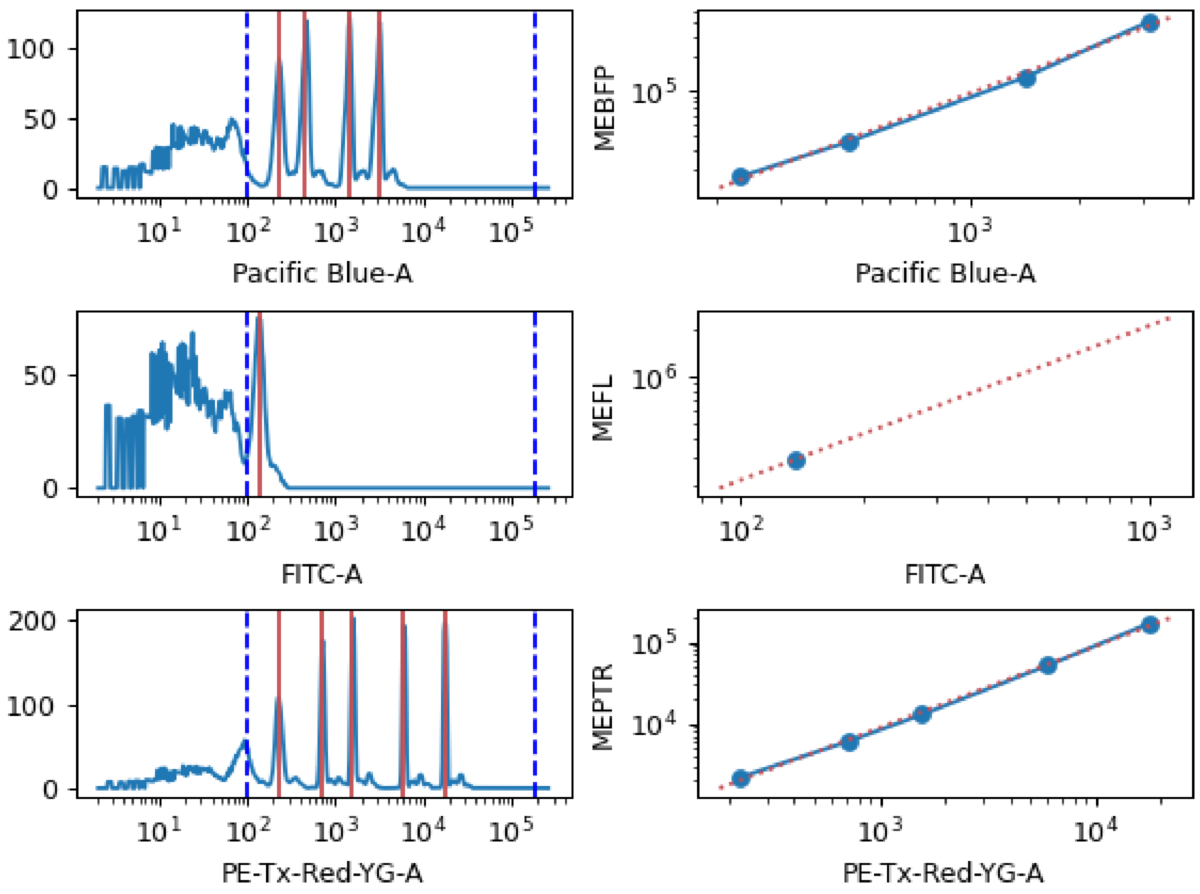
`class cytoflowgui.op_plugins.bead_calibration.`**`BeadCalibrationHandler`**`(`**args: Any*, ***kwargs: Any*`)`

Bases: *`cytoflowgui.op_plugins.op_plugin_base.OpHandler`*

`add_channel`
: alias of `traits.trait_types.Event`

`remove_channel`
: alias of `traits.trait_types.Event`

`channels = <traits.traits.ForwardProperty object>`


100
50
0
10^1
10^2
10^3
10^4
10^5
Pacific Blue-A
MEBFP
10^5
10^3
Pacific Blue-A
50
0
FITC-A
MEFL
10^6
10^2
10^3
FITC-A
200
100
0
PE-Tx-Red-YG-A
MEPTR
10^5
10^4
10^3
10^4
PE-Tx-Red-YG-A

beads_name_choices = <traits.trait_factory.TraitFactory object>

beads_units = <traits.traits.ForwardProperty object>

class cytoflowgui.op_plugins.bead_calibration.**BeadCalibrationViewHandler**(**args: Any*, ***kwargs: Any*)

Bases: *cytoflowgui.view_plugins.view_plugin_base.ViewHandler*

class cytoflowgui.op_plugins.bead_calibration.**BeadCalibrationPlugin**

Bases: envisage.plugin.Plugin, *cytoflowgui.op_plugins.op_plugin_base.PluginHelpMixin*

operation_id = 'edu.mit.synbio.cytoflow.operations.beads_calibrate'

view_id = 'edu.mit.synbio.cytoflow.view.beadcalibrationdiagnosticview'

short_name = 'Bead Calibration'

menu_group = 'Calibration'

get_operation()

get_handler(*model*, *context*)

get_icon()

cytoflowgui.op_plugins.binning

Bin data along an axis.

This operation creates equally spaced bins (in linear or log space) along an axis and adds a condition assigning each event to a bin. The value of the event's condition is the left end of the bin's interval in which the event is located.

Name
The name of the new condition created by this operation.

Channel
The channel to apply the binning to.

Scale
The scale to apply to the channel before binning.

Bin Width
How wide should each bin be? Can only set if **Scale** is *linear* or *log* (in which case, **Bin Width** is in log10-units.)

class cytoflowgui.op_plugins.binning.**BinningHandler**(**args: Any*, ***kwargs: Any*)

Bases: *cytoflowgui.op_plugins.op_plugin_base.OpHandler*

class cytoflowgui.op_plugins.binning.**BinningViewHandler**(**args: Any*, ***kwargs: Any*)

Bases: *cytoflowgui.view_plugins.view_plugin_base.ViewHandler*

class cytoflowgui.op_plugins.binning.**BinningPlugin**

Bases: envisage.plugin.Plugin, *cytoflowgui.op_plugins.op_plugin_base.PluginHelpMixin*

operation_id = 'edu.mit.synbio.cytoflow.operations.binning'

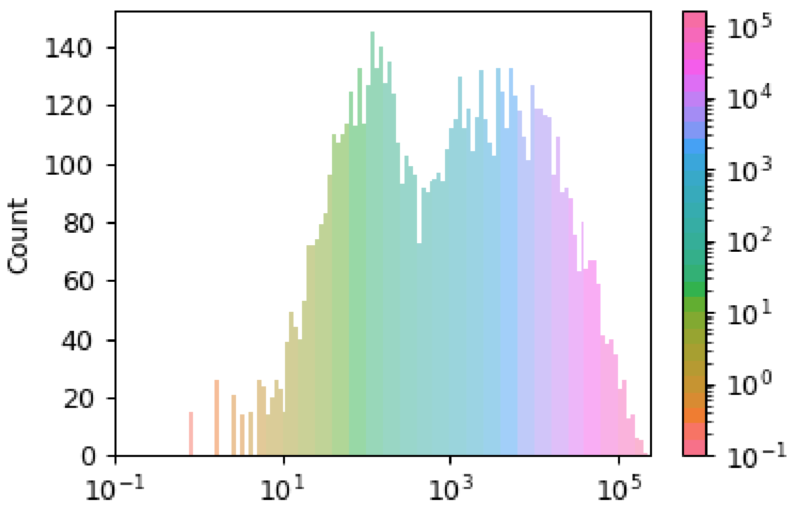
view_id = 'edu.mit.synbio.cytoflow.views.binning'

short_name = 'Binning'

menu_group = 'Gates'

get_operation()

get_handler(*model*, *context*)



get_icon()

cytoflowgui.op_plugins.bleedthrough_linear

Apply matrix-based bleedthrough correction to a set of fluorescence channels.

This is a traditional matrix-based compensation for bleedthrough. For each pair of channels, the module estimates the proportion of the first channel that bleeds through into the second, then performs a matrix multiplication to compensate the raw data.

This works best on data that has had autofluorescence removed first; if that is the case, then the autofluorescence will be subtracted from the single-color controls too.

To use, specify the single-color control files and which channels they should be measured in, then click **Estimate**. Check the diagnostic plot to make sure the estimation looks good. There must be at least two channels corrected.

Add Control, Remove Control
Add or remove single-color controls.

Note: You cannot have any operations before this one which estimate model parameters based on experimental conditions. (Eg, you can't use a **Density Gate** to choose morphological parameters and set *by* to an experimental condition.) If you need this functionality, you can access it using the Python module interface.

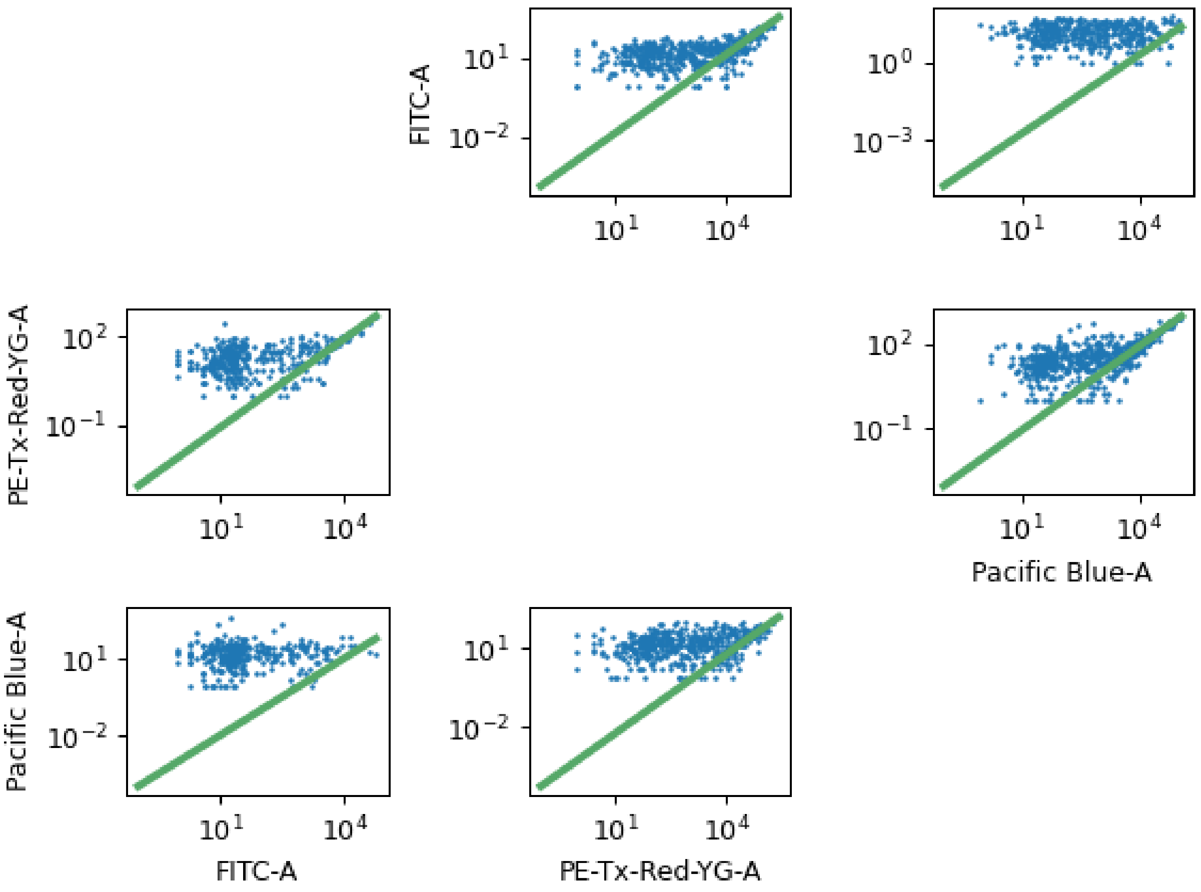
class cytoflowgui.op_plugins.bleedthrough_linear.**ControlHandler**(**args: Any*, ***kwargs: Any*)
Bases: traitsui.api.Controller

class cytoflowgui.op_plugins.bleedthrough_linear.**BleedthroughLinearHandler**(**args: Any*, ***kwargs: Any*)

Bases: *cytoflowgui.op_plugins.op_plugin_base.OpHandler*

add_control
alias of traits.trait_types.Event

remove_control
alias of traits.trait_types.Event


FITC-A
PE-Tx-Red-YG-A
Pacific Blue-A
FITC-A
PE-Tx-Red-YG-A
Pacific Blue-A
10^1
10^4
10^{-2}
10^0
10^{-3}
10^2
10^{-1}

channels = <traits.traits.ForwardProperty object>

class cytoflowgui.op_plugins.bleedthrough_linear.**BleedthroughLinearViewHandler**(**args: Any*, ***kwargs: Any*)

Bases: *cytoflowgui.view_plugins.view_plugin_base.ViewHandler*

class cytoflowgui.op_plugins.bleedthrough_linear.**BleedthroughLinearPlugin**

Bases: envisage.plugin.Plugin, *cytoflowgui.op_plugins.op_plugin_base.PluginHelpMixin*

operation_id = 'edu.mit.synbio.cytoflow.operations.bleedthrough_linear'

view_id = 'edu.mit.synbio.cytoflow.view.linearbleedthroughdiagnostic'

short_name = 'Linear Compensation'

menu_group = 'Gates'

get_operation()

get_handler(*model*, *context*)

get_icon()

cytoflowgui.op_plugins.channel_stat

Apply a function to subsets of a data set, and add it as a statistic to the experiment.

First, the module groups the data by the unique values of the variables in **By**, then applies **Function** to the **Channel** in each group.

Name
The operation name. Becomes the first part of the new statistic's name.

Channel
The channel to apply the function to.

Function
The function to compute on each group.

Subset
Only apply the function to a subset of the data. Useful if the function is very slow.

class cytoflowgui.op_plugins.channel_stat.**ChannelStatisticHandler**(**args: Any*, ***kwargs: Any*)

Bases: *cytoflowgui.op_plugins.op_plugin_base.OpHandler*

class cytoflowgui.op_plugins.channel_stat.**ChannelStatisticPlugin**

Bases: envisage.plugin.Plugin, *cytoflowgui.op_plugins.op_plugin_base.PluginHelpMixin*

operation_id = 'edu.mit.synbio.cytoflow.operations.channel_statistic'

view_id = None

short_name = 'Channel Statistic'

menu_group = 'Gates'

get_operation()

get_handler(*model*, *context*)

get_icon()

cytoflowgui.op_plugins.color_translation

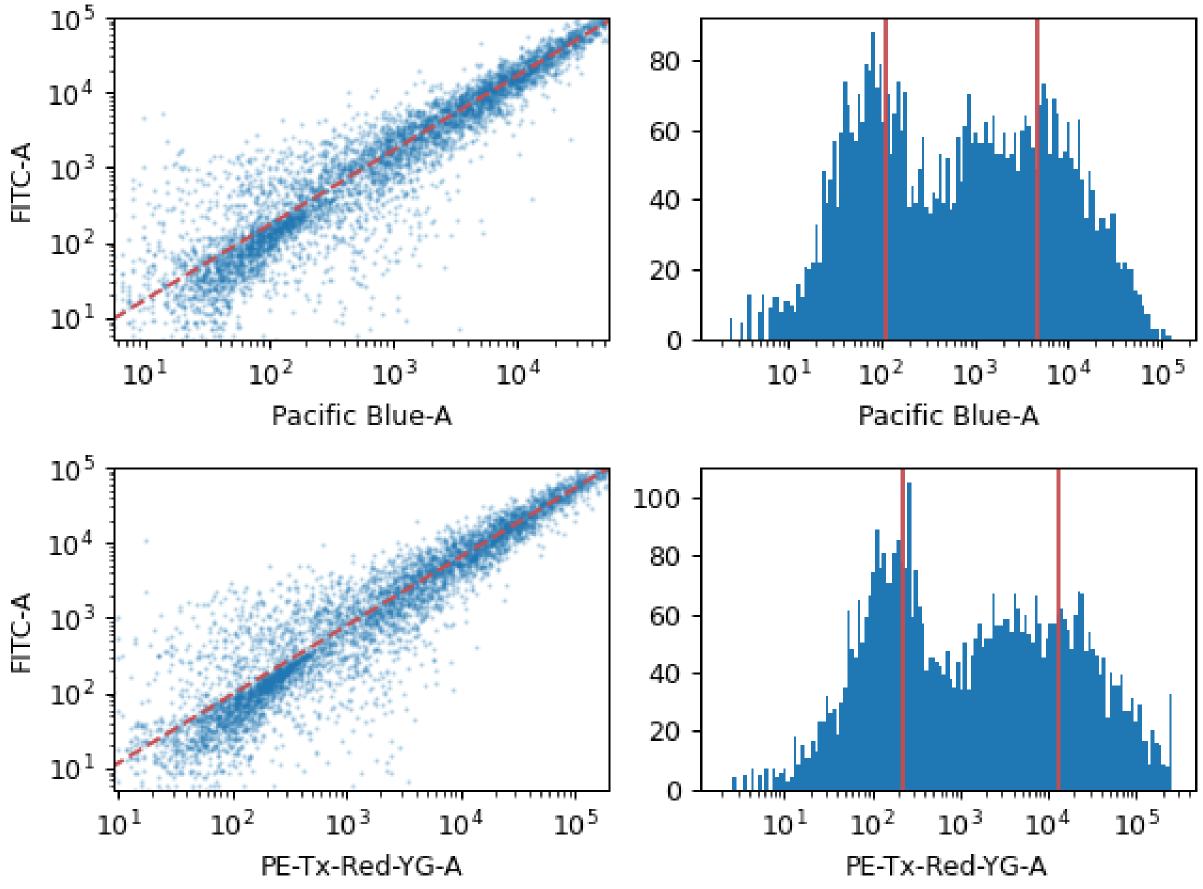
Translate measurements from one color's scale to another, using a two-color or three-color control.

To use, set up the **Controls** list with the channels to convert and the FCS files to compute the mapping. Click **Estimate** and make sure to check that the diagnostic plots look good.

`Add Control, Remove Control`
: Add and remove controls to compute the channel mappings.

`Use mixture model?`
: If `True`, try to model the **from** channel as a mixture of expressing cells and non-expressing cells (as you would get with a transient transfection), then weight the regression by the probability that the the cell is from the top (transfected) distribution. Make sure you check the diagnostic plots to see that this worked!

Note: You cannot have any operations before this one which estimate model parameters based on experimental conditions. (Eg, you can't use a **Density Gate** to choose morphological parameters and set *by* to an experimental condition.) If you need this functionality, you can access it using the Python module interface.



class `cytoflowgui.op_plugins.color_translation.`**`ControlHandler`**`(`**args: Any*, ***kwargs: Any*`)`
: Bases: `traitsui.api.Controller`

class cytoflowgui.op_plugins.color_translation.**ColorTranslationHandler**(**args: Any*, ***kwargs: Any*)

Bases: *cytoflowgui.op_plugins.op_plugin_base.OpHandler*

add_control

alias of `traits.trait_types.Event`

remove_control

alias of `traits.trait_types.Event`

channels = <traits.traits.ForwardProperty object>

class cytoflowgui.op_plugins.color_translation.**ColorTranslationViewHandler**(**args: Any*, ***kwargs: Any*)

Bases: *cytoflowgui.view_plugins.view_plugin_base.ViewHandler*

class cytoflowgui.op_plugins.color_translation.**ColorTranslationPlugin**

Bases: `envisage.plugin.Plugin`, *cytoflowgui.op_plugins.op_plugin_base.PluginHelpMixin*

operation_id = 'edu.mit.synbio.cytoflow.operations.color_translation'

view_id = 'edu.mit.synbio.cytoflow.view.colortranslationdiagnostic'

short_name = 'Color Translation'

menu_group = 'Gates'

get_operation()

get_handler(*model*, *context*)

get_icon()

cytoflowgui.op_plugins.density

Computes a gate based on a 2D density plot. The user chooses what proportion of events to keep, and the module creates a gate that selects those events in the highest-density bins of a 2D density histogram.

A single gate may not be appropriate for an entire experiment. If this is the case, you can use **By** to specify metadata by which to aggregate the data before computing and applying the gate.

Name

The operation name; determines the name of the new metadata

X Channel, Y Channel

The channels to apply the mixture model to.

X Scale, Y Scale

Re-scale the data in **Channel** before fitting.

Keep

The proportion of events to keep in the gate. Defaults to 0.9.

By

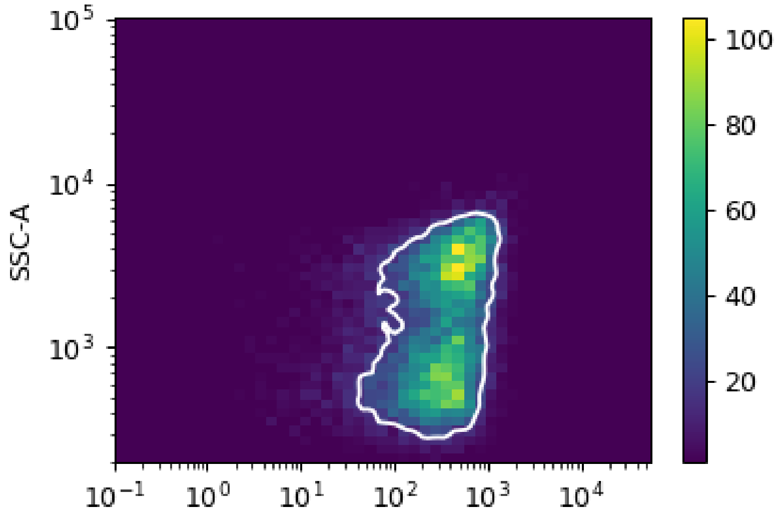
A list of metadata attributes to aggregate the data before estimating the model. For example, if the experiment has two pieces of metadata, `Time` and `Dox`, setting **by** to `["Time", "Dox"]` will fit the model separately to each subset of the data with a unique combination of `Time` and `Dox`.

class cytoflowgui.op_plugins.density.**DensityGateHandler**(**args: Any*, ***kwargs: Any*)

Bases: *cytoflowgui.op_plugins.op_plugin_base.OpHandler*

class cytoflowgui.op_plugins.density.**DensityGateViewHandler**(**args: Any*, ***kwargs: Any*)

Bases: *cytoflowgui.view_plugins.view_plugin_base.ViewHandler*



class cytoflowgui.op_plugins.density.**DensityGatePlugin**

Bases: `envisage.plugin.Plugin`, *cytoflowgui.op_plugins.op_plugin_base.PluginHelpMixin*

operation_id = 'edu.mit.synbio.cytoflow.operations.density'

view_id = 'edu.mit.synbio.cytoflow.view.densitygateview'

short_name = 'Density Gate'

menu_group = 'Gates'

get_operation()

get_handler(*model*, *context*)

get_icon()

cytoflowgui.op_plugins.flowpeaks

This module uses the **flowPeaks** algorithm to assign events to clusters in an unsupervized manner.

Name
The operation name; determines the name of the new metadata

X Channel, Y Channel
The channels to apply the mixture model to.

X Scale, Y Scale
Re-scale the data in **Channel** before fitting.

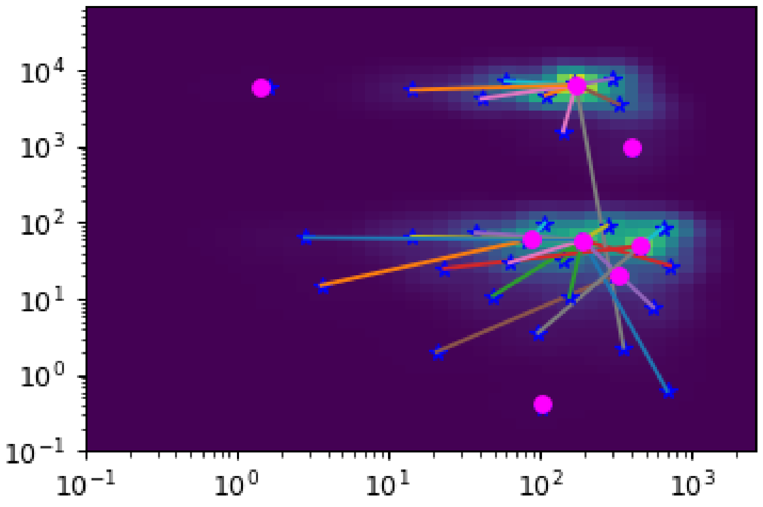
h, h0
Scalar values that control the smoothness of the estimated distribution. Increasing **h** makes it “rougher,” while increasing **h0** makes it smoother.

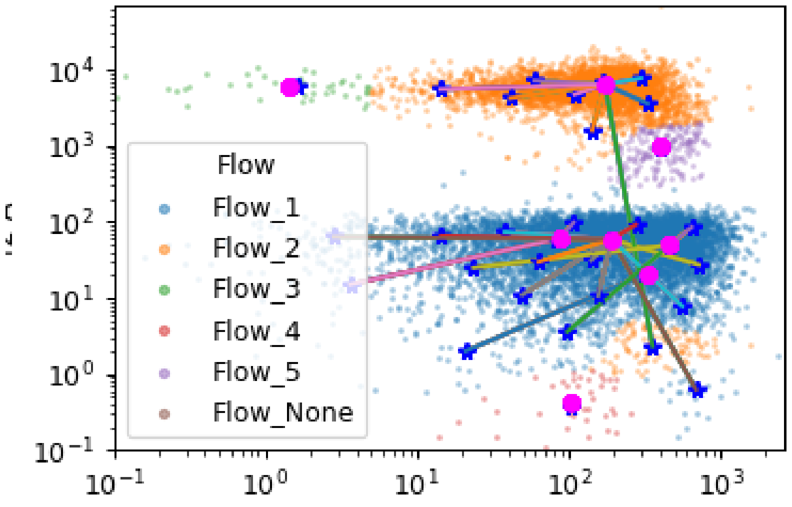
tol
How readily should clusters be merged? Must be between 0 and 1.

Merge Distance
How far apart can clusters be before they are merged?

By

A list of metadata attributes to aggregate the data before estimating the model. For example, if the experiment has two pieces of metadata, `Time` and `Dox`, setting **By** to `["Time", "Dox"]` will fit the model separately to each subset of the data with a unique combination of `Time` and `Dox`.





class cytoflowgui.op_plugins.flowpeaks.**FlowPeaksHandler**(**args: Any*, ***kwargs: Any*)
  Bases: *cytoflowgui.op_plugins.op_plugin_base.OpHandler*

class cytoflowgui.op_plugins.flowpeaks.**FlowPeaksViewHandler**(**args: Any*, ***kwargs: Any*)
  Bases: *cytoflowgui.view_plugins.view_plugin_base.ViewHandler*

class cytoflowgui.op_plugins.flowpeaks.**FlowPeaksPlugin**
  Bases: envisage.plugin.Plugin, *cytoflowgui.op_plugins.op_plugin_base.PluginHelpMixin*

  operation_id = 'edu.mit.synbio.cytoflow.operations.flowpeaks'

  view_id = 'edu.mit.synbio.cytoflowgui.op_plugins.flowpeaks'

short_name = 'Flow Peaks'

menu_group = 'Gates'

get_operation()

get_handler(*model*, *context*)

get_icon()

cytoflowgui.op_plugins.gaussian_1d

Fit a Gaussian mixture model with a specified number of components to one channel.

If **Num Components** is greater than 1, then this module creates a new categorical metadata variable named **Name**, with possible values `{name}_1` .... `name_n` where `n` is the number of components. An event is assigned to `name_i` category if it has the highest posterior probability of having been produced by component `i`. If an event has a value that is outside the range of one of the channels' scales, then it is assigned to `{name}_None`.

Additionally, if **Sigma** is greater than 0, this module creates new boolean metadata variables named `{name}_1` ... `{name}_n` where `n` is the number of components. The column `{name}_i` is `True` if the event is less than **Sigma** standard deviations from the mean of component `i`. If **Num Components** is 1, **Sigma** must be greater than 0.

Finally, the same mixture model (mean and standard deviation) may not be appropriate for every subset of the data. If this is the case, you can use **By** to specify metadata by which to aggregate the data before estimating and applying a mixture model.

Note: Num Components and **Sigma** withh be the same for each subset.

`Name`
The operation name; determines the name of the new metadata

`Channel`
The channels to apply the mixture model to.

`Scale`
Re-scale the data in **Channel** before fitting.

`Num Components`
How many components to fit to the data? Must be a positive integer.

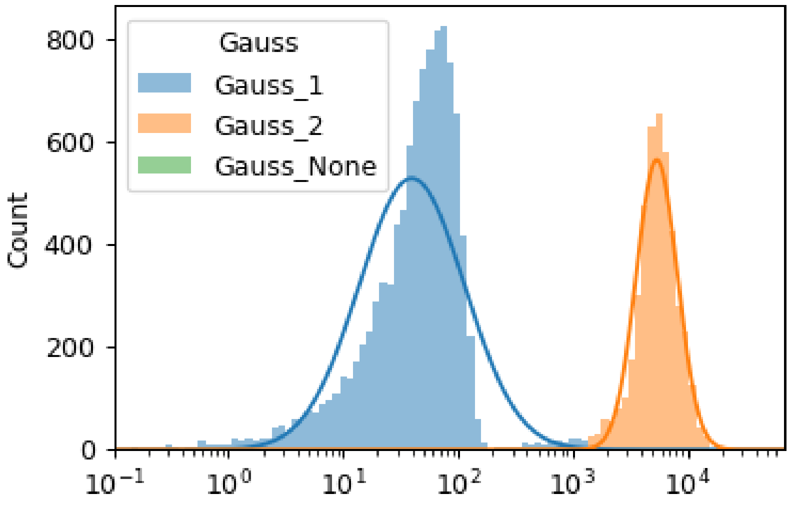
`Sigma`
How many standard deviations on either side of the mean to include in the boolean variable `{name}_i`? Must be `None` or > `0.0`. If **Num Components** is `1`, must be > `0`.

`By`
A list of metadata attributes to aggregate the data before estimating the model. For example, if the experiment has two pieces of metadata, `Time` and `Dox`, setting **By** to `["Time", "Dox"]` will fit the model separately to each subset of the data with a unique combination of `Time` and `Dox`.

class cytoflowgui.op_plugins.gaussian_1d.**GaussianMixture1DHandler**(**args: Any*, ***kwargs: Any*)
Bases: *cytoflowgui.op_plugins.op_plugin_base.OpHandler*

class cytoflowgui.op_plugins.gaussian_1d.**GaussianMixture1DViewHandler**(**args: Any*, ***kwargs: Any*)
Bases: *cytoflowgui.view_plugins.view_plugin_base.ViewHandler*

class cytoflowgui.op_plugins.gaussian_1d.**GaussianMixture1DPlugin**
Bases: envisage.plugin.Plugin, *cytoflowgui.op_plugins.op_plugin_base.PluginHelpMixin*



`operation_id = 'edu.mit.synbio.cytoflowgui.operations.gaussian_1d'`

`view_id = 'edu.mit.synbio.cytoflow.view.gaussianmixture1dview'`

`short_name = '1D Mixture Model'`

`menu_group = 'Gates'`

`get_operation()`

`get_handler`(*model*, *context*)

`get_icon()`

cytoflowgui.op_plugins.gaussian_2d

Fit a Gaussian mixture model with a specified number of components to two channels.

If **Num Components** is greater than 1, then this module creates a new categorical metadata variable named **Name**, with possible values `{name}_1` .... `name_n` where `n` is the number of components. An event is assigned to `name_i` category if it has the highest posterior probability of having been produced by component `i`. If an event has a value that is outside the range of one of the channels' scales, then it is assigned to `{name}_None`.

Additionally, if **Sigma** is greater than 0, this module creates new boolean metadata variables named `{name}_1` … `{name}_n` where `n` is the number of components. The column `{name}_i` is `True` if the event is less than **Sigma** standard deviations from the mean of component `i`. If **Num Components** is 1, **Sigma** must be greater than 0.

Finally, the same mixture model (mean and standard deviation) may not be appropriate for every subset of the data. If this is the case, you can use **By** to specify metadata by which to aggregate the data before estimating and applying a mixture model.

Note: **Num Components** and **Sigma** will be the same for each subset.

`Name`
The operation name; determines the name of the new metadata

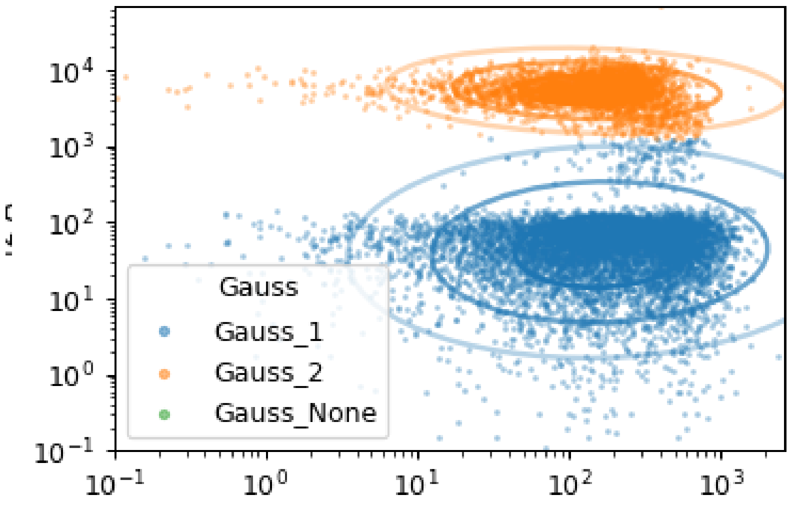
X Channel, Y Channel
The channels to apply the mixture model to.

X Scale, Y Scale
Re-scale the data in **Channel** before fitting.

Num Components
How many components to fit to the data? Must be a positive integer.

Sigma
How many standard deviations on either side of the mean to include in the boolean variable `{name}_i`? Must be >= `0.0`. If **Num Components** is 1, must be > `0`.

By
A list of metadata attributes to aggregate the data before estimating the model. For example, if the experiment has two pieces of metadata, `Time` and `Dox`, setting **By** to `["Time", "Dox"]` will fit the model separately to each subset of the data with a unique combination of `Time` and `Dox`.



class cytoflowgui.op_plugins.gaussian_2d.**GaussianMixture2DHandler**(**args: Any*, ***kwargs: Any*)
Bases: *cytoflowgui.op_plugins.op_plugin_base.OpHandler*

class cytoflowgui.op_plugins.gaussian_2d.**GaussianMixture2DViewHandler**(**args: Any*, ***kwargs: Any*)
Bases: *cytoflowgui.view_plugins.view_plugin_base.ViewHandler*

class cytoflowgui.op_plugins.gaussian_2d.**GaussianMixture2DPlugin**
Bases: envisage.plugin.Plugin, *cytoflowgui.op_plugins.op_plugin_base.PluginHelpMixin*

operation_id = 'edu.mit.synbio.cytoflowgui.operations.gaussian_2d'

view_id = 'edu.mit.synbio.cytoflow.view.gaussianmixture2dview'

short_name = '2D Mixture Model'

menu_group = 'Gates'

get_operation()

get_handler(*model*, *context*)

get_icon()

cytoflowgui.op_plugins.i_op_plugin

class cytoflowgui.op_plugins.i_op_plugin.**IOperationPlugin**(*adaptee*, *default=<class 'traits.adaptation.adaptation_error.AdaptationError'>*)

Bases: `traits.has_traits.Interface`

id

The Envisage ID used to refer to this plugin

Type Constant

operation_id

Same as the `id` attribute of the IOperation this plugin wraps.

Type Constant

short_name

The operation's `short` name - for menus and toolbar tool tips

Type Constant

get_operation()

Makes an instance of the *`IWorkflowOperation`* that this plugin wraps.

Return type *`IWorkflowOperation`*

get_handler(*model*)

Makes an instance of a Controller for the operation.

Parameters model (*IWorkflowOperation*) – The model that this handler handles.

Return type `traitsui.handler.Controller`

get_icon()

Gets the icon for this operation.

Returns The SVG icon

Return type `pyface.i_image_resource.IImageResource`

get_help()

Gets the HTML help text for this plugin, deriving the filename from the class name. Probably best to use the default implementation in *`cytoflowgui.op_plugins.op_plugin_base.PluginHelpMixin`*

Returns The HTML help, in a single string.

Return type string

class cytoflowgui.op_plugins.i_op_plugin.**OpPluginManager**

Bases: `envisage.plugin.Plugin`

cytoflowgui.op_plugins.import_op

Import FCS files and associate them with experimental conditions (metadata.)

Channels

Here, you can rename channels to use names that are more informative, or remove channels you don't need. Names must be valid Python identifiers (must contain only letters, numbers and underscores and must start with a letter or underscore.)

Reset channel names

Reset the channels and channel names.

`Events per sample`
For very large data sets, *Cytoflow*'s interactive operation may be too slow. By setting **Events per sample**, you can tell *Cytoflow* to import a smaller number of events from each FCS file, which will make interactive data exploration much faster. When you're done setting up your workflow, set **Events per sample** to empty or 0 and *Cytoflow* will re-run your workflow with the entire data set.

`Set up experiment....`
Open the sample editor dialog box.

`The sample editor dialog`

`dev_manual/api/images/import.png`

Allows you to specify FCS files in the experiment, and the experimental conditions that each tube (or well) was subject to.

Note: You can select sort the table by clicking on a row header.

Note: You can select multiple entries in a column by clicking one, holding down *Shift*, and clicking another (to select a range); or, by holding down *Ctrl* and clicking multiple additional cells in the table. If multiple cells are selected, typing a value will update all of them.

Note: **Each tube must have a unique set of experimental conditions.** If a tube's conditions are not unique, the row is red and you will not be able to click "OK".

`Add tubes`
Opens a file selector to add tubes.

class cytoflowgui.op_plugins.import_op.**ChannelHandler**(**args: Any*, ***kwargs: Any*)
Bases: `traitsui.api.Controller`

class cytoflowgui.op_plugins.import_op.**ImportHandler**(**args: Any*, ***kwargs: Any*)
Bases: *`cytoflowgui.op_plugins.op_plugin_base.OpHandler`*

`setup_event = <traits.trait_types.Button object>`

`reset_channels_event = <traits.trait_types.Event object>`

`samples = <traits.traits.ForwardProperty object>`

`dialog_model = <traits.trait_types.Instance object>`

class cytoflowgui.op_plugins.import_op.**ImportPlugin**
Bases: `envisage.plugin.Plugin`, *`cytoflowgui.op_plugins.op_plugin_base.PluginHelpMixin`*

`operation_id = 'edu.mit.synbio.cytoflow.operations.import'`

`view_id = None`

`short_name = 'Import data'`

`menu_group = 'TOP'`

get_operation()

get_handler(*model*, *context*)

get_icon()

cytoflowgui.op_plugins.kmeans

This module uses the **KMeans** algorithm to assign events to clusters in an unsupervized manner.

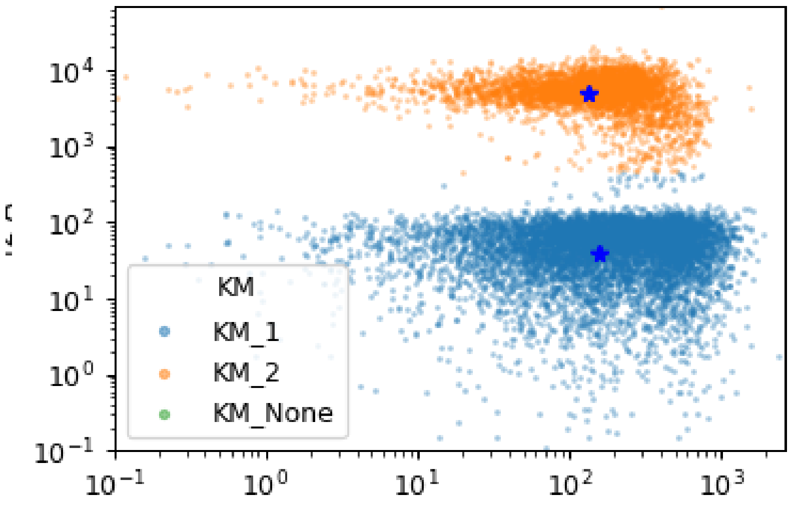
Name
The operation name; determines the name of the new metadata

X Channel, Y Channel
The channels to apply the mixture model to.

X Scale, Y Scale
Re-scale the data in **Channel** before fitting.

Num Clusters
How many clusters to assign the data to.

By
A list of metadata attributes to aggregate the data before estimating the model. For example, if the experiment has two pieces of metadata, `Time` and `Dox`, setting **By** to `["Time", "Dox"]` will fit the model separately to each subset of the data with a unique combination of `Time` and `Dox`.



class cytoflowgui.op_plugins.kmeans.**KMeansHandler**(**args: Any*, ***kwargs: Any*)
Bases: *cytoflowgui.op_plugins.op_plugin_base.OpHandler*

class cytoflowgui.op_plugins.kmeans.**KMeansViewHandler**(**args: Any*, ***kwargs: Any*)
Bases: *cytoflowgui.view_plugins.view_plugin_base.ViewHandler*

class cytoflowgui.op_plugins.kmeans.**KMeansPlugin**
Bases: `envisage.plugin.Plugin`, *cytoflowgui.op_plugins.op_plugin_base.PluginHelpMixin*

operation_id = 'edu.mit.synbio.cytoflow.operations.kmeans'

view_id = 'edu.mit.synbio.cytoflowgui.op_plugins.kmeans'

`short_name = 'KMeans'`

`menu_group = 'Gates'`

`get_operation()`

`get_handler(`*model*, *context*`)`

`get_icon()`

cytoflowgui.op_plugins.op_plugin_base

@author: brian

`class cytoflowgui.op_plugins.op_plugin_base.OpHandler(`**args: Any*, ***kwargs: Any*`)`

Bases: `traitsui.api.Controller`

Base class for operation handlers.

`context = <traits.trait_types.Instance object>`

`class cytoflowgui.op_plugins.op_plugin_base.PluginHelpMixin`

Bases: `traits.has_traits.HasTraits`

`get_help()`

Gets the HTML help for this module, deriving the filename from the class name.

Returns The HTML help, in a single string.

Return type string

cytoflowgui.op_plugins.pca

Use principal components analysis (PCA) to decompose a multivariate data set into orthogonal components that explain a maximum amount of variance.

Creates new "channels" named `{name}_1 ... {name}_n`, where `name` is the **Name** attribute and `n` is **Num components**.

The same decomposition may not be appropriate for different subsets of the data set. If this is the case, you can use the **By** attribute to specify metadata by which to aggregate the data before estimating (and applying) a model. The PCA parameters such as the number of components and the kernel are the same across each subset, though.

`Name`

The operation name; determines the name of the new columns.

`Channels`

The channels to apply the decomposition to.

`Scale`

Re-scale the data in the specified channels before fitting.

`Num components`

How many components to fit to the data? Must be a positive integer.

`By`

A list of metadata attributes to aggregate the data before estimating the model. For example, if the experiment has two pieces of metadata, `Time` and `Dox`, setting **By** to `["Time", "Dox"]` will fit the model separately to each subset of the data with a unique combination of `Time` and `Dox`.

Whiten
Scale each component to unit variance? May be useful if you will be using unsupervized clustering (such as K-means).

class cytoflowgui.op_plugins.pca.**ChannelHandler**(**args: Any*, ***kwargs: Any*)
Bases: traitsui.api.Controller

class cytoflowgui.op_plugins.pca.**PCAHandler**(**args: Any*, ***kwargs: Any*)
Bases: *cytoflowgui.op_plugins.op_plugin_base.OpHandler*

add_channel
alias of traits.trait_types.Event

remove_channel
alias of traits.trait_types.Event

channels = <traits.traits.ForwardProperty object>

class cytoflowgui.op_plugins.pca.**PCAPlugin**
Bases: envisage.plugin.Plugin, *cytoflowgui.op_plugins.op_plugin_base.PluginHelpMixin*

operation_id = 'edu.mit.synbio.cytoflow.operations.pca'

view_id = None

short_name = 'Principal Component Analysis'

menu_group = 'Calibration'

get_operation()

get_handler(*model*, *context*)

get_icon()

cytoflowgui.op_plugins.polygon

Draw a polygon gate. To add vertices, use a single-click; to close the polygon, click the first vertex a second time.

Name
The operation name. Used to name the new metadata field that's created by this module.

X Channel
The name of the channel on the gate's X axis.

Y Channel
The name of the channel on the gate's Y axis.

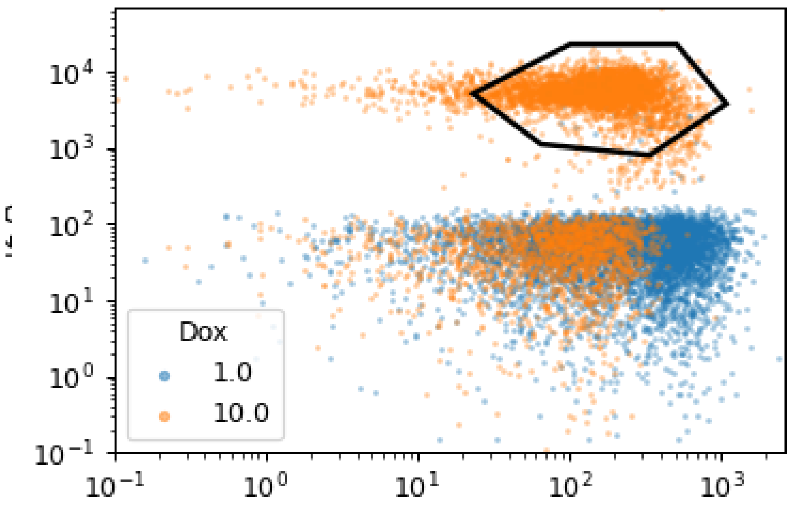
X Scale
The scale of the X axis for the interactive plot.

Y Scale
The scale of the Y axis for the interactive plot

Hue facet
Show different experimental conditions in different colors.

Subset
Show only a subset of the data.

class cytoflowgui.op_plugins.polygon.**PolygonHandler**(**args: Any*, ***kwargs: Any*)
Bases: *cytoflowgui.op_plugins.op_plugin_base.OpHandler*



class cytoflowgui.op_plugins.polygon.**PolygonViewHandler**(**args: Any*, ***kwargs: Any*)

Bases: `cytoflowgui.view_plugins.view_plugin_base.ViewHandler`

class cytoflowgui.op_plugins.polygon.**PolygonPlugin**

Bases: `envisage.plugin.Plugin`, `cytoflowgui.op_plugins.op_plugin_base.PluginHelpMixin`

operation_id = 'edu.mit.synbio.cytoflow.operations.polygon'

view_id = 'edu.mit.synbio.cytoflow.views.polygon'

short_name = 'Polygon Gate'

menu_group = 'Gates'

get_operation()

get_handler(*model*, *context*)

get_icon()

cytoflowgui.op_plugins.quad

Draw a "quadrant" gate. To create a new gate, just click where you'd like the intersection to be. Creates a new metadata column named *name*, with values `name_1` (upper-left quadrant), `name_2` (upper-right), `name_3` (lower-left), and `name_4` (lower-right).

Note: This matches the order of FACSDiva quad gates.

Name
The operation name. Used to name the new metadata field that's created by this operation.

X channel
The name of the channel on the X axis.

X threshold
The threshold in the X channel.

Y channel
The name of the channel on the Y axis.

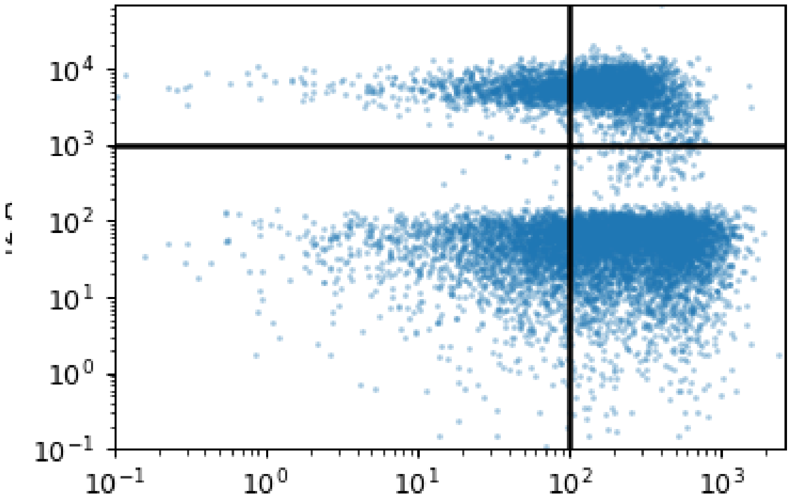
Y threshold
The threshold in the Y channel.

X Scale
The scale of the X axis for the interactive plot.

Y Scale
The scale of the Y axis for the interactive plot

Hue facet
Show different experimental conditions in different colors.

Subset
Show only a subset of the data.



class cytoflowgui.op_plugins.quad.**QuadHandler**(**args: Any*, ***kwargs: Any*)
Bases: *cytoflowgui.op_plugins.op_plugin_base.OpHandler*

class cytoflowgui.op_plugins.quad.**QuadViewHandler**(**args: Any*, ***kwargs: Any*)
Bases: *cytoflowgui.view_plugins.view_plugin_base.ViewHandler*

class cytoflowgui.op_plugins.quad.**QuadPlugin**
Bases: envisage.plugin.Plugin, *cytoflowgui.op_plugins.op_plugin_base.PluginHelpMixin*

operation_id = 'edu.mit.synbio.cytoflow.operations.quad'

view_id = 'edu.mit.synbio.cytoflow.views.quad'

short_name = 'Quad'

menu_group = 'Gates'

get_operation()

get_handler(*model*, *context*)

get_icon()

cytoflowgui.op_plugins.range

Draw a range gate. To draw a new range, click-and-drag across the plot.

Name
: The operation name. Used to name the new metadata field that's created by this module.

Channel
: The name of the channel to apply the gate to.

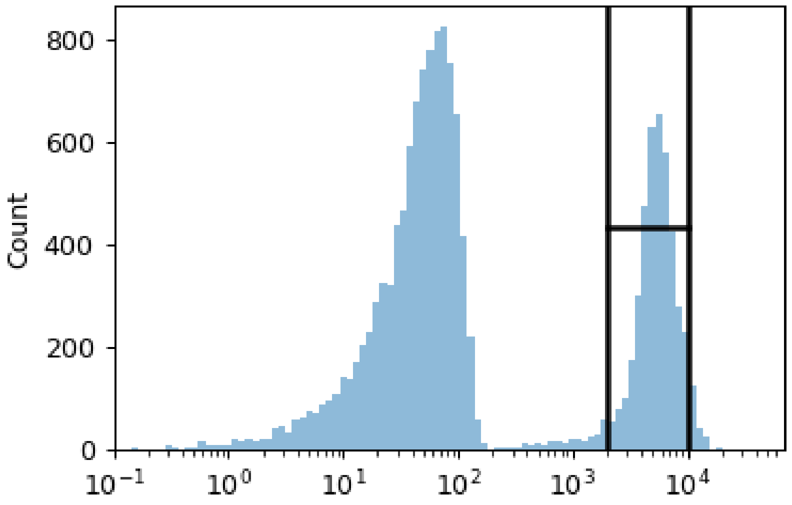
Low
: The low threshold of the gate.

High
: The high threshold of the gate.

Scale
: The scale of the axis for the interactive plot

Hue facet
: Show different experimental conditions in different colors.

Subset
: Show only a subset of the data.



class cytoflowgui.op_plugins.range.**RangeHandler**(**args: Any, **kwargs: Any*)
: Bases: *cytoflowgui.op_plugins.op_plugin_base.OpHandler*

class cytoflowgui.op_plugins.range.**RangeViewHandler**(**args: Any, **kwargs: Any*)
: Bases: *cytoflowgui.view_plugins.view_plugin_base.ViewHandler*

class cytoflowgui.op_plugins.range.**RangePlugin**
: Bases: envisage.plugin.Plugin, *cytoflowgui.op_plugins.op_plugin_base.PluginHelpMixin*

  operation_id = 'edu.mit.synbio.cytoflow.operations.range'

  view_id = 'edu.mit.synbio.cytoflow.views.range'

  short_name = 'Range'

  menu_group = 'Gates'

get_operation()

get_handler(*model*, *context*)

get_icon()

cytoflowgui.op_plugins.range2d

Draw a 2-dimensional range gate (eg, a rectangle). To set the gate, click-and-drag on the plot.

Name
The operation name. Used to name the new metadata field that's created by this operation.

X channel
The name of the channel on the X axis.

X Low
The low threshold in the X channel.

X High
The high threshold in the X channel.

Y channel
The name of the channel on the Y axis.

Y Low
The low threshold in the Y channel.

Y High
The high threshold in the Y channel.

X Scale
The scale of the X axis for the interactive plot.

Y Scale
The scale of the Y axis for the interactive plot

Hue facet
Show different experimental conditions in different colors.

Subset
Show only a subset of the data.

class cytoflowgui.op_plugins.range2d.**Range2DHandler**(**args: Any*, ***kwargs: Any*)
Bases: *cytoflowgui.op_plugins.op_plugin_base.OpHandler*

class cytoflowgui.op_plugins.range2d.**Range2DViewHandler**(**args: Any*, ***kwargs: Any*)
Bases: *cytoflowgui.view_plugins.view_plugin_base.ViewHandler*

class cytoflowgui.op_plugins.range2d.**Range2DPlugin**
Bases: envisage.plugin.Plugin, *cytoflowgui.op_plugins.op_plugin_base.PluginHelpMixin*

class docs

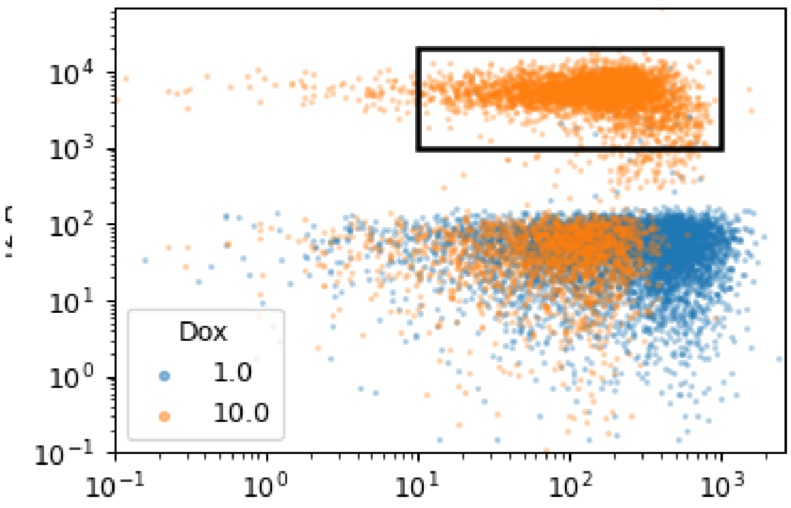
operation_id = 'edu.mit.synbio.cytoflow.operations.range2d'

view_id = 'edu.mit.synbio.cytoflow.views.range2d'

short_name = 'Range 2D'

menu_group = 'Gates'

get_operation()



get_handler(*model*, *context*)

get_icon()

cytoflowgui.op_plugins.ratio

Adds a new "channel" to the workflow, where the value of the channel is the ratio of two other channels.

Name
: The name of the new channel.

Numerator
: The numerator for the ratio.

Denominator
: The denominator for the ratio.

class cytoflowgui.op_plugins.ratio.**RatioHandler**(**args: Any*, ***kwargs: Any*)
: Bases: *cytoflowgui.op_plugins.op_plugin_base.OpHandler*

class cytoflowgui.op_plugins.ratio.**RatioPlugin**
: Bases: envisage.plugin.Plugin, *cytoflowgui.op_plugins.op_plugin_base.PluginHelpMixin*

operation_id = 'edu.mit.synbio.cytoflow.operations.ratio'

view_id = None

short_name = 'Ratio'

menu_group = 'Data'

get_operation()

get_handler(*model*, *context*)

get_icon()

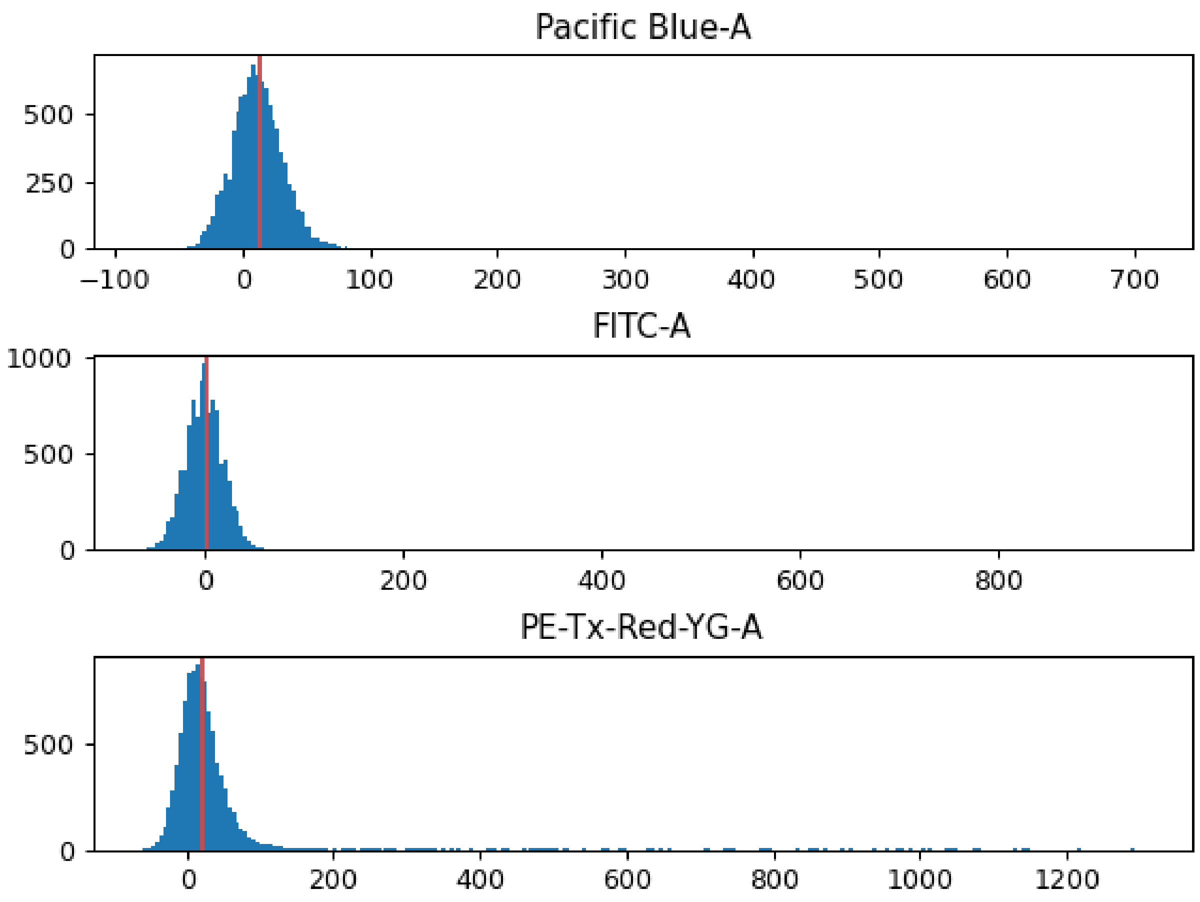
cytoflowgui.op_plugins.tasbe

This module combines all of the other calibrated flow cytometry modules (autofluorescence, bleedthrough compensation, bead calibration, and channel translation) into one easy-use-interface.

`Channels`
Which channels are you calibrating?

`Autofluorescence`

`Blank File`
The FCS file with the blank (unstained or untransformed) cells, for autofluorescence correction.



`Bleedthrough Correction`
A list of single-color controls to use in bleedthrough compensation. There's one entry per channel to compensate.

`Channel`

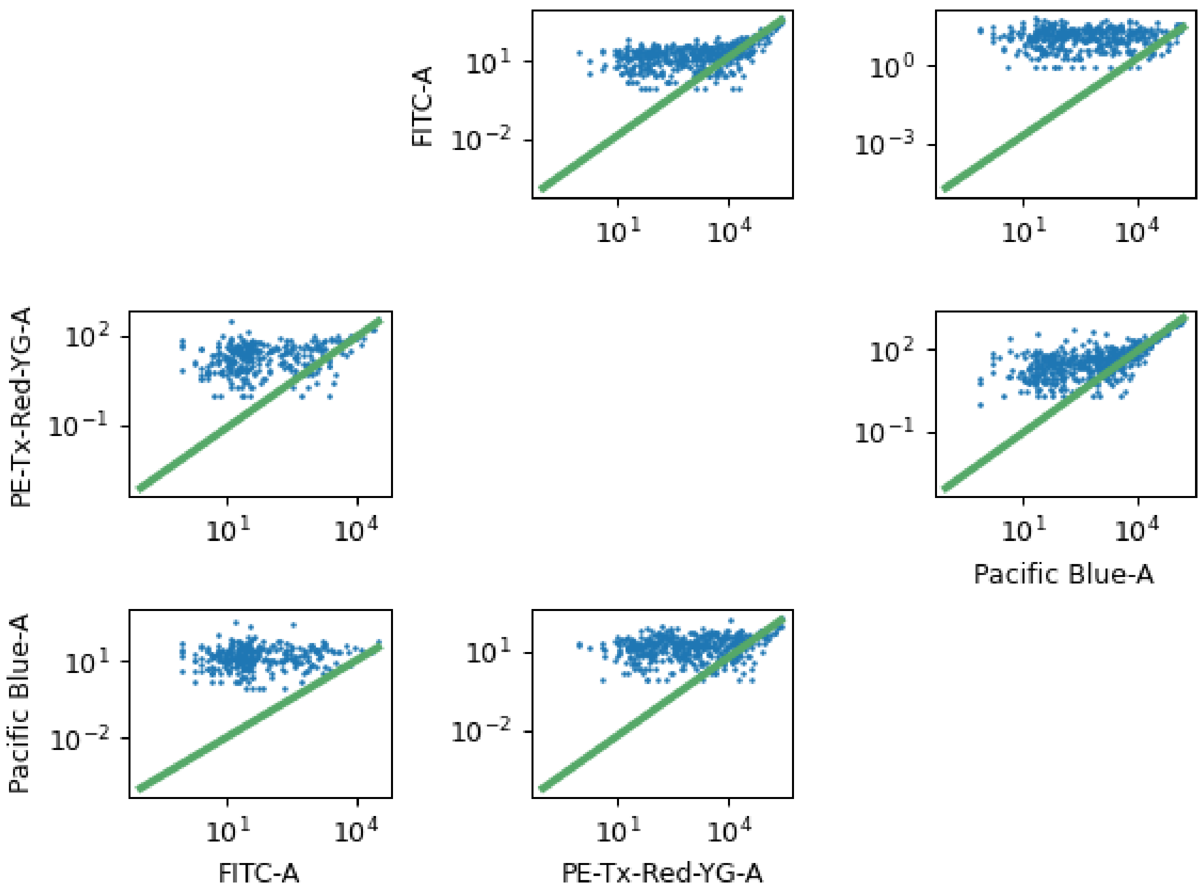
The channel that this file is the single-color control for.

`File`

The FCS file containing the single-color control data.

`Bead Calibration`
The beads that you used for calibration. Make sure to check the lot number as well!

The FCS file containing the bead data.


FITC-A
PE-Tx-Red-YG-A
Pacific Blue-A
FITC-A
PE-Tx-Red-YG-A
Pacific Blue-A

The unit (such as *MEFL*) to calibrate to.

`Peak Quantile`

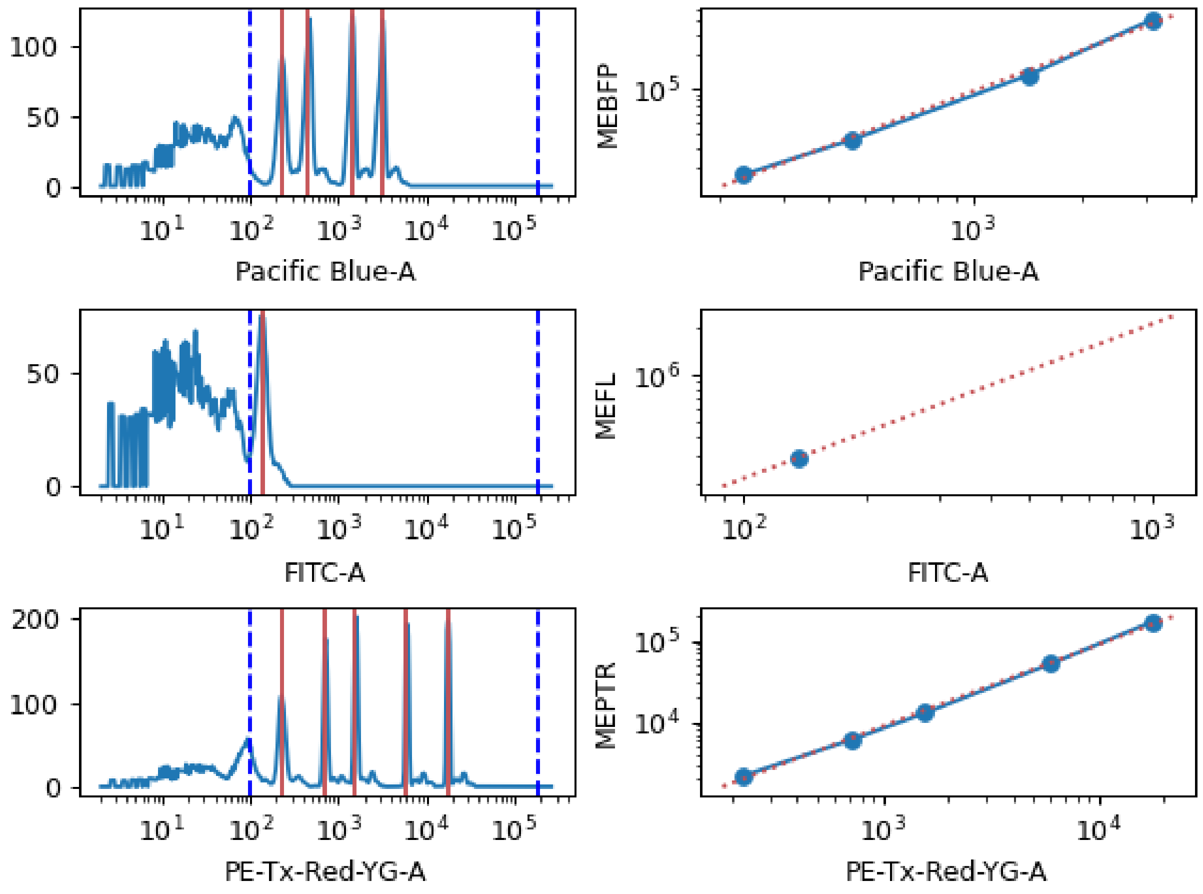
The minimum quantile required to call a peak in the bead data. Check the diagnostic plot: if you have peaks that aren't getting called, decrease this. If you have "noise" peaks that are getting called incorrectly, increase this.

`Peak Threshold`

The minumum brightness where the module will call a peak.

`Peak Cutoff`

The maximum brightness where the module will call a peak. Use this to remove peaks that are saturating the detector.



`Color Translation`

`To Channel`

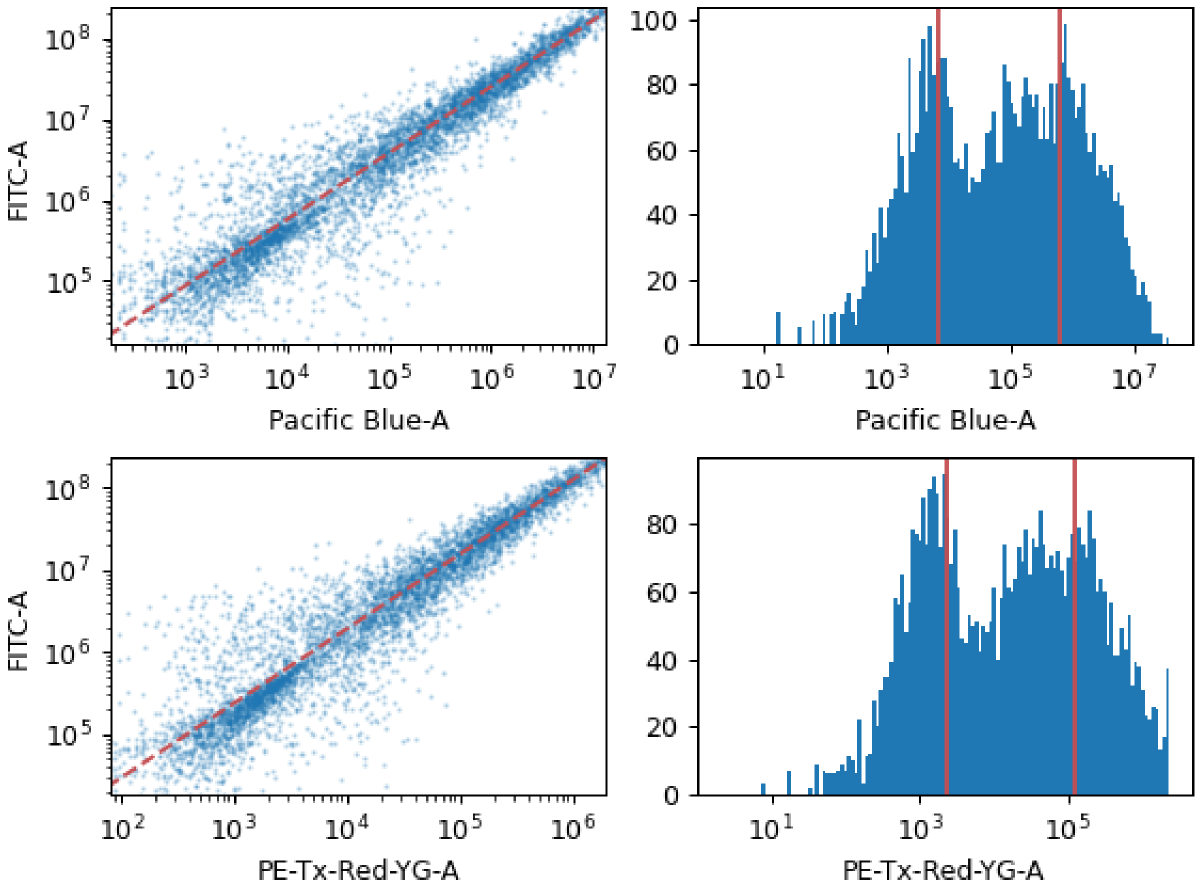
Which channel should we rescale all the other channels to?

`Use mixture model?`

If this is set, the module will try to separate the data using a mixture-of-Gaussians, then only compute the translation using the higher population. This is the kind of behavior that you see in a transient transfection in mammalian cells, for example.

`Translation list`

Each pair of channels must have a multi-color control from which to compute the scaling factor.



class cytoflowgui.op_plugins.tasbe.**BleedthroughControlHandler**(**args: Any*, ***kwargs: Any*)
Bases: `traitsui.api.Controller`

class cytoflowgui.op_plugins.tasbe.**TranslationControlHandler**(**args: Any*, ***kwargs: Any*)
Bases: `traitsui.api.Controller`

class cytoflowgui.op_plugins.tasbe.**UnitHandler**(**args: Any*, ***kwargs: Any*)
Bases: `traitsui.api.Controller`

class cytoflowgui.op_plugins.tasbe.**TasbeHandler**(**args: Any*, ***kwargs: Any*)
Bases: *cytoflowgui.op_plugins.op_plugin_base.OpHandler*

beads_name_choices = <traits.traits.ForwardProperty object>

beads_units = <traits.traits.ForwardProperty object>

class cytoflowgui.op_plugins.tasbe.**TasbeViewHandler**(**args: Any*, ***kwargs: Any*)
Bases: *cytoflowgui.view_plugins.view_plugin_base.ViewHandler*

class cytoflowgui.op_plugins.tasbe.**TasbePlugin**
Bases: `envisage.plugin.Plugin`, *cytoflowgui.op_plugins.op_plugin_base.PluginHelpMixin*

operation_id = 'edu.mit.synbio.cytoflowgui.workflow.operations.tasbe'

view_id = 'edu.mit.synbio.cytoflowgui.workflow.operations.tasbeview'

short_name = 'TASBE Calibration'

menu_group = 'Gates'

get_operation()

get_handler(*model*, *context*)

get_icon()

cytoflowgui.op_plugins.threshold

Draw a threshold gate. To set a new threshold, click on the plot.

Name
The operation name. Used to name the new metadata field that's created by this module.

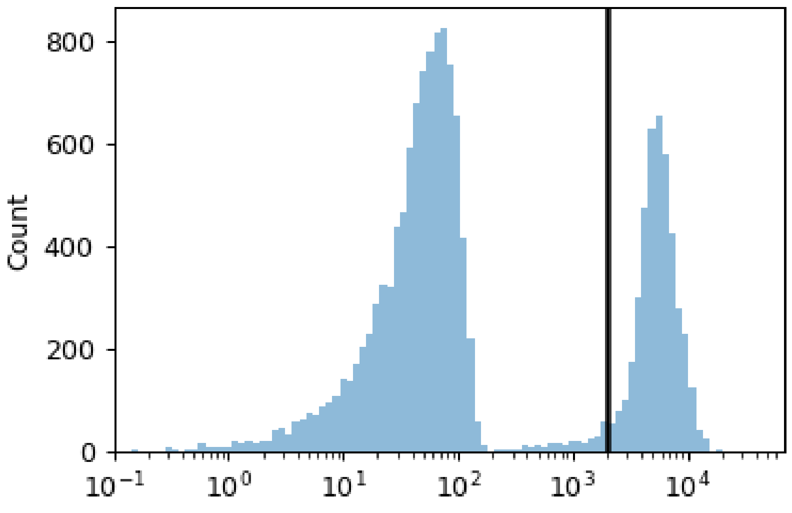
Channel
The name of the channel to apply the gate to.

Threshold
The threshold of the gate.

Scale
The scale of the axis for the interactive plot

Hue facet
Show different experimental conditions in different colors.

Subset
Show only a subset of the data.



class cytoflowgui.op_plugins.threshold.**ThresholdHandler**(**args: Any*, ***kwargs: Any*)
Bases: *cytoflowgui.op_plugins.op_plugin_base.OpHandler*

class cytoflowgui.op_plugins.threshold.**ThresholdViewHandler**(**args: Any*, ***kwargs: Any*)
Bases: *cytoflowgui.view_plugins.view_plugin_base.ViewHandler*

class cytoflowgui.op_plugins.threshold.**ThresholdPlugin**
Bases: envisage.plugin.Plugin, *cytoflowgui.op_plugins.op_plugin_base.PluginHelpMixin*

operation_id = 'edu.mit.synbio.cytoflow.operations.threshold'

`view_id = 'edu.mit.synbio.cytoflow.views.threshold'`

`short_name = 'Threshold'`

`menu_group = 'Gates'`

`get_operation()`

`get_handler(`*model*, *context*`)`

`get_icon()`

cytoflowgui.op_plugins.xform_stat

Apply a function to a statistic, and add it as a statistic to the experiment.

First, the module groups the data by the unique values of the variables in **By**, then applies **Function** to the statistic in each group.

Note: Statistics are a central part of *Cytoflow*. More documentation is forthcoming.

`Name`
The operation name. Becomes the first part of the new statistic's name.

`Statistic`
The statistic to apply the function to.

`Function`
The function to compute on each group.

`Subset`
Only apply the function to a subset of the input statistic. Useful if the function is very slow.

`class cytoflowgui.op_plugins.xform_stat.TransformStatisticHandler(`**args: Any*, ***kwargs: Any*`)`
Bases: *cytoflowgui.op_plugins.op_plugin_base.OpHandler*

`indices = <traits.traits.ForwardProperty object>`

`levels = <traits.traits.ForwardProperty object>`

`class cytoflowgui.op_plugins.xform_stat.TransformStatisticPlugin`
Bases: `envisage.plugin.Plugin`, *cytoflowgui.op_plugins.op_plugin_base.PluginHelpMixin*

`operation_id = 'edu.mit.synbio.cytoflow.operations.transform_statistic'`

`view_id = None`

`short_name = 'Transform Statistic'`

`menu_group = 'Gates'`

`get_operation()`

`get_handler(`*model*, *context*`)`

`get_icon()`

cytoflowgui.utility package

cytoflowgui.utility

A few utility modules to support *cytoflowgui*

Submodules

cytoflowgui.utility.event_tracer

Record trait change events in single and multi-threaded environments. Adapted from https://docs.enthought.com/traits/_modules/traits/util/event_tracer.html

`class cytoflowgui.utility.event_tracer.`**`SentinelRecord`**

Bases: `object`

Sentinel record to separate groups of chained change event dispatches.

`class cytoflowgui.utility.event_tracer.`**`ChangeMessageRecord`**(*time*, *indent*, *name*, *old*, *new*, *class_name*)

Bases: `object`

Message record for a change event dispatch.

`time`
Time stamp in UTC.

`indent`
Depth level in a chain of trait change dispatches.

`name`
The name of the trait that changed

`old`
The old value.

`new`
The new value.

`class_name`
The name of the class that the trait change took place.

`class cytoflowgui.utility.event_tracer.`**`CallingMessageRecord`**(*time*, *indent*, *handler*, *source*)

Bases: `object`

Message record for a change handler call.

`time`
Time stamp in UTC.

`indent`
Depth level of the call in a chain of trait change dispatches.

`handler`
The traits change handler that is called.

`source`
The source file where the handler was defined.

class cytoflowgui.utility.event_tracer.**ExitMessageRecord**(*time*, *indent*, *handler*, *exception*)

Bases: `object`

Message record for returning from a change event dispatch.

time

Time stamp in UTC.

indent

Depth level of the exit in a chain of trait change dispatch.

handler

The traits change handler that is called.

exception

The exception type (if one took place)

class cytoflowgui.utility.event_tracer.**RecordContainer**

Bases: `object`

A simple record container.

This class is commonly used to hold records from a single thread.

record(*record*)

Add the record into the container.

save_to_file(*filename*)

Save the records into a file.

class cytoflowgui.utility.event_tracer.**MultiThreadRecordContainer**

Bases: `object`

A container of record containers that are used by separate threads.

Each record container is mapped to a thread name id. When a RecordContainer does not exist for a specific thread a new empty RecordContainer will be created on request.

get_change_event_collector(*thread_name*)

Return the dedicated RecordContainer for the thread.

If no RecordContainer is found for `thread_name` then a new RecordContainer is created.

save_to_directory(*directory_name*)

Save records files into the directory.

Each RecordContainer will dump its records on a separate file named <thread_name>.trace.

class cytoflowgui.utility.event_tracer.**ChangeEventRecorder**(*container*)

Bases: `object`

A single thread trait change event recorder.

pre_tracer(*obj*, *name*, *old*, *new*, *handler*)

Record a string representation of the trait change dispatch

post_tracer(*obj*, *name*, *old*, *new*, *handler*, *exception=None*)

Record a string representation of the trait change return

class cytoflowgui.utility.event_tracer.**MultiThreadChangeEventRecorder**(*container*)

Bases: `object`

A thread aware trait change recorder.

The class manages multiple ChangeEventRecorders which record trait change events for each thread in a separate file.

close()
Close and stop all logging.

pre_tracer(*obj*, *name*, *old*, *new*, *handler*)
The traits pre event tracer.

This method should be set as the global pre event tracer for traits.

post_tracer(*obj*, *name*, *old*, *new*, *handler*, *exception=None*)
The traits post event tracer.

This method should be set as the global post event tracer for traits.

cytoflowgui.utility.event_tracer.**record_events**()
Multi-threaded trait change event tracer.:

```
from trace_recorder import record_events
with record_events() as change_event_container:
    my_model.some_trait = True
change_event_container.save_to_directory('C:\dev\trace')
```

This will install a tracer that will record all events that occur from setting of some_trait on the my_model instance.

The results will be stored in one file per running thread in the directory 'C:devtrace'. The files are named after the thread being traced.

cytoflowgui.utility.logging

Logging utilities for *cytoflowgui*

class cytoflowgui.utility.logging.**CallbackHandler**(*callback*, ***kwargs*)
Bases: logging.Handler

emit(*record*)
Do whatever it takes to actually log the specified logging record.

This version is intended to be implemented by subclasses and so raises a NotImplementedError.

cytoflowgui.utility.logging.**log_exception**()

cytoflowgui.view_plugins package

cytoflowgui.view_plugins

envisage.plugin.Plugin classes and GUI traitsui.handler.Controller classes to adapt modules from *cytoflowgui.workflow.views* to the Qt / traitsui GUI.

The module docstrings are also the ones that are used for the GUI help panel.

Submodules

cytoflowgui.view_plugins.bar_chart

Plots a bar chart of a statistic.

Each variable in the statistic (ie, each variable chosen in the statistic operation's **Group By**) must be set as **Variable** or as a facet.

`Statistic`
: Which statistic to plot.

`Variable`
: The statistic variable to use as the major bar groups.

`Scale`
: How to scale the statistic plot.

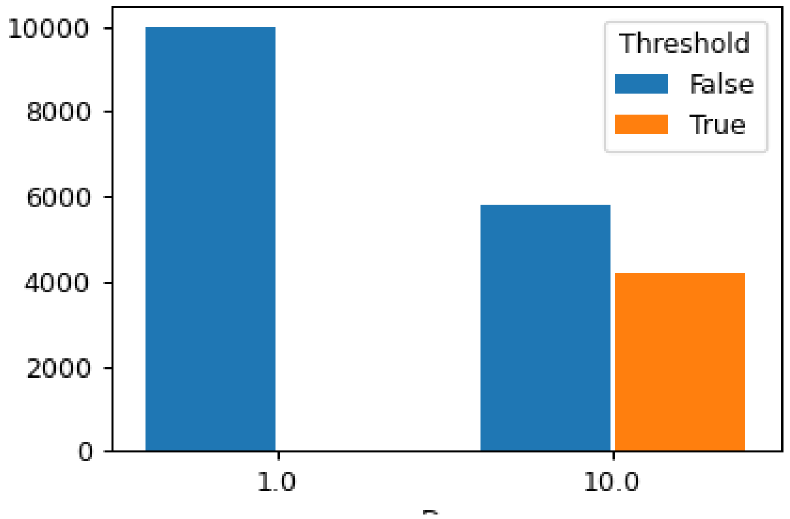
`Horizontal Facet`
: Make muliple plots, with each column representing a subset of the statistic with a different value for this variable.

`Vertical Facet`
: Make multiple plots, with each row representing a subset of the statistic with a different value for this variable.

`Hue Facet`
: Make multiple bars with different colors; each color represents a subset of the statistic with a different value for this variable.

`Error Statistic`
: A statistic to use to make the error bars. Must have the same variables as the statistic in **Statistic**.

`Subset`
: Plot only a subset of the statistic.



class `cytoflowgui.view_plugins.bar_chart.`**`BarChartParamsHandler`**(**args: Any*, ***kwargs: Any*)
: Bases: `traitsui.api.Controller`

class `cytoflowgui.view_plugins.bar_chart.`**`BarChartHandler`**(**args: Any*, ***kwargs: Any*)
: Bases: *`cytoflowgui.view_plugins.view_plugin_base.ViewHandler`*

indices = <traits.traits.ForwardProperty object>

levels = <traits.traits.ForwardProperty object>

class cytoflowgui.view_plugins.bar_chart.**BarChartPlugin**
Bases: envisage.plugin.Plugin, *cytoflowgui.view_plugins.view_plugin_base.PluginHelpMixin*

view_id = 'edu.mit.synbio.cytoflow.view.barchart'

short_name = 'Bar Chart'

get_view()

get_handler(*model*, *context*)

get_icon()

cytoflowgui.view_plugins.density

Plots a 2-dimensional density plot.

X Channel, Y Channel
The channels to plot on the X and Y axes.

X Scale, Y Scale
How to scale the X and Y axes of the plot.

Horizonal Facet
Make multiple plots. Each column has a unique value of this variable.

Vertical Facet
Make multiple plots. Each row has a unique value of this variable.

Color Scale
Scale the color palette and the color bar

Tab Facet
Make multiple plots in differen tabs; each tab's plot has a unique value of this variable.

Subset
Plot only a subset of the data in the experiment.

class cytoflowgui.view_plugins.density.**DensityParamsHandler**(**args: Any*, ***kwargs: Any*)
Bases: traitsui.api.Controller

class cytoflowgui.view_plugins.density.**DensityHandler**(**args: Any*, ***kwargs: Any*)
Bases: *cytoflowgui.view_plugins.view_plugin_base.ViewHandler*

class cytoflowgui.view_plugins.density.**DensityPlugin**
Bases: envisage.plugin.Plugin, *cytoflowgui.view_plugins.view_plugin_base.PluginHelpMixin*

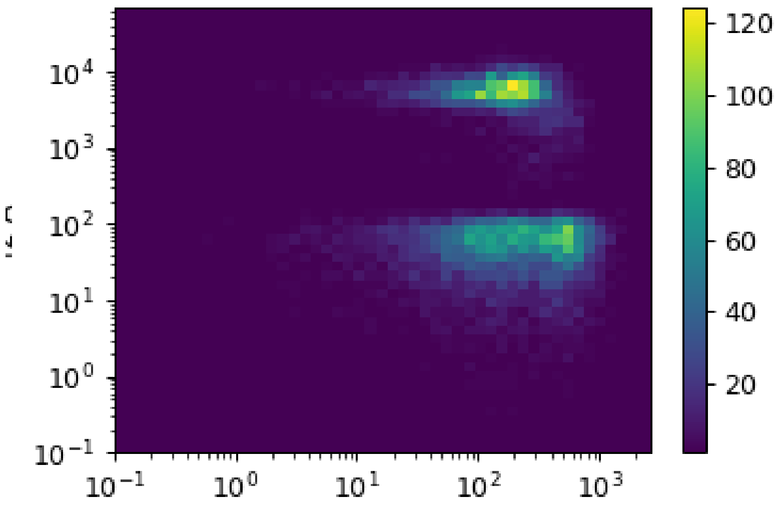
view_id = 'edu.mit.synbio.cytoflow.view.density'

short_name = 'Density Plot'

get_view()

get_handler(*model*, *context*)

get_icon()



cytoflowgui.view_plugins.export_fcs

Exports FCS files from after this operation. Only really useful if you've done a calibration step or created derivative channels using the ratio option. As you set the options, the main plot shows a table of the files that will be created.

`Base`
`The prefix of the FCS file names`

`By`
A list of metadata attributes to aggregate the data before exporting. For example, if the experiment has two pieces of metadata, `Time` and `Dox`, setting **By** to `["Time", "Dox"]` will export one file for each subset of the data with a unique combination of `Time` and `Dox`.

`Keywords`
If you want to add more keywords to the FCS files' TEXT segment, specify them here.

`Export...`
Choose a folder and export the FCS files.

`class cytoflowgui.view_plugins.export_fcs.`**`ExportFCSHandler`**`(*args: Any, **kwargs: Any)`
Bases: *`cytoflowgui.view_plugins.view_plugin_base.ViewHandler`*

`export = <traits.trait_types.Event object>`

`class cytoflowgui.view_plugins.export_fcs.`**`ExportFCSPlugin`**
Bases: `envisage.plugin.Plugin`, *`cytoflowgui.view_plugins.view_plugin_base.PluginHelpMixin`*

`view_id = 'edu.mit.synbio.cytoflow.view.exportfcs'`

`short_name = 'Export FCS'`

`get_view()`

`get_handler``(model, context)`

`get_icon()`

cytoflowgui.view_plugins.histogram

Plots a histogram.

Channel
  The channel for the plot.

Scale
  How to scale the X axis of the plot.

Horizonal Facet
  Make multiple plots. Each column has a unique value of this variable.

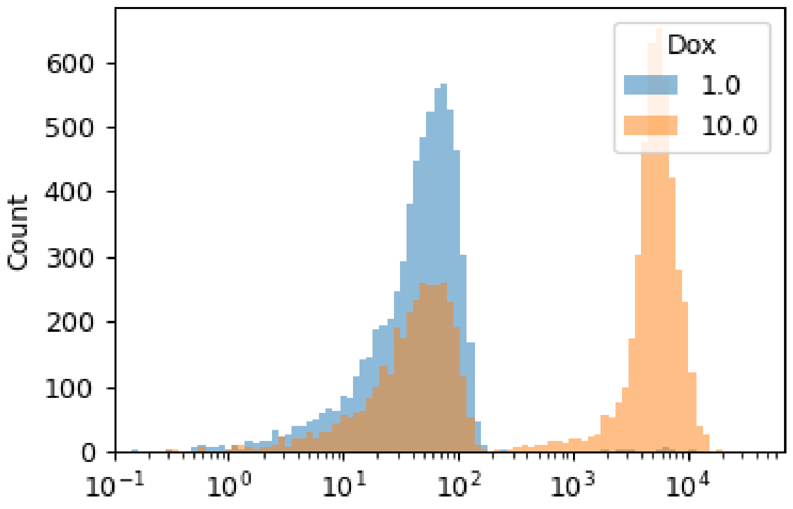
Vertical Facet
  Make multiple plots. Each row has a unique value of this variable.

Color Facet
  Plot with multiple colors. Each color has a unique value of this variable.

Color Scale
  If **Color Facet** is a numeric variable, use this scale for the color bar.

Tab Facet
  Make multiple plots in differen tabs; each tab's plot has a unique value of this variable.

Subset
  Plot only a subset of the data in the experiment.



class `cytoflowgui.view_plugins.histogram.`**`HistogramParamsHandler`**(**args: Any*, ***kwargs: Any*)
  Bases: `traitsui.api.Controller`

class `cytoflowgui.view_plugins.histogram.`**`HistogramHandler`**(**args: Any*, ***kwargs: Any*)
  Bases: *`cytoflowgui.view_plugins.view_plugin_base.ViewHandler`*

class `cytoflowgui.view_plugins.histogram.`**`HistogramPlugin`**
  Bases: `envisage.plugin.Plugin`, *`cytoflowgui.view_plugins.view_plugin_base.PluginHelpMixin`*

  `view_id = 'edu.mit.synbio.cytoflow.view.histogram'`

  `short_name = 'Histogram'`

get_view()

get_handler(*model*, *context*)

get_icon()

cytoflowgui.view_plugins.histogram_2d

Plots a 2-dimensional histogram. Similar to a density plot, but the number of events in a bin change the bin's opacity, so you can use different colors.

X Channel, Y Channel
: The channels to plot on the X and Y axes.

X Scale, Y Scale
: How to scale the X and Y axes of the plot.

Horizonal Facet
: Make multiple plots. Each column has a unique value of this variable.

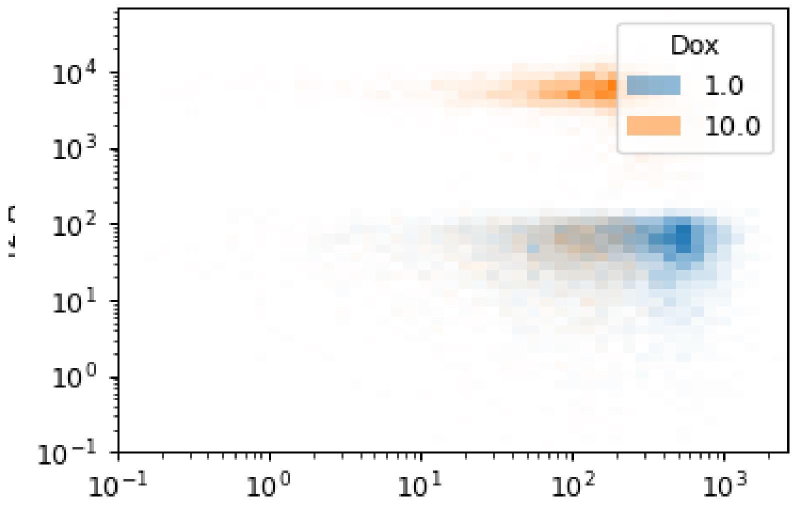
Vertical Facet
: Make multiple plots. Each row has a unique value of this variable.

Color Facet
: Plot with multiple colors. Each color has a unique value of this variable.

Color Scale
: If **Color Facet** is a numeric variable, use this scale for the color bar.

Tab Facet
: Make multiple plots in differen tabs; each tab's plot has a unique value of this variable.

Subset
: Plot only a subset of the data in the experiment.



class cytoflowgui.view_plugins.histogram_2d.**Histogram2DParamsHandler**(**args: Any*, ***kwargs: Any*)

Bases: `traitsui.api.Controller`

class cytoflowgui.view_plugins.histogram_2d.**Histogram2DHandler**(**args: Any*, ***kwargs: Any*)

Bases: *cytoflowgui.view_plugins.view_plugin_base.ViewHandler*

class cytoflowgui.view_plugins.histogram_2d.**Histogram2DPlugin**

Bases: envisage.plugin.Plugin, *cytoflowgui.view_plugins.view_plugin_base.PluginHelpMixin*

view_id = 'edu.mit.synbio.cytoflow.view.histogram2d'

short_name = '2D Histogram'

get_view()

get_handler(*model*, *context*)

get_icon()

cytoflowgui.view_plugins.i_view_plugin

class cytoflowgui.view_plugins.i_view_plugin.**IViewPlugin**(*adaptee*, *default=<class 'traits.adaptation.adaptation_error.AdaptationError'>*)

Bases: traits.has_traits.Interface

id

The envisage ID used to refer to this plugin

Type Str

view_id

Same as the "id" attribute of the IView this plugin wraps Prefix: edu.mit.synbio.cytoflowgui.view

Type Str

short_name

The view's "short" name - for menus, toolbar tips, etc.

Type Str

get_view()

Gets the IView instance that this plugin wraps.

Returns An instance of the view that this plugin wraps

Return type *IView*

get_handler(*model*)

Gets an instance of the handler for this view or params model.

NOTE: You have to check to see what the class of `model` is, and return an appropriate handler!

Parameters

- **model** (*IWorkflowView*) – The model to associate with this handler.
- **context** (*WorkflowItem*) – The WorkflowItem that this model is a part of.

Return type traitsui.handler.Controller

get_icon()

Returns an icon for this plugin

Returns The SVG icon

Return type pyface.i_image_resource.IImageResource

get_help()
Gets the HTML help text for this plugin, deriving the filename from the class name. Probably best to use the default implementation in *cytoflowgui.view_plugins.view_plugin_base.PluginHelpMixin*

Returns The HTML help, in a single string.

Return type string

get_plugin()
Returns an instance of `envisage.plugin.Plugin` implementing *IViewPlugin*. Usually returns `self`.

Return type `envisage.plugin.Plugin`

class cytoflowgui.view_plugins.i_view_plugin.**ViewPluginManager**
Bases: `envisage.plugin.Plugin`

cytoflowgui.view_plugins.kde_1d

Plots a "smoothed" histogram.

Channel
The channel for the plot.

Scale
How to scale the X axis of the plot.

Horizonal Facet
Make multiple plots. Each column has a unique value of this variable.

Vertical Facet
Make multiple plots. Each row has a unique value of this variable.

Color Facet
Plot with multiple colors. Each color has a unique value of this variable.

Color Scale
If **Color Facet** is a numeric variable, use this scale for the color bar.

Tab Facet
Make multiple plots in differen tabs; each tab's plot has a unique value of this variable.

Subset
Plot only a subset of the data in the experiment.

class cytoflowgui.view_plugins.kde_1d.**Kde1DParamsHandler**(**args: Any*, ***kwargs: Any*)
Bases: `traitsui.api.Controller`

class cytoflowgui.view_plugins.kde_1d.**Kde1DHandler**(**args: Any*, ***kwargs: Any*)
Bases: *cytoflowgui.view_plugins.view_plugin_base.ViewHandler*, `traitsui.api.Controller`

class cytoflowgui.view_plugins.kde_1d.**Kde1DPlugin**
Bases: `envisage.plugin.Plugin`, *cytoflowgui.view_plugins.view_plugin_base.PluginHelpMixin*

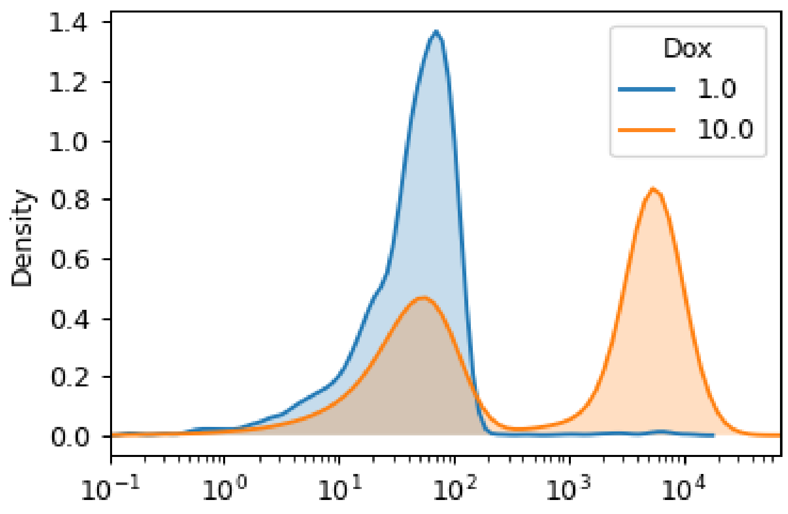
view_id = 'edu.mit.synbio.cytoflow.view.kde1d'

short_name = '1D Kernel Density'

get_view()

get_handler(*model*, *context*)

get_icon()



cytoflowgui.view_plugins.kde_2d

Plots a 2-d kernel-density estimate. Sort of like a smoothed histogram. The density is visualized with a set of isolines.

`X Channel, Y Channel`
The channels to plot on the X and Y axes.

`X Scale, Y Scale`
How to scale the X and Y axes of the plot.

`Horizonal Facet`
Make multiple plots. Each column has a unique value of this variable.

`Vertical Facet`
Make multiple plots. Each row has a unique value of this variable.

`Color Facet`
Plot with multiple colors. Each color has a unique value of this variable.

`Color Scale`
If **Color Facet** is a numeric variable, use this scale for the color bar.

`Tab Facet`
Make multiple plots in differen tabs; each tab's plot has a unique value of this variable.

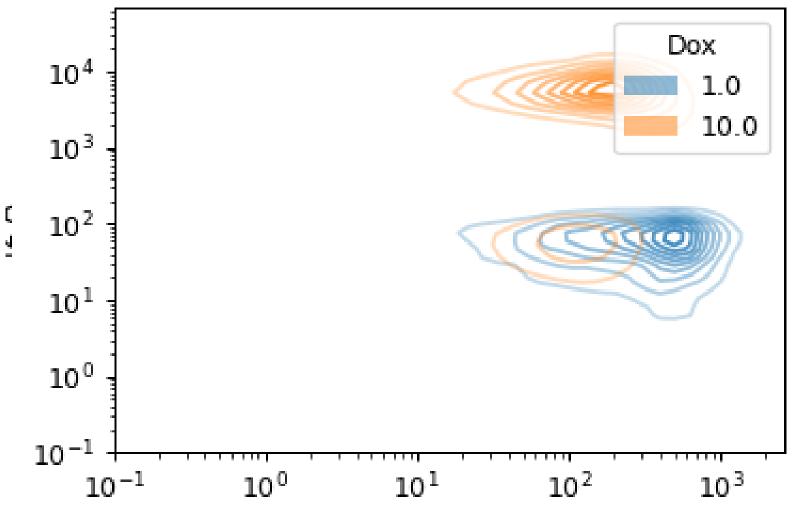
`Subset`
Plot only a subset of the data in the experiment.

class `cytoflowgui.view_plugins.kde_2d.`**`Kde2DParamsHandler`**(**args: Any*, ***kwargs: Any*)
Bases: `traitsui.api.Controller`

class `cytoflowgui.view_plugins.kde_2d.`**`Kde2DHandler`**(**args: Any*, ***kwargs: Any*)
Bases: *`cytoflowgui.view_plugins.view_plugin_base.ViewHandler`*

class `cytoflowgui.view_plugins.kde_2d.`**`Kde2DPlugin`**
Bases: `envisage.plugin.Plugin`, *`cytoflowgui.view_plugins.view_plugin_base.PluginHelpMixin`*

`view_id = 'edu.mit.synbio.cytoflow.view.kde2d'`



`short_name = '2D Kernel Density Estimate'`

get_view()

get_handler(*model*, *context*)

get_icon()

cytoflowgui.view_plugins.parallel_coords

Plots a parallel coordinates plot. PC plots are good for multivariate data; each vertical line represents one attribute, and one set of connected line segments represents one data point.

Channels
The channels to plot, and their scales. Drag the blue dot to re-order.

Add Channel, Remove Channel
Add or remove a channel

Horizonal Facet
Make multiple plots. Each column has a unique value of this variable.

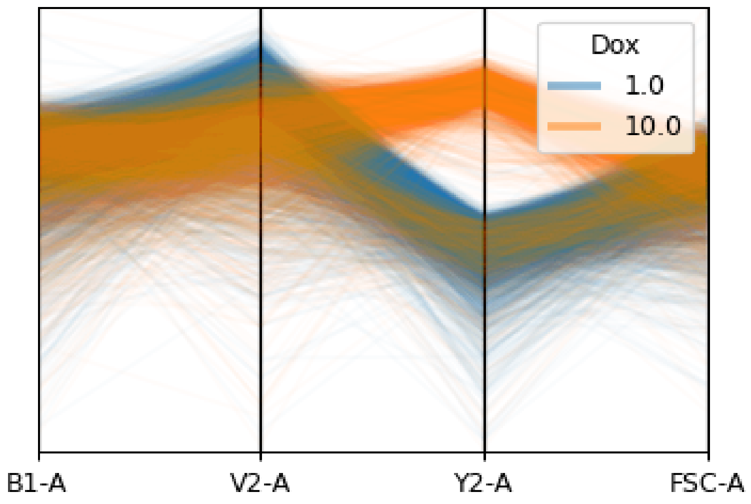
Vertical Facet
Make multiple plots. Each row has a unique value of this variable.

Color Facet
Plot different values of a condition with different colors.

Color Scale
Scale the color palette and the color bar

Tab Facet
Make multiple plots in differen tabs; each tab's plot has a unique value of this variable.

Subset
Plot only a subset of the data in the experiment.



class cytoflowgui.view_plugins.parallel_coords.**ParallelCoordinatesParamsHandler**(**args: Any*, ***kwargs: Any*)

Bases: traitsui.api.Controller

class cytoflowgui.view_plugins.parallel_coords.**ChannelHandler**(**args: Any*, ***kwargs: Any*)

Bases: traitsui.api.Controller

class cytoflowgui.view_plugins.parallel_coords.**ParallelCoordinatesHandler**(**args: Any*, ***kwargs: Any*)

Bases: *cytoflowgui.view_plugins.view_plugin_base.ViewHandler*

add_channel

alias of traits.trait_types.Event

remove_channel

alias of traits.trait_types.Event

channels = <traits.traits.ForwardProperty object>

class cytoflowgui.view_plugins.parallel_coords.**ParallelCoordinatesPlugin**

Bases: envisage.plugin.Plugin, *cytoflowgui.view_plugins.view_plugin_base.PluginHelpMixin*

view_id = 'edu.mit.synbio.cytoflow.view.parallel_coords'

short_name = 'Parallel Coordinates Plot'

get_view()

get_handler(*model*, *context*)

get_icon()

cytoflowgui.view_plugins.radviz

Plots a radviz plot. Radviz plots project multivariate plots into two dimensions. Good for looking for clusters.

`Channels`
: The channels to plot, and their scales. Drag the blue dot to re-order.

`Add Channel, Remove Channel`
: Add or remove a channel

`Horizonal Facet`
: Make multiple plots. Each column has a unique value of this variable.

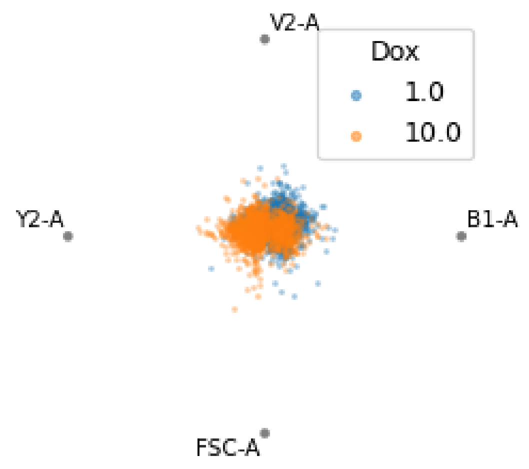
`Vertical Facet`
: Make multiple plots. Each row has a unique value of this variable.

`Color Facet`
: Plot different values of a condition with different colors.

`Color Scale`
: Scale the color palette and the color bar

`Tab Facet`
: Make multiple plots in differen tabs; each tab's plot has a unique value of this variable.

`Subset`
: Plot only a subset of the data in the experiment.



class `cytoflowgui.view_plugins.radviz.`**`RadvizParamsHandler`**(**args: Any*, ***kwargs: Any*)
: Bases: `traitsui.api.Controller`

class `cytoflowgui.view_plugins.radviz.`**`ChannelHandler`**(**args: Any*, ***kwargs: Any*)
: Bases: `traitsui.api.Controller`

class `cytoflowgui.view_plugins.radviz.`**`RadvizHandler`**(**args: Any*, ***kwargs: Any*)
: Bases: *`cytoflowgui.view_plugins.view_plugin_base.ViewHandler`*

    `add_channel`
    : alias of `traits.trait_types.Event`

remove_channel
alias of `traits.trait_types.Event`

channels = <traits.traits.ForwardProperty object>

class `cytoflowgui.view_plugins.radviz.`**RadvizPlugin**
Bases: `envisage.plugin.Plugin`, *`cytoflowgui.view_plugins.view_plugin_base.PluginHelpMixin`*

view_id = 'edu.mit.synbio.cytoflow.view.radviz'

short_name = 'Radviz Plot'

get_view()

get_handler(*model*, *context*)

get_icon()

cytoflowgui.view_plugins.scatterplot

Plot a scatterplot.

X Channel, Y Channel
The channels to plot on the X and Y axes.

X Scale, Y Scale
How to scale the X and Y axes of the plot.

Horizonal Facet
Make multiple plots. Each column has a unique value of this variable.

Vertical Facet
Make multiple plots. Each row has a unique value of this variable.

Color Facet
Plot with multiple colors. Each color has a unique value of this variable.

Color Scale
If **Color Facet** is a numeric variable, use this scale for the color bar.

Tab Facet
Make multiple plots in differen tabs; each tab's plot has a unique value of this variable.

Subset
Plot only a subset of the data in the experiment.

class `cytoflowgui.view_plugins.scatterplot.`**ScatterplotParamsHandler**(**args: Any*, ***kwargs: Any*)
Bases: `traitsui.api.Controller`

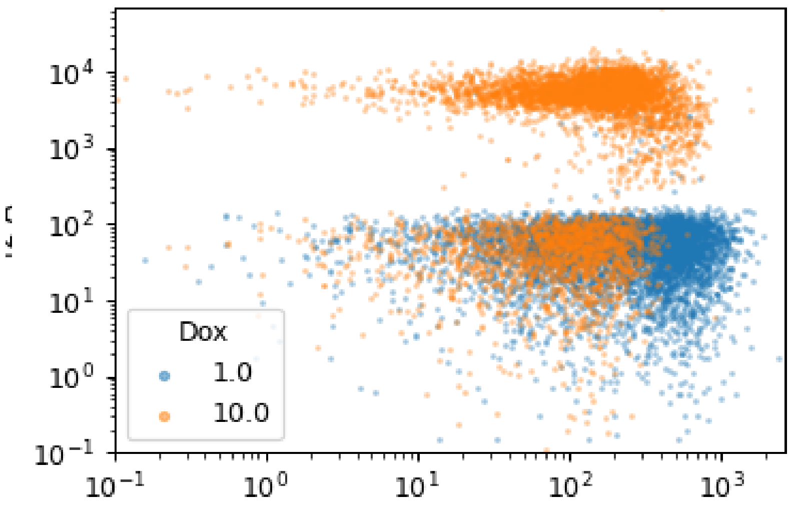
class `cytoflowgui.view_plugins.scatterplot.`**ScatterplotHandler**(**args: Any*, ***kwargs: Any*)
Bases: *`cytoflowgui.view_plugins.view_plugin_base.ViewHandler`*

class `cytoflowgui.view_plugins.scatterplot.`**ScatterplotPlugin**
Bases: `envisage.plugin.Plugin`, *`cytoflowgui.view_plugins.view_plugin_base.PluginHelpMixin`*

view_id = 'edu.mit.synbio.cytoflow.view.scatterplot'

short_name = 'Scatter Plot'

get_view()



get_handler(*model*, *context*)

get_icon()

cytoflowgui.view_plugins.stats_1d

Plots a line plot of a statistic.

Each variable in the statistic (ie, each variable chosen in the statistic operation's **Group By**) must be set as **Variable** or as a facet.

Statistic
: Which statistic to plot.

Variable
: The statistic variable put on the X axis. Must be numeric.

X Scale, Y Scale
: How to scale the X and Y axes.

Horizontal Facet
: Make muliple plots, with each column representing a subset of the statistic with a different value for this variable.

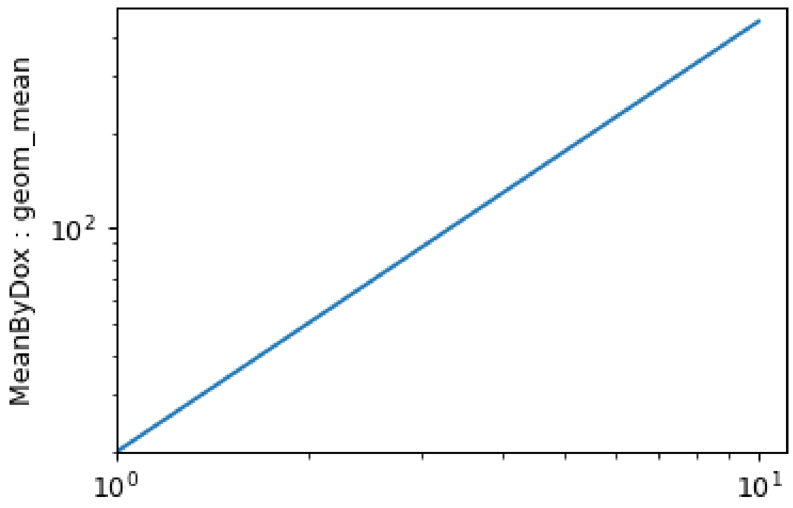
Vertical Facet
: Make multiple plots, with each row representing a subset of the statistic with a different value for this variable.

Hue Facet
: Make multiple bars with different colors; each color represents a subset of the statistic with a different value for this variable.

Color Scale
: If **Color Facet** is a numeric variable, use this scale for the color bar.

Error Statistic
: A statistic to use to make the error bars. Must have the same variables as the statistic in **Statistic**.

Subset
: Plot only a subset of the statistic.



class cytoflowgui.view_plugins.stats_1d.**Stats1DParamsHandler**(**args: Any*, ***kwargs: Any*)

Bases: traitsui.api.Controller

class cytoflowgui.view_plugins.stats_1d.**Stats1DHandler**(**args: Any*, ***kwargs: Any*)

Bases: *cytoflowgui.view_plugins.view_plugin_base.ViewHandler*

indices = <traits.traits.ForwardProperty object>

numeric_indices = <traits.traits.ForwardProperty object>

levels = <traits.traits.ForwardProperty object>

class cytoflowgui.view_plugins.stats_1d.**Stats1DPlugin**

Bases: envisage.plugin.Plugin, *cytoflowgui.view_plugins.view_plugin_base.PluginHelpMixin*

view_id = 'edu.mit.synbio.cytoflow.view.stats1d'

short_name = '1D Statistics View'

get_view()

get_handler(*model*, *context*)

get_icon()

cytoflowgui.view_plugins.stats_2d

Plot two statistics on a scatter plot. A point (X,Y) is drawn for every pair of elements with the same value of **Variable**; the X value is from ** X statistic** and the Y value is from **Y statistic**.

X Statistic
Which statistic to plot on the X axis.

Y Statistic
Which statistic to plot on the Y axis. Must have the same indices as **X Statistic**.

X Scale, Y Scale
How to scale the X and Y axes.

Variable
The statistic variable to put on the plot.

Horizontal Facet
Make muliple plots, with each column representing a subset of the statistic with a different value for this variable.

Vertical Facet
Make multiple plots, with each row representing a subset of the statistic with a different value for this variable.

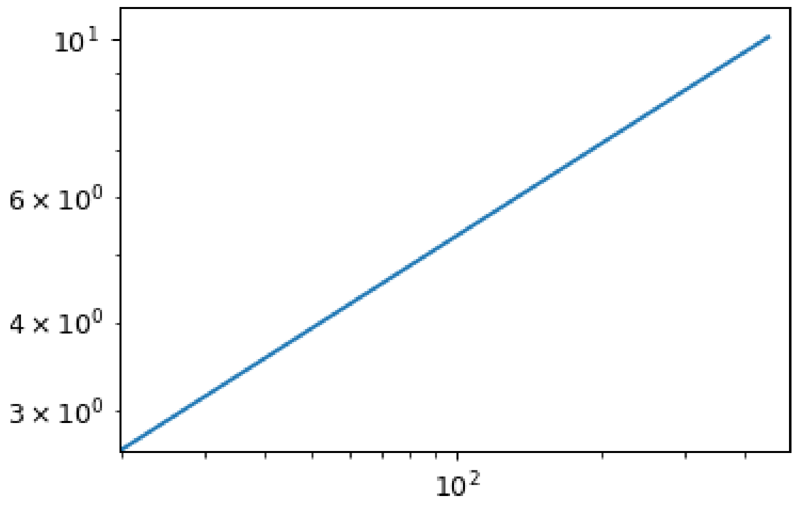
Color Facet
Make lines on the plot with different colors; each color represents a subset of the statistic with a different value for this variable.

Color Scale
If **Color Facet** is a numeric variable, use this scale for the color bar.

X Error Statistic
A statistic to use to make error bars in the X direction. Must have the same indices as the statistic in **X Statistic**.

Y Error Statistic
A statistic to use to make error bars in the Y direction. Must have the same indices as the statistic in **Y Statistic**.

Subset
Plot only a subset of the statistic.



class `cytoflowgui.view_plugins.stats_2d.`**`Stats2DParamsHandler`**(**args: Any*, ***kwargs: Any*)
Bases: `traitsui.api.Controller`

class `cytoflowgui.view_plugins.stats_2d.`**`Stats2DHandler`**(**args: Any*, ***kwargs: Any*)
Bases: *`cytoflowgui.view_plugins.view_plugin_base.ViewHandler`*

`indices = <traits.traits.ForwardProperty object>`

`numeric_indices = <traits.traits.ForwardProperty object>`

`levels = <traits.traits.ForwardProperty object>`

class `cytoflowgui.view_plugins.stats_2d.`**`Stats2DPlugin`**
Bases: `envisage.plugin.Plugin`, *`cytoflowgui.view_plugins.view_plugin_base.PluginHelpMixin`*

view_id = 'edu.mit.synbio.cytoflow.view.stats2d'

short_name = '2D Statistics View'

get_view()

get_handler(*model*, *context*)

get_icon()

cytoflowgui.view_plugins.table

Make a table out of a statistic. The table can then be exported.

Statistic
Which statistic to view.

Rows
Which variable to use for the rows

Subrows
Which variable to use for subrows.

Columns
Which variable to use for the columns.

Subcolumns
Which variable to use for the subcolumns.

Export
Export the table to a CSV file.

class cytoflowgui.view_plugins.table.**TableHandler**(**args: Any*, ***kwargs: Any*)
Bases: *cytoflowgui.view_plugins.view_plugin_base.ViewHandler*

indices = <traits.traits.ForwardProperty object>

levels = <traits.traits.ForwardProperty object>

export = <traits.trait_types.Event object>

class cytoflowgui.view_plugins.table.**TablePlugin**
Bases: envisage.plugin.Plugin, *cytoflowgui.view_plugins.view_plugin_base.PluginHelpMixin*

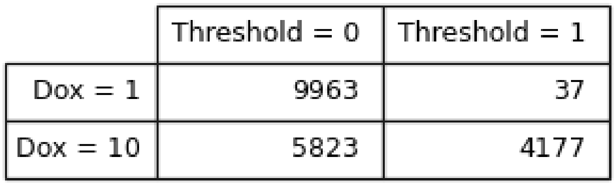
view_id = 'edu.mit.synbio.cytoflow.view.table'

short_name = 'Table View'

get_view()

get_handler(*model*, *context*)

get_icon()



	Threshold = 0	Threshold = 1
Dox = 1	9963	37
Dox = 10	5823	4177

cytoflowgui.view_plugins.view_plugin_base

@author: brian

`class cytoflowgui.view_plugins.view_plugin_base.`**`PluginHelpMixin`**

Bases: `traits.has_traits.HasTraits`

A mixin to get online HTML help for a class. It determines the HTML path name from the class name.

`get_help()`

Gets the HTML help for this class.

Returns The HTML help in a single string.

Return type string

`class cytoflowgui.view_plugins.view_plugin_base.`**`ViewHandler`**`(*args: Any, **kwargs: Any)`

Bases: `traitsui.api.Controller`

Useful bits for view handlers.

`context = <traits.trait_types.Instance object>`

cytoflowgui.view_plugins.violin

Plots a violin plot, which is a nice way to compare several distributions.

`X Variable`
The variable to compare on the X axis.

`Y Channel`
The channel to plot on the Y axis.

`Y Channel Scale`
How to scale the Y axis of the plot.

`Horizonal Facet`
Make multiple plots. Each column has a unique value of this variable.

`Vertical Facet`
Make multiple plots. Each row has a unique value of this variable.

`Color Facet`
Plot with multiple colors. Each color has a unique value of this variable.

`Color Scale`
If **Color Facet** is a numeric variable, use this scale for the color bar.

`Tab Facet`
Make multiple plots in differen tabs; each tab's plot has a unique value of this variable.

`Subset`
Plot only a subset of the data in the experiment.

`class cytoflowgui.view_plugins.violin.`**`ViolinParamsHandler`**`(*args: Any, **kwargs: Any)`

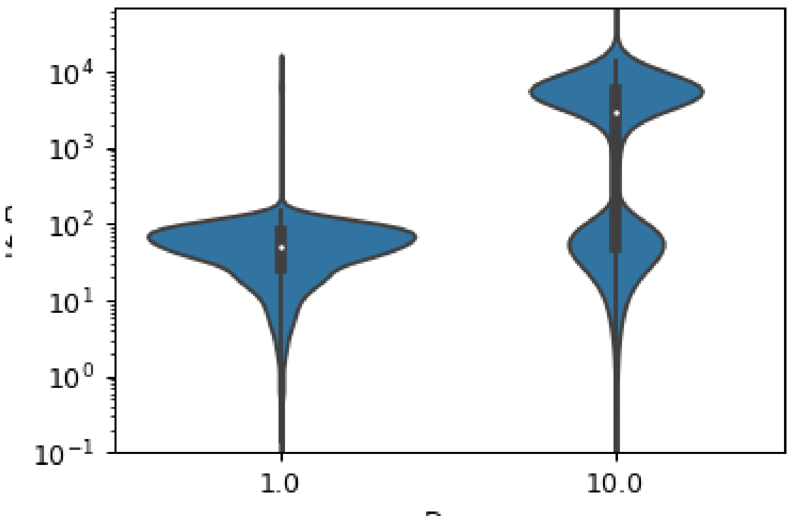
Bases: `traitsui.api.Controller`

`class cytoflowgui.view_plugins.violin.`**`ViolinHandler`**`(*args: Any, **kwargs: Any)`

Bases: *`cytoflowgui.view_plugins.view_plugin_base.ViewHandler`*

`class cytoflowgui.view_plugins.violin.`**`ViolinPlotPlugin`**

Bases: `envisage.plugin.Plugin`, *`cytoflowgui.view_plugins.view_plugin_base.PluginHelpMixin`*



view_id = 'edu.mit.synbio.cytoflow.view.violin'

short_name = 'Violin Plot'

get_view()

get_handler(*model*, *context*)

get_icon()

cytoflowgui.workflow package

cytoflowgui.workflow

class cytoflowgui.workflow.**Changed**

Bases: object

APPLY = 'APPLY'

ESTIMATE = 'ESTIMATE'

VIEW = 'VIEW'

ESTIMATE_RESULT = 'ESTIMATE_RESULT'

OP_STATUS = 'OP_STATUS'

RESULT = 'RESULT'

PREV_RESULT = 'PREV_RESULT'

Subpackages

cytoflowgui.workflow.operations package

cytoflowgui.workflow.operations

Submodules

cytoflowgui.workflow.operations.autofluorescence

class cytoflowgui.workflow.operations.autofluorescence.**AutofluorescenceWorkflowOp**

Bases: *cytoflowgui.workflow.operations.operation_base.WorkflowOperation*, *cytoflow.operations.autofluorescence.AutofluorescenceOp*

default_view(***kwargs*)

Returns a diagnostic plot to see if the autofluorescence estimation is working.

Returns An diagnostic view, call `AutofluorescenceDiagnosticView.plot` to see the diagnostic plots

Return type *IView*

estimate(*experiment*)

Estimate the autofluorescence from `blank_file` in channels specified in `channels`.

Parameters

- **experiment** (`Experiment`) – The experiment to which this operation is applied
- **subset** (*Str (default = "")*) – An expression that specifies the events used to compute the autofluorescence

apply(*experiment*)

Applies the autofluorescence correction to channels in an experiment.

Parameters **experiment** (`Experiment`) – the experiment to which this op is applied

Returns

a new experiment with the autofluorescence median subtracted. The corrected channels have the following metadata added to them:

- **af_median** : Float The median of the non-fluorescent distribution
- **af_stdev** : Float The standard deviation of the non-fluorescent distribution

Return type *Experiment*

clear_estimate()

should_apply(*changed*, *payload*)

should_clear_estimate(*changed*, *payload*)

get_notebook_code(*idx*)

class cytoflowgui.workflow.operations.autofluorescence.**AutofluorescenceWorkflowView**

Bases: *cytoflowgui.workflow.views.view_base.WorkflowView*, *cytoflow.operations.autofluorescence.AutofluorescenceDiagnosticView*

should_plot(*changed*, *payload*)

Should the owning WorkflowItem refresh the plot when certain things change? `changed` can be: - Changed.VIEW – the view's parameters changed - Changed.RESULT – this WorkflowItem's result changed - Changed.PREV_RESULT – the previous WorkflowItem's result changed - Changed.ESTIMATE_RESULT – the results of calling "estimate" changed

If *should_plot* is called from a notification handler, the payload is the handler `event` parameter.

get_notebook_code(*idx*)

cytoflowgui.workflow.operations.bead_calibration

class cytoflowgui.workflow.operations.bead_calibration.**Unit**

Bases: `traits.has_traits.HasTraits`

class cytoflowgui.workflow.operations.bead_calibration.**BeadCalibrationWorkflowOp**

Bases: *cytoflowgui.workflow.operations.operation_base.WorkflowOperation*, *cytoflow.operations.bead_calibration.BeadCalibrationOp*

default_view(***kwargs*)

Returns a diagnostic plot to see if the peak finding is working.

Returns An diagnostic view, call *BeadCalibrationDiagnostic.plot* to see the diagnostic plots

Return type *IView*

apply(*experiment*)

Applies the bleedthrough correction to an experiment.

Parameters **experiment** (*Experiment*) – the experiment to which this operation is applied

Returns

A new experiment with the specified channels calibrated in physical units. The calibrated channels also have new metadata:

- **bead_calibration_fn** [Callable (`pandas.Series` –> `pandas.Series`)] The function to calibrate raw data to bead units
- **bead_units** [Str] The units this channel was calibrated to

Return type *Experiment*

estimate(*experiment*)

Estimate the calibration coefficients from the beads file.

Parameters **experiment** (*Experiment*) – The experiment used to compute the calibration.

should_clear_estimate(*changed*, *payload*)

clear_estimate()

get_notebook_code(*idx*)

class cytoflowgui.workflow.operations.bead_calibration.**BeadCalibrationWorkflowView**

Bases: *cytoflowgui.workflow.views.view_base.WorkflowView*, *cytoflow.operations.bead_calibration.BeadCalibrationDiagnostic*

should_plot(*changed*, *payload*)

Should the owning WorkflowItem refresh the plot when certain things change? `changed` can be: - Changed.VIEW – the view's parameters changed - Changed.RESULT – this WorkflowItem's result changed - Changed.PREV_RESULT – the previous WorkflowItem's result changed - Changed.ESTIMATE_RESULT – the results of calling "estimate" changed

If *should_plot* is called from a notification handler, the payload is the handler `event` parameter.

get_notebook_code(*idx*)

cytoflowgui.workflow.operations.binning

class cytoflowgui.workflow.operations.binning.**BinningWorkflowOp**

Bases: *cytoflowgui.workflow.operations.operation_base.WorkflowOperation*, *cytoflow.operations.binning.BinningOp*

default_view(***kwargs*)

Returns a diagnostic plot to check the binning.

Returns An view instance, call *plot()* to plot the bins.

Return type *IView*

clear_estimate()

get_notebook_code(*idx*)

class cytoflowgui.workflow.operations.binning.**BinningWorkflowView**

Bases: *cytoflowgui.workflow.views.view_base.WorkflowFacetView*, *cytoflow.operations.binning.BinningView*

get_notebook_code(*idx*)

cytoflowgui.workflow.operations.bleedthrough_linear

class cytoflowgui.workflow.operations.bleedthrough_linear.**Control**

Bases: `traits.has_traits.HasTraits`

class cytoflowgui.workflow.operations.bleedthrough_linear.**BleedthroughLinearWorkflowOp**

Bases: *cytoflowgui.workflow.operations.operation_base.WorkflowOperation*, *cytoflow.operations.bleedthrough_linear.BleedthroughLinearOp*

default_view(***kwargs*)

Returns a diagnostic plot to make sure spillover estimation is working.

Returns An *IView*, call *BleedthroughLinearDiagnostic.plot* to see the diagnostic plots

Return type *IView*

estimate(*experiment*)

Estimate the bleedthrough from simgle-channel controls in *controls*

apply(*experiment*)

Applies the bleedthrough correction to an experiment.

Parameters **experiment** (*Experiment*) – The experiment to which this operation is applied

Returns

A new *Experiment* with the bleedthrough subtracted out. The corrected channels have the following metadata added:

- **linear_bleedthrough** : Dict(Str : Float) The values for spillover from other channels into this channel.

- **bleedthrough_channels** : List(Str) The channels that were used to correct this one.
- **bleedthrough_fn** : Callable (Tuple(Float) –> Float) The function that will correct one event in this channel. Pass it the values specified in *controls* and it will return the corrected value for this channel.

Return type *Experiment*

should_clear_estimate(*changed*, *payload*)

clear_estimate()

get_notebook_code(*idx*)

class cytoflowgui.workflow.operations.bleedthrough_linear.**BleedthroughLinearWorkflowView**

Bases: *cytoflowgui.workflow.views.view_base.WorkflowView*, *cytoflow.operations.bleedthrough_linear.BleedthroughLinearDiagnostic*

should_plot(*changed*, *payload*)

Should the owning WorkflowItem refresh the plot when certain things change? `changed` can be: - Changed.VIEW – the view's parameters changed - Changed.RESULT – this WorkflowItem's result changed - Changed.PREV_RESULT – the previous WorkflowItem's result changed - Changed.ESTIMATE_RESULT – the results of calling "estimate" changed

If *should_plot* is called from a notification handler, the payload is the handler `event` parameter.

get_notebook_code(*idx*)

cytoflowgui.workflow.operations.channel_stat

cytoflowgui.workflow.operations.channel_stat.**mean_95ci**(*x*)

cytoflowgui.workflow.operations.channel_stat.**geomean_95ci**(*x*)

class cytoflowgui.workflow.operations.channel_stat.**ChannelStatisticWorkflowOp**

Bases: *cytoflowgui.workflow.operations.operation_base.WorkflowOperation*, *cytoflow.operations.channel_stat.ChannelStatisticOp*

apply(*experiment*)

Apply the operation to an *Experiment*.

Parameters **experiment** – The *Experiment* to apply this operation to.

Returns A new *Experiment*, containing a new entry in *Experiment.statistics*. The key of the new entry is a tuple `(name, function)` (or `(name, statistic_name)` if *statistic_name* is set.

Return type *Experiment*

clear_estimate()

get_notebook_code(*idx*)

cytoflowgui.workflow.operations.color_translation

class `cytoflowgui.workflow.operations.color_translation.`**`Control`**

Bases: `traits.has_traits.HasTraits`

class `cytoflowgui.workflow.operations.color_translation.`**`ColorTranslationWorkflowOp`**

Bases: `cytoflowgui.workflow.operations.operation_base.WorkflowOperation`, `cytoflow.operations.color_translation.ColorTranslationOp`

`default_view`(***kwargs*)

Returns a diagnostic plot to see if the bleedthrough spline estimation is working.

Returns A diagnostic view, call `ColorTranslationDiagnostic.plot` to see the diagnostic plots

Return type *IView*

`estimate`(*experiment*)

Estimate the mapping from the two-channel controls

Parameters

- **experiment** (`Experiment`) – The `Experiment` used to check the voltages, etc. of the control tubes. Also the source of the operation history that is replayed on the control tubes.
- **subset** (*Str*) – A Python expression used to subset the controls before estimating the color translation parameters.

`apply`(*experiment*)

Applies the color translation to an experiment

Parameters **experiment** (`Experiment`) – the old_experiment to which this op is applied

Returns

a new experiment with the color translation applied. The corrected channels also have the following new metadata:

channel_translation : Str Which channel was this one translated to?

channel_translation_fn : Callable (`pandas.Series` –> `pandas.Series`) The function that translated this channel

Return type *Experiment*

`should_clear_estimate`(*changed*, *payload*)

`clear_estimate`()

`get_notebook_code`(*idx*)

class `cytoflowgui.workflow.operations.color_translation.`**`ColorTranslationWorkflowView`**

Bases: `cytoflowgui.workflow.views.view_base.WorkflowView`, `cytoflow.operations.color_translation.ColorTranslationDiagnostic`

`should_plot`(*changed*, *payload*)

Should the owning WorkflowItem refresh the plot when certain things change? `changed` can be: - Changed.VIEW – the view's parameters changed - Changed.RESULT – this WorkflowItem's result changed - Changed.PREV_RESULT – the previous WorkflowItem's result changed - Changed.ESTIMATE_RESULT – the results of calling "estimate" changed

If `should_plot` is called from a notification handler, the payload is the handler `event` parameter.

`get_notebook_code`(*idx*)

cytoflowgui.workflow.operations.density

class `cytoflowgui.workflow.operations.density.`**`DensityGateWorkflowOp`**

Bases: *`cytoflowgui.workflow.operations.operation_base.WorkflowOperation`*, *`cytoflow.operations.density.DensityGateOp`*

`default_view`(***kwargs*)

Returns a diagnostic plot of the Gaussian mixture model.

Returns a diagnostic view, call *`DensityGateView.plot`* to see the diagnostic plot.

Return type *`IView`*

`clear_estimate`()

`apply`(*experiment*)

Creates a new condition based on membership in the gate that was parameterized with *`estimate`*.

Parameters **experiment** (*`Experiment`*) – the *`Experiment`* to apply the gate to.

Returns a new *`Experiment`* with the new gate applied.

Return type *`Experiment`*

`get_notebook_code`(*idx*)

class `cytoflowgui.workflow.operations.density.`**`DensityGateWorkflowView`**

Bases: *`cytoflowgui.workflow.views.view_base.WorkflowByView`*, *`cytoflow.operations.density.DensityGateView`*

`should_plot`(*changed*, *payload*)

Should the owning WorkflowItem refresh the plot when certain things change? `changed` can be: - Changed.VIEW – the view's parameters changed - Changed.RESULT – this WorkflowItem's result changed - Changed.PREV_RESULT – the previous WorkflowItem's result changed - Changed.ESTIMATE_RESULT – the results of calling "estimate" changed

If *`should_plot`* is called from a notification handler, the payload is the handler `event` parameter.

`get_notebook_code`(*idx*)

cytoflowgui.workflow.operations.flowpeaks

class `cytoflowgui.workflow.operations.flowpeaks.`**`FlowPeaksWorkflowOp`**

Bases: *`cytoflowgui.workflow.operations.operation_base.WorkflowOperation`*, *`cytoflow.operations.flowpeaks.FlowPeaksOp`*

`default_view`(***kwargs*)

Returns a diagnostic plot of the Gaussian mixture model.

Parameters

- **channels** (*List(Str)*) – Which channels to plot? Must be contain either one or two channels.
- **scale** (*List({'linear', 'log', 'logicle'})*) – How to scale the channels before plotting them
- **density** (*bool*) – Should we plot a scatterplot or the estimated density function?

Returns an *`IView`*, call *`plot`* to see the diagnostic plot.

Return type *`IView`*

`estimate`(*experiment*)

Estimate the k-means clusters, then hierarchically merge them.

Parameters

- **experiment** (*Experiment*) – The *Experiment* to use to estimate the k-means clusters
- **subset** (*str (default = None)*) – A Python expression that specifies a subset of the data in `experiment` to use to parameterize the operation.

`apply`(*experiment*)

Assign events to a cluster.

Assigns each event to one of the k-means centroids from *estimate*, then groups together events in the same cluster hierarchy.

Parameters **experiment** (*Experiment*) – the *Experiment* to apply the gate to.

Returns A new *Experiment* with the gate applied to it. TODO - document the extra statistics

Return type *Experiment*

`clear_estimate`()

`get_notebook_code`(*idx*)

class cytoflowgui.workflow.operations.flowpeaks.**FlowPeaksWorkflowView**

Bases: *cytoflowgui.workflow.views.view_base.WorkflowByView*, *cytoflow.operations.base_op_views.By2DView*

`id = 'edu.mit.synbio.cytoflowgui.op_plugins.flowpeaks'`

`friendly_id = 'FlowPeaks'`

`plot`(*experiment*, ***kwargs*)

A default *plot* that passes *current_plot* as the plot name.

`get_notebook_code`(*idx*)

cytoflowgui.workflow.operations.gaussian_1d

class cytoflowgui.workflow.operations.gaussian_1d.**GaussianMixture1DWorkflowOp**

Bases: *cytoflowgui.workflow.operations.operation_base.WorkflowOperation*, *cytoflow.operations.gaussian.GaussianMixtureOp*

`estimate`(*experiment*)

Estimate the Gaussian mixture model parameters

Parameters

- **experiment** (*Experiment*) – The data to use to estimate the mixture parameters
- **subset** (*str (default = None)*) – If set, a Python expression to determine the subset of the data to use to in the estimation.

`apply`(*experiment*)

Assigns new metadata to events using the mixture model estimated in *estimate*.

Returns

A new *Experiment* with the new condition variables as described in the class documentation. Also adds the following new statistics:

- **mean** [Float] the mean of the fitted gaussian in each channel for each component.
- **sigma** [(Float, Float)] the locations the mean +/- one standard deviation in each channel for each component.

- **correlation** [Float] the correlation coefficient between each pair of channels for each component.
- **proportion** [Float] the proportion of events in each component of the mixture model. only added if `num_components` > 1.

Return type *Experiment*

`default_view`(***kwargs*)
Returns a diagnostic plot of the Gaussian mixture model.

Returns An `IView`, call `plot` to see the diagnostic plot.

Return type `IView`

`clear_estimate`()

`get_notebook_code`(*idx*)

class cytoflowgui.workflow.operations.gaussian_1d.**GaussianMixture1DWorkflowView**
Bases: `cytoflowgui.workflow.views.view_base.WorkflowByView`, `cytoflow.operations.gaussian.GaussianMixture1DView`

`plot`(*experiment*, ***kwargs*)
A default `plot` that passes `current_plot` as the plot name.

`get_notebook_code`(*idx*)

cytoflowgui.workflow.operations.gaussian_2d

class cytoflowgui.workflow.operations.gaussian_2d.**GaussianMixture2DWorkflowOp**
Bases: `cytoflowgui.workflow.operations.operation_base.WorkflowOperation`, `cytoflow.operations.gaussian.GaussianMixtureOp`

`default_view`(***kwargs*)
Returns a diagnostic plot of the Gaussian mixture model.

Returns An `IView`, call `plot` to see the diagnostic plot.

Return type `IView`

`estimate`(*experiment*)
Estimate the Gaussian mixture model parameters

Parameters

- **experiment** (*Experiment*) – The data to use to estimate the mixture parameters
- **subset** (*str (default = None)*) – If set, a Python expression to determine the subset of the data to use to in the estimation.

`apply`(*experiment*)
Assigns new metadata to events using the mixture model estimated in `estimate`.

Returns

A new `Experiment` with the new condition variables as described in the class documentation. Also adds the following new statistics:

- **mean** [Float] the mean of the fitted gaussian in each channel for each component.
- **sigma** [(Float, Float)] the locations the mean +/- one standard deviation in each channel for each component.

- **correlation** [Float] the correlation coefficient between each pair of channels for each component.
- **proportion** [Float] the proportion of events in each component of the mixture model. only added if `num_components` > 1.

Return type *Experiment*

`clear_estimate()`

`get_notebook_code(idx)`

`class cytoflowgui.workflow.operations.gaussian_2d.GaussianMixture2DWorkflowView`

Bases: `cytoflowgui.workflow.views.view_base.WorkflowByView`, `cytoflow.operations.gaussian.GaussianMixture2DView`

`plot(experiment, **kwargs)`

A default `plot` that passes `current_plot` as the plot name.

`get_notebook_code(idx)`

cytoflowgui.workflow.operations.import_op

`class cytoflowgui.workflow.operations.import_op.ValidPythonIdentifier(default_value=<traits.trait_type._NoDefaul`
`object>, **metadata)`

Bases: `traits.trait_types.BaseCStr`

`info_text = 'a valid python identifier'`

A description of the type of value this trait accepts:

`validate(obj, name, value)`

Validates that a specified value is valid for this trait.

Note: The 'fast validator' version performs this check in C.

`class cytoflowgui.workflow.operations.import_op.Channel`

Bases: `traits.has_traits.HasTraits`

`class cytoflowgui.workflow.operations.import_op.ImportWorkflowOp`

Bases: `cytoflowgui.workflow.operations.operation_base.WorkflowOperation`, `cytoflow.operations.import_op.ImportOp`

`reset_channels()`

`estimate(_)`

`apply(*args, **kwargs)`

Load a new `Experiment`.

Parameters

- **experiment** (*Experiment*) – Ignored
- **metadata_only** (*bool (default = False)*) – Only "import" the metadata, creating an Experiment with all the expected metadata and structure but 0 events.

Returns

The new `Experiment`. New channels have the following metadata:

- **voltage - int** The voltage that this channel was collected at. Determined by the `$PnV` field from the first FCS file.

- **range - int** The maximum range of this channel. Determined by the `$PnR` field from the first FCS file.

New experimental conditions do not have **voltage** or **range** metadata, obviously. Instead, they have **experiment** set to `True`, to distinguish the experimental variables from the conditions that were added by gates, etc.

If `ignore_v` is set, it is added as a key to the `Experiment`-wide metadata.

Return type *Experiment*

`clear_estimate()`

`get_notebook_code(idx)`

cytoflowgui.workflow.operations.kmeans

`class cytoflowgui.workflow.operations.kmeans.KMeansWorkflowOp`

Bases: `cytoflowgui.workflow.operations.operation_base.WorkflowOperation`, `cytoflow.operations.kmeans.KMeansOp`

`default_view(**kwargs)`

Returns a diagnostic plot of the k-means clustering.

Returns **IView**

Return type an IView, call `KMeans1DView.plot` to see the diagnostic plot.

`estimate(experiment)`

Estimate the k-means clusters

Parameters

- **experiment** (*Experiment*) – The `Experiment` to use to estimate the k-means clusters
- **subset** (*str (default = None)*) – A Python expression that specifies a subset of the data in `experiment` to use to parameterize the operation.

`apply(experiment)`

Apply the KMeans clustering to the data.

Returns

a new Experiment with one additional entry in `Experiment.conditions` named `name`, of type `category`. The new category has values `name_1`, `name_2`, etc to indicate which k-means cluster an event is a member of.

The new `Experiment` also has one new statistic called `centers`, which is a list of tuples encoding the centroids of each k-means cluster.

Return type *Experiment*

`clear_estimate()`

`get_notebook_code(idx)`

`class cytoflowgui.workflow.operations.kmeans.KMeansWorkflowView`

Bases: `cytoflowgui.workflow.views.view_base.WorkflowByView`

`id = 'edu.mit.synbio.cytoflowgui.op_plugins.kmeans'`

`friendly_id = 'KMeans'`

plot(*experiment*, ***kwargs*)
A default `plot` that passes `current_plot` as the plot name.

enum_plots(*experiment*)

get_notebook_code(*idx*)

cytoflowgui.workflow.operations.operation_base

class cytoflowgui.workflow.operations.operation_base.**IWorkflowOperation**(*adaptee*, *default=<class 'traits.adaptation.adaptation_error.Adapt*

Bases: `cytoflow.operations.i_operation.IOperation`

An interface that extends a `cytoflow` operation with functions required for GUI support.

In addition to implementing the interface below, another common thing to do in the derived class is to override traits of the underlying class in order to add metadata that controls their handling by the workflow. Currently, relevant metadata include:

- **apply** - This trait is used by the operations `apply` method.
- **estimate** - This trait is used by the operation's `estimate` method.
- **estimate_result** - This trait is set as a result of calling `estimate`.
- **status** - Holds status variables like the number of events from the `ImportOp`.
- **transient** - A temporary variable (not copied between processes or serialized).

do_estimate
Firing this event causes the operation's `estimate` method to be called.

Type Event

changed
Used to transmit status information back from the operation to the workflow. Set its value to the name of the trait that was changed

Type Event

should_apply(*changed*, *payload*)
Should the owning WorkflowItem apply this operation when certain things change? `changed` can be:

- Changed.APPLY – the parameters required to run apply() changed
- Changed.PREV_RESULT – the previous WorkflowItem's result changed
- Changed.ESTIMATE_RESULT – the results of calling "estimate" changed

If `should_apply` is called from a notification handler, then `payload` is the `event` object from the notification handler.

should_clear_estimate(*changed*, *payload*)
Should the owning WorkflowItem clear the estimated model by calling op.clear_estimate()? `changed` can be:

- Changed.ESTIMATE – the parameters required to call `estimate` (ie traits with `estimate = True` metadata) have changed
- Changed.PREV_RESULT – the previous `WorkflowItem`'s result changed

If `should_clear_estimate` is called from a notificaion handler, then `payload` is the `event` object.

clear_estimate()

Clear whatever variables hold the results of calling estimate()

get_notebook_code(*idx*)

Return Python code suitable for a Jupyter notebook cell that runs this operation.

Parameters **idx** (*integer*) – The index of the `WorkflowItem` that holds this operation.

Returns The Python code that calls this module.

Return type string

class cytoflowgui.workflow.operations.operation_base.**WorkflowOperation**

Bases: `traits.has_traits.HasStrictTraits`

A default implementation of `IWorkflowOperation`

should_apply(*changed*, *payload*)

should_clear_estimate(*changed*, *payload*)

clear_estimate()

get_notebook_code(*idx*)

cytoflowgui.workflow.operations.pca

class cytoflowgui.workflow.operations.pca.**Channel**

Bases: `traits.has_traits.HasTraits`

class cytoflowgui.workflow.operations.pca.**PCAWorkflowOp**

Bases: `cytoflowgui.workflow.operations.operation_base.WorkflowOperation`, `cytoflow.operations.pca.PCAOp`

estimate(*experiment*)

Estimate the decomposition

Parameters

- **experiment** (*Experiment*) – The `Experiment` to use to estimate the k-means clusters
- **subset** (*str (default = None)*) – A Python expression that specifies a subset of the data in `experiment` to use to parameterize the operation.

apply(*experiment*)

Apply the PCA decomposition to the data.

Returns a new Experiment with additional `Experiment.channels` named `name_1 ... name_n`

Return type *Experiment*

clear_estimate()

get_notebook_code(*idx*)

cytoflowgui.workflow.operations.polygon

class cytoflowgui.workflow.operations.polygon.**PolygonSelectionView**

Bases: *cytoflowgui.workflow.views.view_base.WorkflowView*, *cytoflow.operations.polygon.ScatterplotPolygonSelectionView*

clear_estimate()

get_notebook_code(*idx*)

class cytoflowgui.workflow.operations.polygon.**PolygonWorkflowOp**

Bases: *cytoflowgui.workflow.operations.operation_base.WorkflowOperation*, *cytoflow.operations.polygon.PolygonOp*

default_view(***kwargs*)

Returns an *IView* that allows a user to view the polygon or interactively draw it.

Parameters density (*bool, default = False*) – If True, return a density plot instead of a scatterplot.

get_notebook_code(*idx*)

cytoflowgui.workflow.operations.quad

class cytoflowgui.workflow.operations.quad.**QuadSelectionView**

Bases: *cytoflowgui.workflow.views.view_base.WorkflowView*, *cytoflow.operations.quad.ScatterplotQuadSelectionView*

clear_estimate()

get_notebook_code(*idx*)

class cytoflowgui.workflow.operations.quad.**QuadWorkflowOp**

Bases: *cytoflowgui.workflow.operations.operation_base.WorkflowOperation*, *cytoflow.operations.quad.QuadOp*

default_view(***kwargs*)

Returns an *IView* that allows a user to view the quad selector or interactively draw it.

Parameters density (*bool, default = False*) – If True, return a density plot instead of a scatterplot.

clear_estimate()

get_notebook_code(*idx*)

cytoflowgui.workflow.operations.range

class cytoflowgui.workflow.operations.range.**RangeSelectionView**

Bases: *cytoflowgui.workflow.views.view_base.WorkflowView*, *cytoflow.operations.range.RangeSelection*

clear_estimate()

get_notebook_code(*idx*)

class cytoflowgui.workflow.operations.range.**RangeWorkflowOp**

Bases: *cytoflowgui.workflow.operations.operation_base.WorkflowOperation*, *cytoflow.operations.range.RangeOp*

default_view(***kwargs*)

get_notebook_code(*idx*)

cytoflowgui.workflow.operations.range2d

class cytoflowgui.workflow.operations.range2d.**Range2DSelectionView**

Bases: *cytoflowgui.workflow.views.view_base.WorkflowView*, *cytoflow.operations.range2d.ScatterplotRangeSelection2DView*

clear_estimate()

get_notebook_code(*idx*)

class cytoflowgui.workflow.operations.range2d.**Range2DWorkflowOp**

Bases: *cytoflowgui.workflow.operations.operation_base.WorkflowOperation*, *cytoflow.operations.range2d.Range2DOp*

default_view(***kwargs*)

Returns an *IView* that allows a user to view the selection or interactively draw it.

Parameters **density** (*bool, default = False*) – If True, return a density plot instead of a scatter-plot.

get_notebook_code(*idx*)

cytoflowgui.workflow.operations.ratio

class cytoflowgui.workflow.operations.ratio.**RatioWorkflowOp**

Bases: *cytoflowgui.workflow.operations.operation_base.WorkflowOperation*, *cytoflow.operations.ratio.RatioOp*

clear_estimate()

get_notebook_code(*idx*)

cytoflowgui.workflow.operations.tasbe

class cytoflowgui.workflow.operations.tasbe.**BleedthroughControl**

Bases: traits.has_traits.HasTraits

class cytoflowgui.workflow.operations.tasbe.**TranslationControl**

Bases: traits.has_traits.HasTraits

class cytoflowgui.workflow.operations.tasbe.**BeadUnit**

Bases: traits.has_traits.HasTraits

class cytoflowgui.workflow.operations.tasbe.**Progress**

Bases: object

NO_MODEL = 'No model estimated'

AUTOFLUORESCENCE = 'Estimating autofluorescence'

BLEEDTHROUGH = 'Estimating bleedthrough'

BEAD_CALIBRATION = 'Estimating bead calibration'

COLOR_TRANSLATION = 'Estimating color translation'

VALID = 'Valid model estimated!'

class cytoflowgui.workflow.operations.tasbe.**TasbeWorkflowOp**

Bases: *cytoflowgui.workflow.operations.operation_base.WorkflowOperation*

estimate(*experiment*, *subset=None*)

should_clear_estimate(*changed*, *payload*)

clear_estimate()

apply(*experiment*)

default_view(***kwargs*)

get_notebook_code(*idx*)

class cytoflowgui.workflow.operations.tasbe.**TasbeWorkflowView**

Bases: *cytoflowgui.workflow.views.view_base.WorkflowView*

id = 'edu.mit.synbio.cytoflowgui.workflow.operations.tasbeview'

friendly_id = 'TASBE Calibration'

enum_plots(*experiment*)

should_plot(*changed*, *payload*)

Should the owning WorkflowItem refresh the plot when certain things change? `changed` can be: - Changed.VIEW – the view's parameters changed - Changed.RESULT – this WorkflowItem's result changed - Changed.PREV_RESULT – the previous WorkflowItem's result changed - Changed.ESTIMATE_RESULT – the results of calling "estimate" changed

If *should_plot* is called from a notification handler, the payload is the handler `event` parameter.

plot(*experiment*, ***kwargs*)

get_notebook_code(*idx*)

cytoflowgui.workflow.operations.threshold

class cytoflowgui.workflow.operations.threshold.**ThresholdSelectionView**

Bases: *cytoflowgui.workflow.views.view_base.WorkflowView*, *cytoflow.operations.threshold.ThresholdSelection*

get_notebook_code(*idx*)

class cytoflowgui.workflow.operations.threshold.**ThresholdWorkflowOp**

Bases: *cytoflowgui.workflow.operations.operation_base.WorkflowOperation*, *cytoflow.operations.threshold.ThresholdOp*

default_view(***kwargs*)

clear_estimate()

get_notebook_code(*idx*)

cytoflowgui.workflow.operations.xform_stat

cytoflowgui.workflow.operations.xform_stat.**mean_95ci**(*x*)

cytoflowgui.workflow.operations.xform_stat.**geomean_95ci**(*x*)

class cytoflowgui.workflow.operations.xform_stat.**TransformStatisticWorkflowOp**

Bases: *cytoflowgui.workflow.operations.operation_base.WorkflowOperation*, *cytoflow.operations.xform_stat.TransformStatisticOp*

apply(*experiment*)

Applies *function* to a statistic.

Parameters **experiment** (*Experiment*) – The *Experiment* to apply the operation to

Returns The same as the old experiment, but with a new statistic that results from applying *function* to the statistic specified in *statistic*.

Return type *Experiment*

clear_estimate()

get_notebook_code(*idx*)

cytoflowgui.workflow.views package

cytoflowgui.workflow.views

Submodules

cytoflowgui.workflow.views.bar_chart

class cytoflowgui.workflow.views.bar_chart.**BarChartPlotParams**

Bases: *cytoflowgui.workflow.views.view_base.Stats1DPlotParams*

class cytoflowgui.workflow.views.bar_chart.**BarChartWorkflowView**

Bases: *cytoflowgui.workflow.views.view_base.WorkflowByView*, *cytoflow.views.bar_chart.BarChartView*

get_notebook_code(*idx*)

cytoflowgui.workflow.views.density

class cytoflowgui.workflow.views.density.**DensityPlotParams**

Bases: *cytoflowgui.workflow.views.view_base.Data2DPlotParams*

class cytoflowgui.workflow.views.density.**DensityWorkflowView**

Bases: *cytoflowgui.workflow.views.view_base.WorkflowFacetView*, *cytoflow.views.densityplot.DensityView*

get_notebook_code(*idx*)

cytoflowgui.workflow.views.export_fcs

class `cytoflowgui.workflow.views.export_fcs.`**`ExportFCSWorkflowView`**

Bases: *cytoflowgui.workflow.views.view_base.WorkflowView*, *cytoflow.views.export_fcs.ExportFCS*

`enum_conditions_and_files`(*experiment*)

Return an iterator over the conditions and file names that this export will produce from a given experiment.

Parameters **experiment** (*Experiment*) – The `Experiment` to export

`plot`(*experiment*, *plot_name=None*, ***kwargs*)

Plot a table of the conditions, filenames, and number of events

`get_notebook_code`(*idx*)

cytoflowgui.workflow.views.histogram

class `cytoflowgui.workflow.views.histogram.`**`HistogramPlotParams`**

Bases: *cytoflowgui.workflow.views.view_base.Data1DPlotParams*

class `cytoflowgui.workflow.views.histogram.`**`HistogramWorkflowView`**

Bases: *cytoflowgui.workflow.views.view_base.WorkflowFacetView*, *cytoflow.views.histogram.HistogramView*

`get_notebook_code`(*idx*)

cytoflowgui.workflow.views.histogram_2d

class `cytoflowgui.workflow.views.histogram_2d.`**`Histogram2DPlotParams`**

Bases: *cytoflowgui.workflow.views.view_base.Data2DPlotParams*

class `cytoflowgui.workflow.views.histogram_2d.`**`Histogram2DWorkflowView`**

Bases: *cytoflowgui.workflow.views.view_base.WorkflowFacetView*, *cytoflow.views.histogram_2d.Histogram2DView*

`get_notebook_code`(*idx*)

cytoflowgui.workflow.views.kde_1d

class `cytoflowgui.workflow.views.kde_1d.`**`Kde1DPlotParams`**

Bases: *cytoflowgui.workflow.views.view_base.Data1DPlotParams*

class `cytoflowgui.workflow.views.kde_1d.`**`Kde1DWorkflowView`**

Bases: *cytoflowgui.workflow.views.view_base.WorkflowFacetView*, *cytoflow.views.kde_1d.Kde1DView*

`get_notebook_code`(*idx*)

cytoflowgui.workflow.views.kde_2d

class cytoflowgui.workflow.views.kde_2d.**Kde2DPlotParams**

Bases: *cytoflowgui.workflow.views.view_base.Data2DPlotParams*

class cytoflowgui.workflow.views.kde_2d.**Kde2DWorkflowView**

Bases: *cytoflowgui.workflow.views.view_base.WorkflowFacetView*, *cytoflow.views.kde_2d.Kde2DView*

get_notebook_code(*idx*)

cytoflowgui.workflow.views.parallel_coords

class cytoflowgui.workflow.views.parallel_coords.**ParallelCoordinatesPlotParams**

Bases: *cytoflowgui.workflow.views.view_base.DataPlotParams*

class cytoflowgui.workflow.views.parallel_coords.**ParallelCoordinatesWorkflowView**

Bases: *cytoflowgui.workflow.views.view_base.WorkflowFacetView*, *cytoflow.views.parallel_coords.ParallelCoordinatesView*

get_notebook_code(*idx*)

cytoflowgui.workflow.views.radviz

class cytoflowgui.workflow.views.radviz.**RadvizPlotParams**

Bases: *cytoflowgui.workflow.views.view_base.DataPlotParams*

class cytoflowgui.workflow.views.radviz.**Channel**

Bases: traits.has_traits.HasTraits

class cytoflowgui.workflow.views.radviz.**RadvizWorkflowView**

Bases: *cytoflowgui.workflow.views.view_base.WorkflowFacetView*, *cytoflow.views.radviz.RadvizView*

get_notebook_code(*idx*)

cytoflowgui.workflow.views.scatterplot

class cytoflowgui.workflow.views.scatterplot.**ScatterplotPlotParams**

Bases: *cytoflowgui.workflow.views.view_base.Data2DPlotParams*

class cytoflowgui.workflow.views.scatterplot.**ScatterplotWorkflowView**

Bases: *cytoflowgui.workflow.views.view_base.WorkflowFacetView*, *cytoflow.views.scatterplot.ScatterplotView*

get_notebook_code(*idx*)

cytoflowgui.workflow.views.stats_1d

class cytoflowgui.workflow.views.stats_1d.**Stats1DPlotParams**

Bases: *cytoflowgui.workflow.views.view_base.Stats1DPlotParams*

class cytoflowgui.workflow.views.stats_1d.**Stats1DWorkflowView**

Bases: *cytoflowgui.workflow.views.view_base.WorkflowByView*, *cytoflow.views.stats_1d.Stats1DView*

get_notebook_code(*idx*)

cytoflowgui.workflow.views.stats_2d

class cytoflowgui.workflow.views.stats_2d.**Stats2DPlotParams**

Bases: *cytoflowgui.workflow.views.view_base.Stats2DPlotParams*

class cytoflowgui.workflow.views.stats_2d.**Stats2DWorkflowView**

Bases: *cytoflowgui.workflow.views.view_base.WorkflowByView*, *cytoflow.views.stats_2d.Stats2DView*

get_notebook_code(*idx*)

cytoflowgui.workflow.views.table

class cytoflowgui.workflow.views.table.**TableWorkflowView**

Bases: *cytoflowgui.workflow.views.view_base.WorkflowByView*, *cytoflow.views.table.TableView*

plot(*experiment*, ***kwargs*)

A default *plot* that passes *current_plot* as the plot name.

get_notebook_code(*idx*)

cytoflowgui.workflow.views.view_base

class cytoflowgui.workflow.views.view_base.**IterWrapper**(*iterator*, *by*)

Bases: object

class cytoflowgui.workflow.views.view_base.**IWorkflowView**(*adaptee*, *default=<class 'traits.adaptation.adaptation_error.AdaptationError'>*)

Bases: *cytoflow.views.i_view.IView*

An interface that extends a *cytoflow* view with functions required for GUI support.

In addition to implementing the interface below, another common thing to do in the derived class is to override traits of the underlying class in order to add metadata that controls their handling by the workflow. Currently, relevant metadata include:

- **status** - Holds status variables – only sent from the remote process to the local one, and doesn't re-plot the view. Example: the possible plot names.
- **transient** - A temporary variable (not copied between processes or serialized).

TODO - finish this docstring (plotfacet, plot_params, etc)

changed
Used to transmit status information back from the operation to the workflow. Set its value to the name of the trait that was changed

Type Event

should_plot(*changed*, *payload*)
Should the owning WorkflowItem refresh the plot when certain things change? *changed* can be: - Changed.VIEW – the view's parameters changed - Changed.RESULT – this WorkflowItem's result changed - Changed.PREV_RESULT – the previous WorkflowItem's result changed - Changed.ESTIMATE_RESULT – the results of calling "estimate" changed

If *should_plot* was called from an event handler, the event is passed in as `payload`

get_notebook_code(*idx*)
Return Python code suitable for a Jupyter notebook cell that plots this view.

Parameters **idx** (*integer*) – The index of the `WorkflowItem` that holds this view.

Returns The Python code that calls this module.

Return type string

class cytoflowgui.workflow.views.view_base.**WorkflowView**
Bases: `traits.has_traits.HasStrictTraits`

Default implementation of IWorkflowView.

Make sure this class is FIRST in the derived class's declaration so it shows up earlier in the MRO than the base class from the *cytoflow* module.

current_plot
Passed as the `current_plot` keyword to the underlying *IView.plot*

Type Any

enum_plots(*experiment*)

should_plot(*changed*, *payload*)
Should the owning WorkflowItem refresh the plot when certain things change? `changed` can be: - Changed.VIEW – the view's parameters changed - Changed.RESULT – this WorkflowItem's result changed - Changed.PREV_RESULT – the previous WorkflowItem's result changed - Changed.ESTIMATE_RESULT – the results of calling "estimate" changed

If *should_plot* is called from a notification handler, the payload is the handler `event` parameter.

get_notebook_code(*idx*)

class cytoflowgui.workflow.views.view_base.**WorkflowFacetView**
Bases: *cytoflowgui.workflow.views.view_base.WorkflowView*

A *WorkflowView* that subsets the data by a facet before plotting.

plotfacet
What facet should *current_plot* refer to?

Type Str

enum_plots(*experiment*)

plot(*experiment*, ***kwargs*)
A default *plot* that subsets by the *plotfacet* and *current_plot*. If you need it to do something else, you must override this method!

class cytoflowgui.workflow.views.view_base.**WorkflowByView**
Bases: *cytoflowgui.workflow.views.view_base.WorkflowView*

plot(*experiment*, ***kwargs*)
A default *plot* that passes *current_plot* as the plot name.

enum_plots(*experiment*)

class cytoflowgui.workflow.views.view_base.**BasePlotParams**
Bases: traits.has_traits.HasStrictTraits

class cytoflowgui.workflow.views.view_base.**DataPlotParams**
Bases: *cytoflowgui.workflow.views.view_base.BasePlotParams*

class cytoflowgui.workflow.views.view_base.**Data1DPlotParams**
Bases: *cytoflowgui.workflow.views.view_base.DataPlotParams*

class cytoflowgui.workflow.views.view_base.**Data2DPlotParams**
Bases: *cytoflowgui.workflow.views.view_base.DataPlotParams*

class cytoflowgui.workflow.views.view_base.**Stats1DPlotParams**
Bases: *cytoflowgui.workflow.views.view_base.BasePlotParams*

class cytoflowgui.workflow.views.view_base.**Stats2DPlotParams**
Bases: *cytoflowgui.workflow.views.view_base.BasePlotParams*

class cytoflowgui.workflow.views.view_base.**Channel**
Bases: traits.has_traits.HasStrictTraits

cytoflowgui.workflow.views.violin

class cytoflowgui.workflow.views.violin.**ViolinPlotParams**
Bases: *cytoflowgui.workflow.views.view_base.Data1DPlotParams*

class cytoflowgui.workflow.views.violin.**ViolinPlotWorkflowView**
Bases: *cytoflowgui.workflow.views.view_base.WorkflowFacetView*, *cytoflow.views.violin.ViolinPlotView*

get_notebook_code(*idx*)

Submodules

cytoflowgui.workflow.serialization

Utility bits that let us use camel to serialize a *RemoteWorkflow*.

Many of the dumpers and loaders support serializing pandas types, such as pandas.Series and pandas.DataFrame, or for testing serialization with unit tests.

cytoflowgui.workflow.serialization.**load_yaml**(*path*)
Load a Python object from a YAML file.

Parameters **path** (*string*) – The path to the YAML file to load

Returns The Python object loaded from the YAML file

Return type object

cytoflowgui.workflow.serialization.**save_yaml**(*data*, *path*, *lock_versions={}*)
Save a Python object to a YAML file

Parameters

- **data** (*object*) – The Python object to serialize
- **path** (*string*) – The path to save to
- **lock_versions** (*dict*) – A dictionary of types and versions of dumpers to use when serializing.

`cytoflowgui.workflow.serialization.`**`traits_eq`**`(`*self*, *other*`)`

Are the copyable traits of two `traits.has_traits.HasTraits` equal?

`cytoflowgui.workflow.serialization.`**`traits_hash`**`(`*self*`)`

Get a unique hash of a `traits.has_traits.HasTraits`

`cytoflowgui.workflow.serialization.`**`traits_repr`**`(`*obj*`)`

A uniform implementation of **__repr__** for `traits.has_traits.HasTraits`

`cytoflowgui.workflow.serialization.`**`traits_str`**`(`*obj*`)`

A uniform implementation of **__str__** for `traits.has_traits.HasTraits`

cytoflowgui.workflow.subset

Structured representation of the possible clauses for a **subset** argument to *`IOperation.estimate`* or *`IView.plot`*.

`ISubset` - The top-level `traits.has_traits.Interface`

`BoolSubset` **- represents a subset of True or False values from a boolean** condition

`CategorySubset` **- represents a subset of one or more values from a categorical** condition.

`RangeSubset` **- represents a subset that is a range of values from a** numerical condition.

`class cytoflowgui.workflow.subset.`**`ISubset`**`(`*adaptee*, *default=<class 'traits.adaptation.adaptation_error.AdaptationError'>*`)`

Bases: `traits.has_traits.Interface`

The interface that the rest of the subset classes must implement.

`class cytoflowgui.workflow.subset.`**`BoolSubset`**

Bases: `traits.has_traits.HasStrictTraits`

A subset that selects either True or False values from a boolean condition

`class cytoflowgui.workflow.subset.`**`CategorySubset`**

Bases: `traits.has_traits.HasStrictTraits`

A subset that selects one or more values from a categorical condition

`class cytoflowgui.workflow.subset.`**`RangeSubset`**

Bases: `traits.has_traits.HasStrictTraits`

A subset that selects a range from a numerical condition

cytoflowgui.workflow.workflow

The main model for the GUI.

At its core, the model is a list of `WorkflowItem` instances. A `WorkflowItem` wraps an operation, its completion status, the result of applying it to the previous `WorkflowItem`'s result, and `IView`'s on that result. The Workflow also maintains a "current" or selected WorkflowItem.

The left panel of the GUI is a View on this object (viewing the list of WorkflowItem instances), and the right panel of the GUI is a View of the selected WorkflowItem's current view.

So far, so simple. However, in a single-threaded GUI application, the UI freezes when something processor-intensive is happening. Adding another thread doesn't help matters because of the CPython global interpreter lock; while Python is otherwise computing, the GUI doesn't update. To solve this, the Workflow maintains a copy of itself in a separate process. The `LocalWorkflow` is the one that is viewed by the GUI; the `RemoteWorkflow` is the one that actually loads the data and does the processing. Thus the GUI remains responsive. Changed attributes in either Workflow are noticed by a set of Traits handlers, which send those changes to the other process.

This process is also where the plotting happens. For an explanation of how the plots are ferried back to the GUI, see the module docstring for `cytoflowgui.matplotlib_backend_local` and `cytoflowgui.matplotlib_backend_remote`

class cytoflowgui.workflow.workflow.**Msg**

Bases: `object`

Messages passed between the local and remote workflows

NEW_WORKFLOW = 'NEW_WORKFLOW'

ADD_ITEMS = 'ADD_ITEMS'

REMOVE_ITEMS = 'REMOVE_ITEMS'

SELECT = 'SELECT'

UPDATE_OP = 'UPDATE_OP'

UPDATE_VIEW = 'UPDATE_VIEW'

CHANGE_CURRENT_VIEW = 'CHANGE_CURRENT_VIEW'

CHANGE_CURRENT_PLOT = 'CHANGE_CURRENT_PLOT'

UPDATE_WI = 'UPDATE_WI'

ESTIMATE = 'ESTIMATE'

APPLY_CALLED = 'APPLY_CALLED'

PLOT_CALLED = 'PLOT_CALLED'

EVAL = 'EVAL'

EXEC = 'EXEC'

RUN_ALL = 'RUN_ALL'

SHUTDOWN = 'SHUTDOWN'

class cytoflowgui.workflow.workflow.**UniquePriorityQueue**(*maxsize=0*)

Bases: `queue.PriorityQueue`

A PriorityQueue that only allows one copy of each item. http://stackoverflow.com/questions/5997189/how-can-i-make-a-unique-value-priority-queue-in-python

`cytoflowgui.workflow.workflow.`**`filter_unpicklable`**`(`*obj*`)`
Recursively filter unpicklable items from lists and dictionaries

`class` `cytoflowgui.workflow.workflow.`**`LocalWorkflow`**`(`*remote_workflow_connection*, ***kwargs*`)`
Bases: `traits.has_traits.HasStrictTraits`

The workflow that is maintained in the "local" process – ie, the same one that showing a GUI.

`workflow`
The list of *`WorkflowItem`*s

`selected`
The currently-selected *`WorkflowItem`*

`modified`
Has this workflow been modified since it was loaded?

`message_q`
The `queue.Queue` of messages to send to the remote process

`recv_thread`
The `threading.Thread` that receives messages from the remote process

`send_thread`
The `threading.Thread` that sends messages to the remote process

`recv_main``(`*child_conn*`)`
The method that runs in *`recv_thread`* to receive messages from the remote process.

`send_main``(`*child_conn*`)`
The method that runs in *`send_thread`* to send messages from *`message_q`* to the remote process

`run_all``()`
Send the RUN_ALL message to the remote process

`remote_eval``(`*expr*`)`
Evaluate an expression in the remote process and return the result

`remote_exec``(`*expr*`)`
Execute an expression in the remote process and wait until it completes

`wi_sync``(`*wi*, *variable*, *value*, *timeout=30*`)`
Set *`WorkflowItem.status`* on the remote workflow, then wait for it to propogate here.

`wi_waitfor``(`*wi*, *variable*, *value*, *timeout=30*`)`
Waits a configurable amount of time for wi's status to change to status

`shutdown_remote_process``(`*remote_process*`)`
Shut down the remote process

`class` `cytoflowgui.workflow.workflow.`**`RemoteWorkflow`**
Bases: `traits.has_traits.HasStrictTraits`

The workflow that is maintained in the "remote" process – ie, the one that actually does the processing.

`workflow`
The list of *`WorkflowItem`*'s

`selected`
The currently-selected *`WorkflowItem`*

`last_view_plotted`
The last *`IWorkflowView`* that was plotted

send_thread
The threading.Thread that sends messages to the local process

recv_thread
The threading.Thread that receives messages from the local process

message_q
The queue.Queue of messages to send to the local process

matplotlib_events
threading.Event to synchronize matplotlib plotting across process boundaries

plot_lock
threading.Lock to synchronize matplotlib plotting across process boundaries

run(*parent_workflow_conn*, *parent_mpl_conn=None*)
The method that runs the main loop of the remote process

recv_main(*parent_conn*)
The method that runs in the *recv_thread* to receive messages from the local process.

send_main(*parent_conn*)
The method that runs in *send_thread* to send messages to the local process.

shutdown()
Shut down the remote process

cytoflowgui.workflow.workflow_item

Represents one step in an analysis pipeline. Wraps a single *IOperation* and any *IView* of its result.

class cytoflowgui.workflow.workflow_item.**WorkflowItem**
Bases: traits.has_traits.HasStrictTraits

The basic unit of a Workflow: wraps an operation and a list of views. This class is serialized and synchronized between the *LocalWorkflow* and the *RemoteWorkflow*.

Notes

Because we serialize instances of this, we have to pay careful attention to which traits are `transient` (and aren't serialized). Additionally, traits marked as `status` are only serialized remote –> local. For more details about the synchronization, see the module docstring for *cytoflowgui.workflow*

friendly_id
The operation's id

name
The operation's name

operation
The operation that this *WorkflowItem* wraps

views
The *IView*'s associated with this operation

current_view
The currently selected view

result
The *Experiment* that is the result of applying *operation* to the *previous_wi*'s *result*

channels
The channels from *result*

conditions
The conditions from *result*

metadata
The metadata from *result*

statistics
The statistics from *result*

default_view
The default view for this workflow item (if any)

previous_wi
The previous *WorkflowItem* in the workflow

next_wi
The next *WorkflowItem* in the workflow

workflow
The *LocalWorkflow* or *RemoteWorkflow* that this *WorkflowItem* is a part of

status
This *WorkflowItem*'s status

op_error
Errors from *operation*'s *IOperation.apply* method

op_error_trait
The trait that caused the error in *op_error*

op_warning
Warnings from *operation*'s *IOperation.apply* method

op_warning_trait
The trait that caused the warning in *op_warning*

estimate_error
Errors from *operation*'s *IOperation.estimate* method

estimate_warning
Warnings from *operation*'s *IOperation.estimate* method

view_error
Errors from the most recently plotted view's *IView.plot* method

view_error_trait
The trait that caused the error in *view_error*

view_warning
Warnings from the most recently plotted view's *IView.plot* method

view_warning_trait
The trait that caused the warning in *view_warning*

plot_names
The possible values for the **plot_name** parameter of *current_view*'s *IView.plot* method. Retrieved from that view's **enum_plots()** method and updated automatically when *result* or *current_view* changes.

plot_names_label
The GUI label for the element that allows users to select a plot name from `plot_names`. Updated automatically when `result` or `current_view` changes.

matplotlib_events
`threading.Event` to synchronize matplotlib plotting across process boundaries

plot_lock
`threading.Lock` to synchronize matplotlib plotting across process boundaries

lock
`threading.Lock` for updating this `WorkflowItem`'s traits

edit_traits(*view=None*, *parent=None*, *kind=None*, *context=None*, *handler=None*, *id=''*, *scrollable=None*, ***args*)
Override the base `traits.has_traits.HasTraits.edit_traits` to make it go looking for views in the handler.

trait_view(*name=None*, *view_element=None*, *handler=None*)
Gets or sets a ViewElement associated with an object's class.

If both *name* and *view_element* are specified, the view element is associated with *name* for the current object's class. (That is, *view_element* is added to the ViewElements object associated with the current object's class, indexed by *name*.)

If only *name* is specified, the function returns the view element object associated with *name*, or None if *name* has no associated view element. View elements retrieved by this function are those that are bound to a class attribute in the class definition, or that are associated with a name by a previous call to this method.

If neither *name* nor *view_element* is specified, the method returns a View object, based on the following order of preference:

1. If there is a View object named `traits_view` associated with the current object, it is returned.
2. If there is exactly one View object associated the current object, it is returned.
3. Otherwise, it returns a View object containing items for all the non-event trait attributes on the current object.

Parameters

- **name** (*str*) – Name of a view element
- **view_element** (*ViewElement*) – View element to associate

Return type A view element.

estimate()
Call `operation`'s `IOperation.estimate`

apply()
Calls `operation`'s `IOperation.apply`; applies this `WorkflowItem`'s operation to `previous_wi`'s result

plot()
Call `current_view`'s `IView.plot` on `result`, or on `previous_wi`'s `result` if there's no current `result`.

Submodules

cytoflowgui.cytoflow_application

The `pyface.tasks` application.

`CytoflowApplication` – the `pyface.tasks.tasks_application.TasksApplication` class for the *`cytoflow`* Qt GUI

`cytoflowgui.cytoflow_application.`**`gui_handler_callback`**(*msg*, *app*)

`class` `cytoflowgui.cytoflow_application.`**`CytoflowApplication`**(*plugins=None*, ***traits*)

Bases: `envisage.ui.tasks.tasks_application.TasksApplication`

The cytoflow Tasks application

`id`
The application's GUID

`name`
The application's user-visible name.

`default_layout`
The default window-level layout for the application.

`always_use_default_layout`
Restore the previous application-level layout?

`dpi`
The application's DPI

`debug`
Are we debugging?

`filename`
Filename from the command-line, if present

`application_error`
If there's an ERROR-level log message, drop it here

`application_log`
Keep the application log in memory

`model`
The model that's shared across both tasks

`controller`
The *`WorkflowController`*, shared across both tasks

`remote_process`
The `multiprocessing.Process` containing the remote workflow

`remote_workflow_connection`
The `multiprocessing.Pipe` to communicate with the remote process

`remote_canvas_connection`
The `multiprocessing.Pipe` to communicate with the remote `matplotlib` canvas, FigureCanvasAggRemote`.

`canvas`
The shared `matplotlib` canvas

run()
Run the application: configure logging, set up the model, controller and canvas, and initialize the GUI.

show_error(*error_string*)
GUI error handler

stop()
Overridden from `envisage.ui.tasks.tasks_application.TasksApplication` to shut down the remote process

preferences_helper
Cytoflow preferences manager

cytoflowgui.experiment_pane

A dock pane with an experiment browser.

class cytoflowgui.experiment_pane.**ExperimentBrowserDockPane**(**args: Any*, ***kwargs: Any*)
Bases: `pyface.tasks.api.TraitsDockPane`

A DockPane with a read-only view of some of the traits of the `Experiment` that results from the currently-selected `WorkflowItem`.

id = 'edu.mit.synbio.cytoflowgui.experiment_pane'

name = 'Experiment Browser'

handler = <traits.trait_types.Instance object>

closable = True

dock_area = 'left'

floatable = True

movable = True

visible = True

create_contents(*parent*)
Create and return the toolkit-specific contents of the dock pane.

cytoflowgui.experiment_pane_model

The classes that provide the model for the `ExperimentBrowserDockPane`.

class cytoflowgui.experiment_pane_model.**WorkflowItemNode**(**args: Any*, ***kwargs: Any*)
Bases: `traitsui.api.TreeNodeObject`

A tree node for the Experiment

wi = <traits.trait_types.Instance object>

label = 'WorkflowItem'

tno_get_label(_)

tno_allows_children(*node*)

tno_has_children(*node*)

tno_get_menu(_)

tno_get_children(_)

class cytoflowgui.experiment_pane_model.**ChannelsNode**(**args: Any*, ***kwargs: Any*)
Bases: traitsui.api.TreeNodeObject

A tree node for the group of channels

wi = <traits.trait_types.Instance object>

label = 'Channels'

tno_get_label(*node*)

tno_allows_children(*node*)

tno_has_children(*node*)

tno_get_menu(_)

tno_get_icon(*node*, *is_expanded*)

tno_get_children(_)

class cytoflowgui.experiment_pane_model.**ChannelNode**(**args: Any*, ***kwargs: Any*)
Bases: traitsui.api.TreeNodeObject

A tree node for a single channel

wi = <traits.trait_types.Instance object>
The `WorkflowItem` that this channel is part of

channel = <traits.trait_types.Str object>

tno_get_label(_)

tno_allows_children(_)

tno_has_children(_)

tno_get_menu(_)

tno_get_icon(*node*, *is_expanded*)

tno_get_children(_)

class cytoflowgui.experiment_pane_model.**ConditionsNode**(**args: Any*, ***kwargs: Any*)
Bases: traitsui.api.TreeNodeObject

A tree node for all the conditions

wi = <traits.trait_types.Instance object>

label = 'Conditions'

tno_get_label(_)

tno_allows_children(*node*)

tno_has_children(*node*)

tno_get_menu(_)

tno_get_icon(*node*, *is_expanded*)

tno_get_children(_)

class cytoflowgui.experiment_pane_model.**ConditionNode**(**args: Any*, ***kwargs: Any*)
Bases: traitsui.api.TreeNodeObject

A tree node for a single condition

wi = <traits.trait_types.Instance object>
The `WorkflowItem` that this condition is part of

condition = <traits.trait_types.Str object>

tno_get_label(_)

tno_allows_children(_)

tno_has_children(_)

tno_get_menu(_)

tno_get_icon(*node*, *is_expanded*)

tno_get_children(_)

class cytoflowgui.experiment_pane_model.**StatisticsNode**(**args: Any*, ***kwargs: Any*)
Bases: `traitsui.api.TreeNodeObject`

A tree node for all the statistics

wi = <traits.trait_types.Instance object>

label = 'Statistics'

tno_get_label(_)

tno_allows_children(*node*)

tno_has_children(*node*)

tno_get_menu(_)

tno_get_icon(*node*, *is_expanded*)

tno_get_children(_)

class cytoflowgui.experiment_pane_model.**StatisticNode**(**args: Any*, ***kwargs: Any*)
Bases: `traitsui.api.TreeNodeObject`

A tree node for a single statistic

wi = <traits.trait_types.Instance object>
The `WorkflowItem` that this condition is part of

statistic = <traits.trait_types.Tuple object>

tno_get_label(_)

tno_allows_children(_)

tno_has_children(_)

tno_get_menu(_)

tno_get_icon(*node*, *is_expanded*)

tno_get_children(_)

class cytoflowgui.experiment_pane_model.**StringNode**(**args: Any*, ***kwargs: Any*)
Bases: `traitsui.api.TreeNodeObject`

A tree node for strings

name = <traits.trait_types.Str object>
Name of the value

`label = <traits.trait_types.Str object>`
User-specified override of the default label

`value = <traits.trait_types.Str object>`
The value itself

`tno_allows_children(node)`

`tno_has_children(node)`

`tno_get_menu(_)`

`tno_get_icon(node, is_expanded)`

`tno_get_label(node)`
Gets the label to display for a specified object.

`format_value(value)`
Returns the formatted version of the value.

cytoflowgui.export_task

The `pyface.tasks` task that exports a figure.

ExportPane – the `pyface.tasks.traits_dock_pane.TraitsDockPane` to determine the width and height of the exported figure.

ExportTaskPane – the central pane of the task, shows the plot.

ExportTask – the `pyface.tasks.task.Task` to export a figure.

ExportFigurePlugin – the `envisage` `envisage.plugin.Plugin` that wraps *ExportTask*.

`class cytoflowgui.export_task.ExportPane(*args, **kwargs)`
Bases: `pyface.tasks.traits_dock_pane.TraitsDockPane`

Determine the width and height of the exported figure.

`id = 'edu.mit.synbio.cytoflowgui.export_pane'`

`name = 'Export'`

`task = <traits.trait_types.Instance object>`

`closable = False`

`dock_area = 'right'`

`floatable = False`

`movable = False`

`visible = True`

`default_traits_view()`

`create_contents(parent)`
Create and return the toolkit-specific contents of the dock pane.

`class cytoflowgui.export_task.ExportTaskPane(*args, **kwargs)`
Bases: `pyface.base_toolkit.Toolkit.__call__.<locals>.Unimplemented`

The center pane for the UI; contains the matplotlib canvas for plotting data views.

`id = 'edu.mit.synbio.cytoflow.export_task_pane'`

name = 'Cytometry Data Viewer'

model = <traits.trait_types.Instance object>
The shared *LocalWorkflow* model

handler = <traits.trait_types.Instance object>
The shared *WorkflowController*

layout = <traits.trait_types.Instance object>
The center window's layout

canvas = <traits.trait_types.Instance object>
The shared canvas, an instance of *FigureCanvasQTAggLocal*

create(*parent*)
Create a layout for the tab widget and the main view

activate()

class cytoflowgui.export_task.**ExportTask**
Bases: `pyface.tasks.task.Task`

classdocs

menu_bar
The menu bar schema

model
The shared *LocalWorkflow* model

handler
The shared *WorkflowController*

params_pane
Plot parameters pane

export_pane
Pane with size, DPI and buttons

width
Width, in inches

height
Height, in inches

dpi
Resolution, in dots per inch

create_central_pane()
Create and return the central pane, which must implement ITaskPane.

create_dock_panes()
Create and return the task's dock panes (IDockPane instances).

This method is called *after* create_central_pane() when the task is added to a TaskWindow.

activated()
Called after the task has been activated in a TaskWindow. Places the shared canvas in the center pane's layout.

activate_cytoflow_task(_)
Switch to the `FlowTask` task

on_export(_)
Shows a dialog to export a file

`class cytoflowgui.export_task.`**`ExportFigurePlugin`**

Bases: `envisage.plugin.Plugin`

An Envisage plugin wrapping ExportTask

`TASKS = 'envisage.ui.tasks.tasks'`

cytoflowgui.help_pane

Defines the dock pane to show a help page for the currently-selected operation or view.

`class cytoflowgui.help_pane.`**`HelpDockPane`**`(*args: Any, **kwargs: Any)`

Bases: `pyface.tasks.api.TraitsDockPane`

A `pyface.tasks.i_dock_pane.IDockPane` to view help for the current *`IWorkflowOperation`* or *`IWorkflowView`*.

`id = 'edu.mit.synbio.cytoflowgui.help_pane'`

This pane's GUID

`name = 'Help'`

This pane's name

`task = <traits.trait_types.Instance object>`

The Task that serves as the controller

`view_plugins = <traits.trait_types.List object>`

The *`IViewPlugin`*s to search for help pages

`op_plugins = <traits.trait_types.List object>`

The *`IOperationPlugin`*s to search for help pages

`help_id`

The GUID of the operation or view whose help we're currently showing

alias of `traits.trait_types.Str`

`html = <traits.trait_types.HTML object>`

The HTML trait containing the current help page

`create_contents``(parent)`

Create the pane's contents, which is just the view's UI

cytoflowgui.import_dialog

A modal dialog that allows the user to set up their experiment, mapping FCS files to experimental conditions.

There are a few main classes:

- *`Tube`* – represents one tube (well, an FCS file) in an experiment – the filename and its metadata.
- *`TubeTrait`* – represents one trait (the trait type and name)
- *`ExperimentDialogModel`* – the tabular model of tubes, traits, and trait values
- *`ExperimentDialogHandler`* – the controller which contains the view and the logic for connecting it to the model.

Additionally, there are several utility functions and classes:

- *`not_true`* and *`not_false`* – obvious

- `sanitize_metadata` – replaces all non-Python-safe characters in a tube's metadata with '_'
- `eval_bool` – interprets various strings as True or False
- `ConvertingBool` – trait type that uses `eval_bool` to convert strings to boolean values

`cytoflowgui.import_dialog.`**`not_true`**`(`*value*`)`

`cytoflowgui.import_dialog.`**`not_false`**`(`*value*`)`

`cytoflowgui.import_dialog.`**`sanitize_metadata`**`(`*meta*`)`

replaces all non-Python-safe characters in a tube's metadata with '_'

`class cytoflowgui.import_dialog.`**`Tube`**

Bases: `traits.has_traits.HasStrictTraits`

The model for a tube in an experiment.

I originally wanted to make the Tube in the ImportDialog and the Tube in the ImportOp the same, but that fell apart when I tried to implement serialization (dynamic traits don't survive pickling when sending tubes to the remote process) (well, their values do, but neither the trait type nor the metadata do.)

Oh well.

This model depends on duck-typing ("if it walks like a duck, and quacks like a duck…"). Because we want to use all of the TableEditor's nice features, each row needs to be an instance, and each column a Trait. So, each Tube instance represents a single tube, and each experimental condition (as well as the tube name, its file, and optional plate row and col) are traits. These traits are dynamically added to Tube INSTANCES (NOT THE TUBE CLASS.) Then, we add appropriate columns to the table editor to access these traits.

We also derive traits from tubes' FCS metadata. One can make a new column from metadata, then convert it into a condition to use in the analysis.

We also use the "transient" flag to specify traits that shouldn't be displayed in the editor. This matches well with the traits that every HasTraits-derived class has by default (all of which are transient.)

`index`

The tube index

`file`

The FCS filename

`parent`

Which model are we part of?

`conditions`

A Dict of the conditions (for hashing)

`metadata`

The FCS metadata

`all_conditions_set`

Do all of the conditions have a value?

`conditions_hash``()`

Return a hash of this tube's conditions (for equality testing)

`class cytoflowgui.import_dialog.`**`ExperimentColumn`**`(`**args: Any*, ***kwargs: Any*`)`

Bases: `traitsui.table_column.ObjectColumn`

A `traitsui.table_column.ObjectColumn` with setable color

`get_cell_color``(`*obj*`)`

`cytoflowgui.import_dialog.`**`eval_bool`**`(`*x*`)`

Evaluate "f", "false", "n" or "no" as `False`; and "t", "true", "y" or "yes" as `True`

`class` `cytoflowgui.import_dialog.`**`ConvertingBool`**`(`*default_value=<traits.trait_type._NoDefaultSpecifiedType object>*, ***metadata*`)`

Bases: `traits.trait_types.BaseCBool`

A trait that converts "f", "false", "n", or "no" to `False` and "t", "true", "y" or "yes" to `True`

`evaluate``()`

Evaluate "f", "false", "n" or "no" as `False`; and "t", "true", "y" or "yes" as `True`

`validate``(_,` *name*, *value*`)`

Validates that a specified value is valid for this trait.

Note: The 'fast validator' version performs this check in C.

`class` `cytoflowgui.import_dialog.`**`TubeTrait`**

Bases: `traits.has_traits.HasStrictTraits`

A class representing a trait on a tube. A trait has an underlying `traits.trait_type.TraitType`, a name, a type ("metadata", "category", "float" or "bool"), and a default view.

`model`

Which model are we a part of?

`name`

The name of the trait

`type`

What type of trait is it?

`class` `cytoflowgui.import_dialog.`**`ExperimentDialogModel`**

Bases: `traits.has_traits.HasStrictTraits`

The model for the Experiment setup dialog.

`tubes`

T he list of Tubes (rows in the table)

`tube_traits`

A list of the traits that have been added to Tube instances (columns in the table)

`tube_traits_dict`

A dictionary of trait name –> TubeTrait instances

`counter`

A dictionary of tube hash –> # of tubes with that hash; keeps track of whether a tube is unique or not

`valid`

Are all the tubes unique and filled?

`dummy_experiment`

A dummy Experiment, with the first Tube and no events, so we can check subsequent tubes for voltage etc. and fail early.

`init``(`*import_op*`)`

Initializes the model from a pre-existing *`ImportOp`*

`update_import_op``(`*import_op*`)`

Update an *`ImportOp`* with the information in this dialog

`is_tube_unique``(`*tube*`)`

Is a tube unique?

class `cytoflowgui.import_dialog.`**`ExperimentDialogHandler`**(**args: Any*, ***kwargs: Any*)

Bases: `traitsui.api.Controller`

A controller that contains the import dialog's view and the logic that connects it to the *`ExperimentDialogModel`*

`import_op = <traits.trait_types.Instance object>`

`add_tubes`

alias of `traits.trait_types.Event`

`remove_tubes`

alias of `traits.trait_types.Event`

`add_variable`

alias of `traits.trait_types.Event`

`import_csv`

alias of `traits.trait_types.Event`

`selected_tubes`

alias of `traits.trait_types.List`

`table_editor = <traits.trait_types.Instance object>`

`updating = <traits.trait_types.Bool object>`

`init`(*info*)

`close`(*info*, *is_ok*)

Handles the user attempting to close a dialog-based user interface.

This method is called when the user attempts to close a window, by clicking an **OK** or **Cancel** button, or clicking a Close control on the window). It is called before the window is actually destroyed. Override this method to perform any checks before closing a window.

While Traits UI handles "OK" and "Cancel" events automatically, you can use the value of the *is_ok* parameter to implement additional behavior.

Parameters

- **info** (*UIInfo object*) – The UIInfo object associated with the view
- **is_ok** (*Boolean*) – Indicates whether the user confirmed the changes (such as by clicking **OK**.)

Returns **allow_close** – A Boolean, indicating whether the window should be allowed to close.

Return type bool

`closed`(*info*, *is_ok*)

cytoflowgui.matplotlib_backend_local

A matplotlib backend that renders across a process boundary. This module has the "local" canvas – the part that actually renders to a (Qt) window.

By default, matplotlib only works in one thread. For a GUI application, this is a problem because when matplotlib is working (ie, scaling a bunch of data points) the GUI freezes.

This module and *`matplotlib_backend_remote`* implement a matplotlib backend where the plotting done in one process (ie via pyplot, etc) shows up in a canvas running in another process (the GUI). The canvas is the interface across the process boundary: a "local" canvas, which is a GUI widget (in this case a QWidget) and a "remote" canvas (running in the process where pyplot.plot() etc. are used.) The remote canvas is a subclass of the Agg renderer; when draw() is called, the remote canvas pulls the current buffer out of the renderer and pushes it through a pipe to the local canvas, which draws it on the screen. blit() is implemented too.

This takes care of one direction of data flow, and would be enough if we were just plotting. However, we want to use matplotlib widgets as well, which means there's data flowing from the local canvas to the remote canvas too. The local canvas is a subclass of `matplotlib.backends.backend_qt5agg.FigureCanvasQTAgg`, which is itself a sublcass of QWidget. The local canvas overrides several of the event handlers, passing the event information to the remote canvas which in turn runs the matplotlib event handlers.

class `cytoflowgui.matplotlib_backend_local.`**`Msg`**

Bases: `object`

Messages sent between the local and remote canvases. There is an identical class in *`matplotlib_backend_remote`* because we don't want these two modules requiring one another

`DRAW = 'DRAW'`

`BLIT = 'BLIT'`

`WORKING = 'WORKING'`

`RESIZE_EVENT = 'RESIZE'`

`MOUSE_PRESS_EVENT = 'MOUSE_PRESS'`

`MOUSE_MOVE_EVENT = 'MOUSE_MOVE'`

`MOUSE_RELEASE_EVENT = 'MOUSE_RELEASE'`

`MOUSE_DOUBLE_CLICK_EVENT = 'MOUSE_DOUBLE_CLICK'`

`DPI = 'DPI'`

`PRINT = 'PRINT'`

`cytoflowgui.matplotlib_backend_local.`**`log_exception`**`()`

Catch and log exceptions (with their tracebacks

class `cytoflowgui.matplotlib_backend_local.`**`FigureCanvasQTAggLocal`**`(`*figure*, *child_conn*, *working_pixmap*`)`

Bases: `matplotlib.backends.backend_qtagg.FigureCanvasQTAgg`

The local canvas; ie, the one in the GUI.

`listen_for_remote``()`

The main method for the thread that listens for messages from the remote canvas

`send_to_remote``()`

The main method for the thread that sends messages to the remote canvas

`leaveEvent``(`*event*`)`

Override the Qt event leaveEvent

`mousePressEvent``(`*event*`)`

Override the Qt event mousePressEvent

`mouseDoubleClickEvent``(`*event*`)`

Override the Qt event mouseDoubleClickEvent

`mouseMoveEvent``(`*event*`)`

Override the Qt event mouseMoveEvent

`mouseReleaseEvent``(`*event*`)`

Override the Qt event mouseReleaseEvent

resizeEvent(*event*)
Override the Qt event resizeEvent

paintEvent(*e*)
Copy the image from the buffer to the qt.drawable. In Qt, all drawing should be done inside of here when a widget is shown onscreen.

print_figure(**args*, ***kwargs*)
Pass a "print" request to the remote canvas (actually this is for rastering a figure and saving it to disk)

cytoflowgui.matplotlib_backend_remote

A matplotlib backend that renders across a process boundary. This file has the "remote" canvas – the Agg renderer into which pyplot.plot() renders.

By default, matplotlib only works in one thread. For a GUI application, this is a problem because when matplotlib is working (ie, scaling a bunch of data points) the GUI freezes.

This module implements a matplotlib backend where the plotting done in one process (ie via pyplot, etc) shows up in a canvas running in another process (the GUI). The canvas is the interface across the process boundary: a "local" canvas, which is a GUI widget (in this case a QWidget) and a "remote" canvas (running in the process where pyplot.plot() etc. are used.) The remote canvas is a subclass of the Agg renderer; when draw() is called, the remote canvas pulls the current buffer out of the renderer and pushes it through a pipe to the local canvas, which draws it on the screen. blit() is implemented too.

This takes care of one direction of data flow, and would be enough if we were just plotting. However, we want to use matplotlib widgets as well, which means there's data flowing from the local canvas to the remote canvas too. The local canvas is a subclass of FigureCanvasQTAgg, which is itself a sublcass of QWidget. The local canvas overrides several of the event handlers, passing the event information to the remote canvas which in turn runs the matplotlib event handlers.

class cytoflowgui.matplotlib_backend_remote.**Msg**
Bases: `object`

Messages sent between the local and remote canvases. There is an identical class in *matplotlib_backend_local* because we don't want these two modules requiring one another

DRAW = 'DRAW'

BLIT = 'BLIT'

WORKING = 'WORKING'

RESIZE_EVENT = 'RESIZE'

MOUSE_PRESS_EVENT = 'MOUSE_PRESS'

MOUSE_MOVE_EVENT = 'MOUSE_MOVE'

MOUSE_RELEASE_EVENT = 'MOUSE_RELEASE'

MOUSE_DOUBLE_CLICK_EVENT = 'MOUSE_DOUBLE_CLICK'

DPI = 'DPI'

PRINT = 'PRINT'

cytoflowgui.matplotlib_backend_remote.**log_exception**()
Catch and log exceptions (with their tracebacks

class `cytoflowgui.matplotlib_backend_remote.`**`FigureCanvasAggRemote`**(*parent_conn*, *process_events*, *plot_lock*, *figure*)

Bases: `matplotlib.backends.backend_agg.FigureCanvasAgg`

The canvas the figure renders into in the remote process (ie, the one where someone is calling pyplot.plot())

`listen_for_local`()

The main method for the thread that listens for messages from the local canvas

`send_to_local`()

The main method for the thread that sends messages to the local canvas

`draw`(**args*, ***kwargs*)

When the canvas is instructed to draw itself, copy the Agg buffer out to a numpy array and send it to the local process.

`blit`(*bbox=None*)

When instructed to blit a bounding box, copy the region in the bounding box to a numpy array and send it to the local canvas.

`set_working`(*working*)

`cytoflowgui.matplotlib_backend_remote.`**`new_figure_manager`**(*num*, **args*, ***kwargs*)

Create a new figure manager instance. This maintains the remote canvas as a singleton – else, each new canvas would need a copy of the pipes, locks, etc.

`cytoflowgui.matplotlib_backend_remote.`**`draw_if_interactive`**()

`cytoflowgui.matplotlib_backend_remote.`**`show`**()

`cytoflowgui.matplotlib_backend_remote.`**`FigureCanvas`**

alias of *`cytoflowgui.matplotlib_backend_remote.FigureCanvasAggRemote`*

`cytoflowgui.matplotlib_backend_remote.`**`tight_layout`**(*self*, **args*, ***kwargs*)

cytoflowgui.preferences

Skeleton preferences manager. At the moment, the only preference is whether to always use the default application layout.

class `cytoflowgui.preferences.`**`CytoflowPreferences`**(***traits*)

Bases: `apptools.preferences.preferences_helper.PreferencesHelper`

The preferences helper for the Cytoflow application.

`preferences_path`

The path to the preference node that contains the preferences.

`always_use_default_layout`

Whether to always apply the default application layout.

class `cytoflowgui.preferences.`**`CytoflowPreferencesPane`**(**args: Any*, ***kwargs: Any*)

Bases: `envisage.ui.tasks.preferences_pane.PreferencesPane`

The preferences pane for the Cytoflow application.

`model_factory`

The factory to use for creating the preferences model object.

alias of *`cytoflowgui.preferences.CytoflowPreferences`*

`view`

The view for the preferences dialog box

cytoflowgui.run

The entry-point for the GUI – sets up and starts the remote process, configures logging, loads the Envisage plugins, and starts the GUI loop.

`cytoflowgui.run.`**`log_notification_handler`**`(_, `*`trait_name`*`, `*`old`*`, `*`new`*`)`
Exception handler for traits notifications

`cytoflowgui.run.`**`log_excepthook`**`(`*`typ`*`, `*`val`*`, `*`tb`*`)`
Exception handler for global exceptions

`cytoflowgui.run.`**`run_gui`**`()`
Run the GUI!

`cytoflowgui.run.`**`monitor_remote_process`**`(`*`proc`*`)`
The main method for the (local) thread that monitors the remote process

`cytoflowgui.run.`**`start_remote_process`**`()`
Start the remote process. Creates pipes and synchronization primitives, sets up logging, and starts the remote process and the monitoring thread.

`cytoflowgui.run.`**`remote_main`**`(`*`parent_workflow_conn`*`, `*`parent_mpl_conn`*`, `*`log_q`*`, `*`running_event`*`)`
The main method for the remote process. Configures logging, sets up the matplotlib backend, and instantiates the *`RemoteWorkflow`*.

cytoflowgui.subset_controllers

Instances of `traitsui.handler.Controller` for the various *`ISubset`* classes. These contain the `traitsui.view.Views` for them. Also a utility function to return the appropriate handler for an arbitrary *`ISubset`* model.

`class` `cytoflowgui.subset_controllers.`**`BoolSubsetHandler`**`(`**`*`*args: Any*`, `**`**`*kwargs: Any*`)`
Bases: `traitsui.api.Controller`

Controller for *`BoolSubset`*

`subset_view``()`

`class` `cytoflowgui.subset_controllers.`**`CategorySubsetHandler`**`(`**`*`*args: Any*`, `**`**`*kwargs: Any*`)`
Bases: `traitsui.api.Controller`

Controller for *`CategorySubset`*

`subset_view``()`

`class` `cytoflowgui.subset_controllers.`**`RangeSubsetHandler`**`(`**`*`*args: Any*`, `**`**`*kwargs: Any*`)`
Bases: `traitsui.api.Controller`

Controller for *`RangeSubset`*

`subset_view``()`

`cytoflowgui.subset_controllers.`**`subset_handler_factory`**`(`*`model`*`)`
A factory method to produce the right handler for a given implementation of *`ISubset`*

cytoflowgui.util

A few utility classes for `cytoflowgui`

class `cytoflowgui.util.`**`DefaultFileDialog`**(**args: Any*, ***kwargs: Any*)

Bases: `pyface.ui.qt4.file_dialog.FileDialog`

A `pyface.ui.qt4.file_dialog.FileDialog` with a default suffix

`default_suffix`

alias of `traits.trait_types.Str`

class `cytoflowgui.util.`**`HintedMainWindow`**(**args: Any*, ***kwargs: Any*)

Bases: `pyface.qt.QtGui.QMainWindow`

When pyface makes a new dock pane, it sets the width and height as fixed (from the new layout or from the default). Then, after it's finished setting up, it resets the minimum and maximum widget sizes. In Qt5, this triggers a re-layout according to the widgets' hinted sizes. So, here we keep track of "fixed" sizes, then return those sizes as the size hint to the layout engine.

`hint_width = None`

`hint_height = None`

`setFixedWidth`(**args*, ***kwargs*)

`setFixedHeight`(**args*, ***kwargs*)

`sizeHint`(**args*, ***kwargs*)

class `cytoflowgui.util.`**`HintedWidget`**(**args: Any*, ***kwargs: Any*)

Bases: `pyface.qt.QtGui.QWidget`

When pyface makes a new widget, it sets the width and height as fixed (from the new layout or from the default). Then, after it's finished setting up, it resets the minimum and maximum widget sizes. In Qt5, this triggers a re-layout according to the widgets' hinted sizes. So, here we keep track of "fixed" sizes, then return those sizes as the size hint to the layout engine.

`hint_width = None`

`hint_height = None`

`setFixedWidth`(**args*, ***kwargs*)

`setFixedHeight`(**args*, ***kwargs*)

`sizeHint`(**args*, ***kwargs*)

cytoflowgui.view_pane

Dock panes for modifying an `IWorkflowView`s traits and the parameters that are passed to `IView.plot`.

- `ViewDockPane` – the dock pane to manipulate the traits of the currently selected view.
- `PlotParamsPane` – the dock pane to manipulate the parameters passed to `IView.plot`.

class `cytoflowgui.view_pane.`**`ViewDockPane`**(**args*, ***kwargs*)

Bases: `pyface.tasks.traits_dock_pane.TraitsDockPane`

A DockPane to manipulate the traits of the currently selected view.

`id = 'edu.mit.synbio.cytoflowgui.view_traits_pane'`

`name = 'View Properties'`

task = <traits.trait_types.Instance object>

view_plugins = <traits.trait_types.List object>

handler = <traits.trait_types.Instance object>

image_size = <traits.trait_types.Tuple object>

create_contents(*parent*)
Create and return the toolkit-specific contents of the dock pane.

class cytoflowgui.view_pane.**PlotParamsPane**(**args*, ***kwargs*)
Bases: `pyface.tasks.traits_dock_pane.TraitsDockPane`

id = 'edu.mit.synbio.cytoflowgui.params_pane'

name = 'Plot Parameters'

handler = <traits.trait_types.Instance object>

closable = True

dock_area = 'right'

floatable = True

movable = True

visible = True

create_contents(*parent*)
Create and return the toolkit-specific contents of the dock pane.

cytoflowgui.workflow_controller

Controllers for *`LocalWorkflow`* and the *`WorkflowItem`*s it contains – these dynamically create the `traitsui.view.View` instances for the workflow's operations and views.

Perhaps the most confusing thing in the entire codebase is the way that these views are created. The difficulty is that a view for a *`WorkflowItem`* is polymorphic depending on the *`IWorkflowOperation`* that it wraps and the *`IWorkflowView`* that is currently active.

Here's how this works. The "Workflow" pane contains a view of the *`LocalWorkflow`* (the model), created by *`WorkflowController.workflow_traits_view`*. The editor is a *`VerticalNotebookEditor`*, configured to use *`WorkflowController.handler_factory`* to create a new *`WorkflowItemHandler`* for each *`WorkflowItem`* in the *`LocalWorkflow`*.

Here's our opportunity for polymorphism! Because each *`WorkflowItem`* has its own *`WorkflowItemHandler`* instance, the *`WorkflowItemHandler.operation_traits_view`* method can return a `traitsui.view.View` *specifically for that `WorkflowItem``'s operation.* The `traitsui.view.View` it returns contains an *`InstanceHandlerEditor`*, which uses `WorkflowItemHandler._get_operation_handler` to get a handler specifically for the *`IWorkflowOperation`* that this *`WorkflowItem`* wraps. And this handler, in turn, creates the view specifically for the *`IWorkflowOperation`* (and contains the logic for connecting it to the *`IWorkflowOperation`* that is its model.)

The logic for the view traits and view parameters panes is similar. Each pane contains a view of *`LocalWorkflow.selected`*, the currently-selected *`WorkflowItem`*. In turn that *`WorkflowItem`*'s handler creates a view for the currently-displayed *`IWorkflowView`* (which is in *`WorkflowItem.current_view`*). This handler, in turn, creates a view for the *`IWorkflowView`*'s traits or plot parameters.

One last non-obvious thing. Many of the operations and views require choosing a value from the `Experiment`. For example, if an `IWorkflowView` is plotting a histogram, one of its traits is the channel whose histogram is being plotted. These values – channels, conditions, statistics and numeric statistics, for both the current `WorkflowItem.result` and the previous `WorkflowItem.result` – are presented as properties of the `WorkflowItemHandler`. In turn, the `WorkflowItemHandler` appears in the view's context dictionary as `context`. So, if a view wants to get the previous `WorkflowItem`'s channels, it can refer to them as `context.channels`. (Examples of this pattern are scattered throughout the submodules of `view_plugins`.

class `cytoflowgui.workflow_controller.`**`WorkflowItemHandler`**(**args: Any*, ***kwargs: Any*)

Bases: `traitsui.api.Controller`

A controller for a `WorkflowItem`. It dynamically creates views for the `IWorkflowOperation` and `IWorkflowView`s that are contained, as well as exposing channels, conditions, statistics and numeric statistics as properties (so they can be accessed by the views.

`deletable = <traits.traits.ForwardProperty object>`

For the vertical notebook view, is this page deletable?

`icon = <traits.traits.ForwardProperty object>`

The icon for the vertical notebook view

`name = <traits.trait_types.DelegatesTo object>`

`friendly_id = <traits.trait_types.DelegatesTo object>`

`op_plugins`

alias of `traits.trait_types.List`

`view_plugins`

alias of `traits.trait_types.List`

`conditions = <traits.traits.ForwardProperty object>`

The conditions in this `WorkflowItem.result`

`conditions_names = <traits.traits.ForwardProperty object>`

The names of the conditions in this `WorkflowItem.result`

`previous_conditions = <traits.traits.ForwardProperty object>`

The conditions in the previous `WorkflowItem.result`

`previous_conditions_names = <traits.traits.ForwardProperty object>`

The names of the conditions in the previous `WorkflowItem.result`

`statistics_names = <traits.traits.ForwardProperty object>`

The names of the statistics in this `WorkflowItem.result`

`numeric_statistics_names = <traits.traits.ForwardProperty object>`

The names of the numeric statistics in this `WorkflowItem.result`

`previous_statistics_names = <traits.traits.ForwardProperty object>`

The names of the statistics in the previous `WorkflowItem.result`

`channels = <traits.traits.ForwardProperty object>`

The channels in this `WorkflowItem.result`

`previous_channels = <traits.traits.ForwardProperty object>`

The channels in the previous `WorkflowItem.result`

`tree_node = <traits.traits.ForwardProperty object>`

`operation_traits_view`()

Returns the `traitsui.view.View` of the `IWorkflowOperation` that this `WorkflowItem` wraps. The view is actually defined by the operation's handler's `operation_traits_view` attribute.

view_traits_view()
Returns the traitsui.view.View showing the traits of the current *IWorkflowView*. The view is actually defined by the view's handler's view_traits_view attribute.

view_params_view()
Returns the traitsui.view.View showing the plot parameters of the current *IWorkflowView*. The view is actually defined by the view's handler's view_params_view attribute.

view_plot_name_view()
Returns the traitsui.view.View showing the plot names of the current *IWorkflowView*. The view is actually defined by the view's handler's view_plot_name_view attribute.

experiment_view()
Returns a traitsui.view.View of *LocalWorkflow.selected*, showing some things about the experiment – channels, conditions, statistics, etc.

class cytoflowgui.workflow_controller.**WorkflowController**(**args: Any*, ***kwargs: Any*)
Bases: traitsui.api.Controller

A controller for a *LocalWorkflow*. It dynamically creates views for the major panes in the UI: the workflow, the selected view traits, and the selected view parameters. It also contains the logic for adding operations and activating views. Both of which are triggered by the button bars on the sides of their respective panes.

workflow_handlers = <traits.trait_types.Dict object>

op_plugins
alias of traits.trait_types.List

view_plugins
alias of traits.trait_types.List

workflow_traits_view()
Returns a traitsui.view.View of the *LocalWorkflow* for the Workflow pane. Its editor is a *VerticalNotebookEditor*. Each item's instance view is created by *WorkflowItemHandler.operation_traits_view*.

selected_view_traits_view()
Returns a traitsui.view.View of *LocalWorkflow.selected* for the View traits pane. The actual view is created by *WorkflowItemHandler.view_traits_view*.

selected_view_params_view()
Returns a traitsui.view.View of *LocalWorkflow.selected* for the View parameters pane. The actual view is created by *WorkflowItemHandler.view_params_view*.

selected_view_plot_name_view()
Returns a traitsui.view.View of *LocalWorkflow.selected* for the plot names toolbar. The actual view is created by *WorkflowItemHandler.view_plot_name_view*.

experiment_view()
Returns a traitsui.view.View of *LocalWorkflow.selected* for the experiment viewer.

handler_factory(*wi*)
Return an instance of *WorkflowItemHandler* for a *WorkflowItem* in *LocalWorkflow*

add_operation(*operation_id*)
The logic to add an *IWorkflowOperation* to *LocalWorkflow*. Creates a new *WorkflowItem*, figures out where to add it, inserts it into the model and activates the default view (if present.)

activate_view(*view_id*)
The logic to activate a view on the selected *WorkflowItem*. Creates a new instance of the view if necessary and makes it the current view; event handlers on *WorkflowItem.current_view* take care of everything else.

cytoflowgui.workflow_pane

The pane that has the operation toolbar and the workflow.

`class cytoflowgui.workflow_pane.`**`WorkflowDockPane`**`(`*`*args`*`, `*`**kwargs`*`)`

Bases: `pyface.tasks.traits_dock_pane.TraitsDockPane`

Workflow dock pane

`id = 'edu.mit.synbio.cytoflowgui.workflow_pane'`

`name = 'Workflow'`

`plugins = <traits.trait_types.List object>`

`handler = <traits.trait_types.Instance object>`

`image_size = <traits.trait_types.Tuple object>`

`create_contents``(`*`parent`*`)`

Create and return the toolkit-specific contents of the dock pane.

Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX