FlowCal Documentation

Release 1.3.0

S. Castillo-Hair, J. Sexton, B. Landry, E. Olson, J. Tabor

Jan 25, 2021

Contents

`FlowCal` is a library for reading, analyzing, and calibrating flow cytometry data in Python. It features:

- Extraction of Flow Cytometry Standard (FCS) files into numpy array-like structures
- Traditional and non-standard gating, including automatic density-based two-dimensional gating.
- Transformation functions that allow conversion of data from raw FCS channel numbers to arbitrary fluorescence units (a.u.).
- Plotting, including generation of histograms, density plots and scatter plots.

Most importantly, `FlowCal` automatically processes calibration beads data in order to convert fluorescence to calibrated units, **Molecules of Equivalent Fluorophore (MEF)**. The most important advantages of using MEF are 1) fluorescence can be reported independently of acquisition settings, and 2) variation in data due to instrument shift is eliminated.

Finally, `FlowCal` includes a user-fiendly Excel User Interface to perform all of these operations automatically, without the need to write any code.

CHAPTER 1

Cite FlowCal

If you use `FlowCal` in your research, we would appreciate citations to the following article:

> Castillo-Hair S.M., Sexton J.T., *et al*. FlowCal: A User-Friendly, Open Source Software Tool for Automatically Converting Flow Cytometry Data from Arbitrary to Calibrated Units.. ACS Synth. Biol. 2016.

CHAPTER 2

Table of Contents

2.1 Getting Started

`FlowCal` requires the Python programming language. We recommend most Windows and macOS users to *install FlowCal with Anaconda*, a Python distribution that already includes many necessary Python packages. macOS already includes its own version of Python, but it does not include some Python tools that `FlowCal` requires. Therefore, Anaconda is recommended.

Users who have an existing python installation and are comfortable with command-line interfaces can *install FlowCal in their existing environment*.

2.1.1 Installing FlowCal with Anaconda

To install Anaconda and `FlowCal`, do the following:

1. Navigate to this page and scroll down to the "Anaconda Installers" section. Click on the "Graphical Installer" link below the name of your operating system (Windows, MacOS, or Linux). This will download the installer.

Note: **Windows**: If your computer is a 32-bit PC, click on the message "32-Bit Graphical Installer" instead of the "Download" button. If you don't know whether yours is a 32 or 64-bit computer but you have purchased it in the last five years, it is probably a 64-bit computer and you can ignore this message.

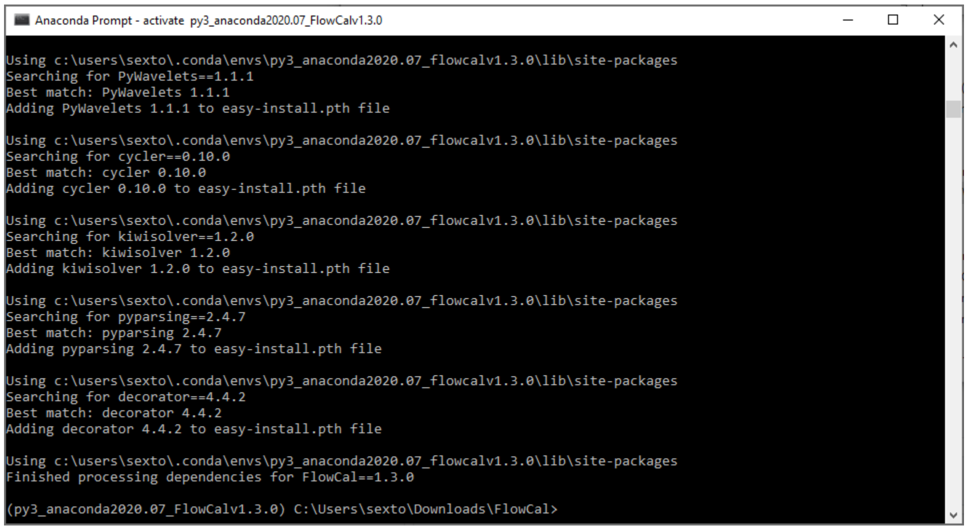
Note: Python 2.7 is also supported. However, we recommend downloading the Python 3.8 version of Anaconda.

2. Double click the installer (.exe in Windows, .pkg in OS X) and follow the instructions on screen.

Note: **Windows**: During installation, on the "Advanced Installation Options" screen, make sure to check both "Add Anaconda to my PATH environment variable" and "Register Anaconda as my default Python". Recent versions of

Anaconda suggest to keep the first option unchecked. However, this option is necessary for the installation script on step 4 to work.

3. Download `FlowCal` from here. A file called `FlowCal-master.zip` will be downloaded. Unzip this file.
4. Inside the unzipped folder, double click on `Install FlowCal (Windows).bat` or `Install FlowCal (macOS)` if you are using Windows or OS X, respectively. This will open a terminal window and install `FlowCal`. The installation procedure may take a few minutes. When installation is finished, the terminal will show the message "Press Enter to finish…". If the installation was successful, your terminal should look like the figure below. Press Enter to close the terminal window.


Anaconda Prompt - activate py3_anaconda2020.07_FlowCalv1.3.0

```
Using c:\users\sexto\.conda\envs\py3_anaconda2020.07_flowcalv1.3.0\lib\site-packages
Searching for PyWavelets==1.1.1
Best match: PyWavelets 1.1.1
Adding PyWavelets 1.1.1 to easy-install.pth file

Using c:\users\sexto\.conda\envs\py3_anaconda2020.07_flowcalv1.3.0\lib\site-packages
Searching for cycler==0.10.0
Best match: cycler 0.10.0
Adding cycler 0.10.0 to easy-install.pth file

Using c:\users\sexto\.conda\envs\py3_anaconda2020.07_flowcalv1.3.0\lib\site-packages
Searching for kiwisolver==1.2.0
Best match: kiwisolver 1.2.0
Adding kiwisolver 1.2.0 to easy-install.pth file

Using c:\users\sexto\.conda\envs\py3_anaconda2020.07_flowcalv1.3.0\lib\site-packages
Searching for pyparsing==2.4.7
Best match: pyparsing 2.4.7
Adding pyparsing 2.4.7 to easy-install.pth file

Using c:\users\sexto\.conda\envs\py3_anaconda2020.07_flowcalv1.3.0\lib\site-packages
Searching for decorator==4.4.2
Best match: decorator 4.4.2
Adding decorator 4.4.2 to easy-install.pth file

Using c:\users\sexto\.conda\envs\py3_anaconda2020.07_flowcalv1.3.0\lib\site-packages
Finished processing dependencies for FlowCal==1.3.0

(py3_anaconda2020.07_FlowCalv1.3.0) C:\Users\sexto\Downloads\FlowCal>
```

Note: **Windows**: If the following message appears after double clicking `Install FlowCal (Windows)`: "Windows protected your PC – Windows SmartScreen prevented an unrecognized app from starting…", click on the "More info" link under the text, and then click on the "Run anyway" button. This will remove the security restriction from the program and allow it to run properly.

Note: **Mac OS X**: If the following error message appears after double clicking `Install FlowCal (macOS)`: "'Install FlowCal (macOS)' can't be opened because it is from an unidentified developer.", navigate to System Preferences -> Security and Privacy -> General, and click the "Open Anyways" button adjacent to the message stating "'Install FlowCal (macOS)' was blocked from opening because it is not from an identified developer". This will remove the security restriction from the program and allow it to run properly.

To see `FlowCal` in action, head to the *Excel UI* section. The `FlowCal` zip file includes an `examples` folder with files that you can use while following the instructions.

2.1.2 Installing FlowCal in an Existing Python Evironment

Python (2.7, 3.6, 3.7, or 3.8) is required, along with `pip` and `setuptools`. The easiest way is to install `FlowCal` is to use `pip`:

```
pip install FlowCal
```

This should take care of all the requirements automatically. Linux and macOS users may need to request administrative permissions by preceding this command with `sudo`.

Alternatively, download `FlowCal` from here. Next, make sure that the following Python packages are present:

- `packaging` (>=16.8)
- `six` (>=1.10.0)
- `numpy` (>=1.9.0)
- `scipy` (>=0.19.0)
- `matplotlib` (>=2.0.0)
- `scikit-image` (>=0.10.0)
- `scikit-learn` (>=0.16.0)
- `pandas` (>=0.23.0)
- `xlrd` (>=0.9.2,<2.0.0)
- `openpyxl` (>=2.2.0)

If you have `pip`, a `requirements.txt` file is provided, such that the required packages can be installed by running:

```
pip install -r requirements.txt
```

To install `FlowCal`, run the following in `FlowCal`'s root directory:

```
python setup.py install
```

Again, some users may need to precede the previous commands with `sudo`.

Note: Ubuntu/Linux Mint: `FlowCal` might need more recent versions of some python packages than the ones provided via `apt`. To upgrade these, some non-python packages need to be installed in your system. On freshly installed systems, the following packages may need to be manually installed:

- `gcc`
- `g++`
- `gfortran`
- `libblas-dev`
- `liblapack-dev`
- `libfreetype6-dev`
- `python-dev`
- `python-tk`
- `python-pip`

All of these can be installed using:

```
sudo apt install <package-name>
```

Next, `pip` should be upgraded by using:

```
sudo pip install --upgrade pip
```

After this, you may install `FlowCal` by following the steps above.

2.2 Fundamentals

Here we explain the fundamentals of two of the main features of `FlowCal`: conversion of fluorescence to calibrated units, and automatic gating of flow cytometry data based on density.

2.2.1 Calibration

Introduction to Calibration and MEF

Fluorescence data obtained via flow cytometry is frequently reported in arbitrary units (a.u.), which have the following issues:

1. Fluorescence values in a.u depend on the instrument used.
2. Even when using the same instrument, fluorescence values in a.u. depend on the acquisition settings used.
3. Even when these two are kept constant, fluorescence values in a.u. can change in time due to instrument drift.

Because of this, the only meaningful results based on flow cytometry that are frequently presented are ratios of measured reporter in two different conditions (i.e. fold-change). However, absolute levels of reporter cannot be quantitatively compared across laboratories, or between different different biological systems that require different acquisition settings, and not even between different samples of the same system taken by the same person across large periods of time.

To compensate for some of these effects, manufacturers provide calibration particles. These are a mixture of 4-8 subpopulations of microbeads, each one containing different amounts of a certain fluorophore. The fluorescence of each subpopulation is specified by the manufacturer in **Molecules of Equivalent Fluorophore (MEF)**, the number of fluorophores in solution that result in the same fluorescence as one microbead. Calibration particles can then be measured in every experiment to obtain the fluorescence of each subpopulation in a.u. Using these fluorescence values and the MEF values provided by the manufacturer, one can construct a standard curve that maps fluorescence from a.u. to MEF. This standard curve can then be used to convert the fluorescence of cell samples to MEF.

Expressing fluorescence of cellular samples in MEF automatically eliminates issues 2 and 3. Issue 1 is also eliminated if the calibration beads' fluorophore is the same as the one used in cellular samples. If not, instrument-dependence can still be eliminated by performing a one-time calibration using a common cellular sample. At the very least, transforming to MEF makes cellular samples inside a laboratory comparable.

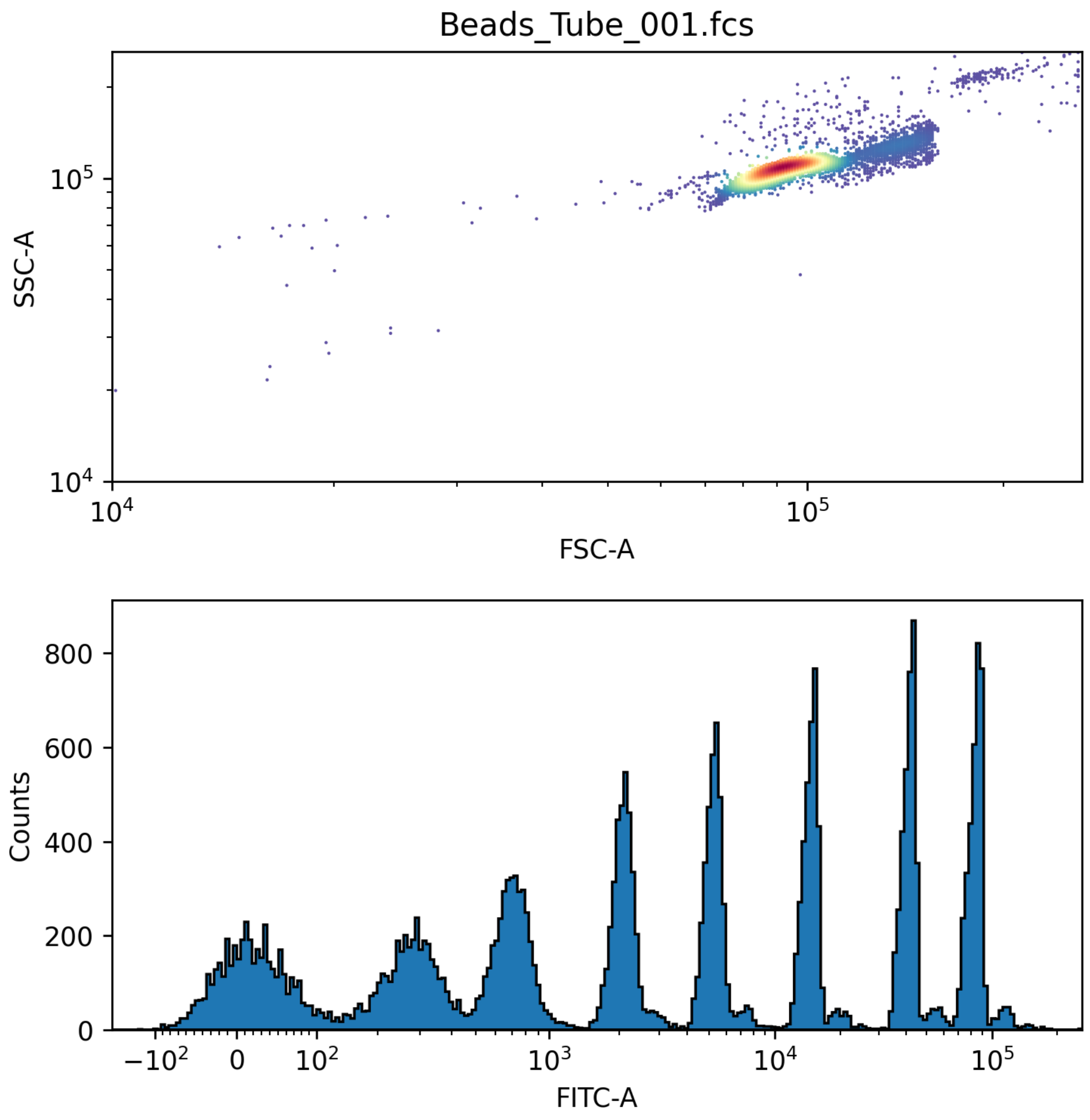
The Process of MEF Calibration

We will now give a short description of the process that `FlowCal` uses to calibrate fluorescence data to MEF, and show some of the plots produced in the process. A discussion on the exact figures generated by the Excel UI and how to use these to debug common problems can be found *here*. A more technical discussion of the MEF calibration procedure from the perspective of *`FlowCal.mef.get_transform_fxn()`*, the function that does most of the calibration work, can be found *here*.

To perform MEF calibration, the following steps are typically followed:

1. Measurement of Calibration Beads

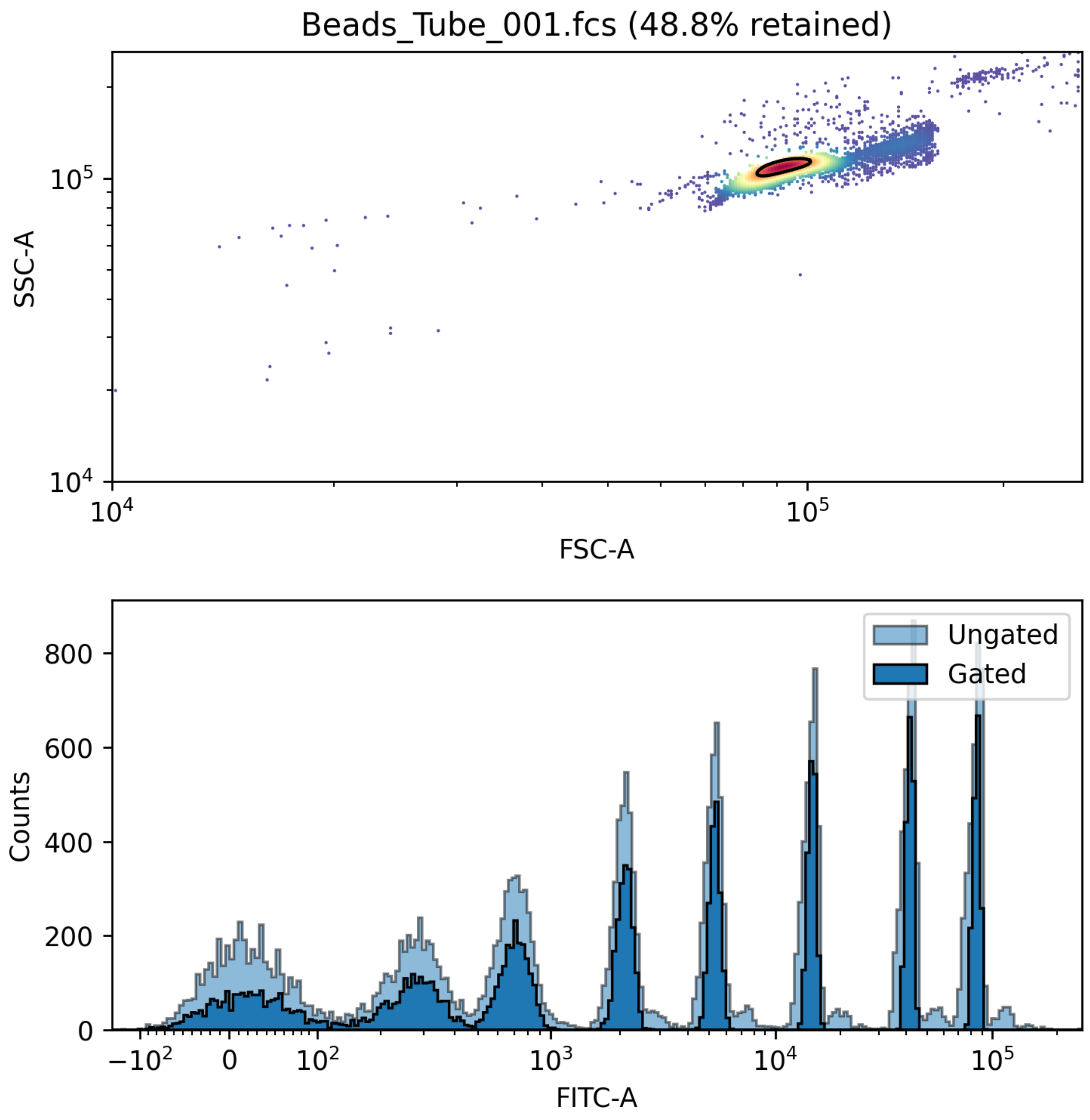
Calibration beads must be measured in every experiment, using the same acquisition settings as when measuring cell samples. The figure below shows typical flow cytometry data from calibration beads.



The top subfigure shows data from the forward/side scatter channels, whereas the bottom one shows one of the fluorescence channels. Note how several populations with different fluorescence values are evident in the bottom plot.

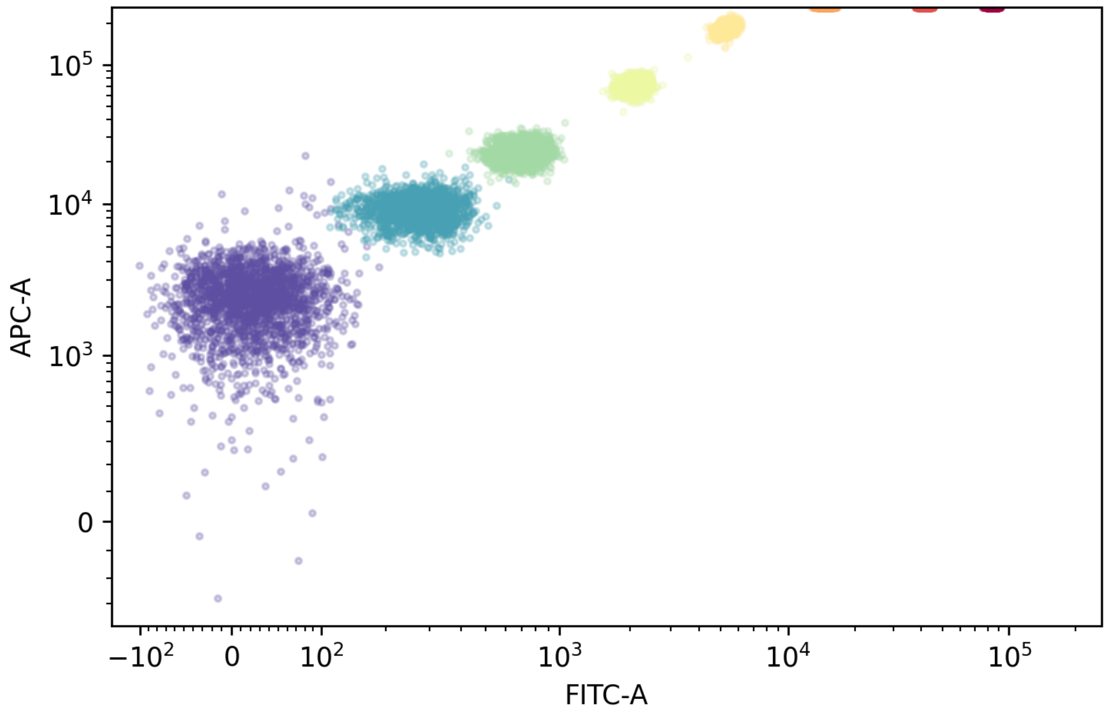
2. Elimination of Bead Aggregates and Other Debris

Notice, in the figure above, that two different populations are present in the forward/side scatter plot. The faint population on the right/upper portion of the plot corresponds to bead aggregates. These are obviously undesired, as we are only interested in single bead fluorescence. This sort of situation is normally dealt with by "gating", which involves manually drawing a region of interest and retaining the events that fall inside. `FlowCal` performs *density gating*, an automated procedure to eliminate aggregates and other events that are clearly different from the main population of interest. The figure below shows a black contour surrounding the region identified by density gating in the forward/side scatter plot, showing that density gating can distinguish single beads from aggregates. Notice also how small peaks in the fluorescence plot disappear after density-gating, which is consistent with the eliminated population being composed of agglomerations of multiple beads.



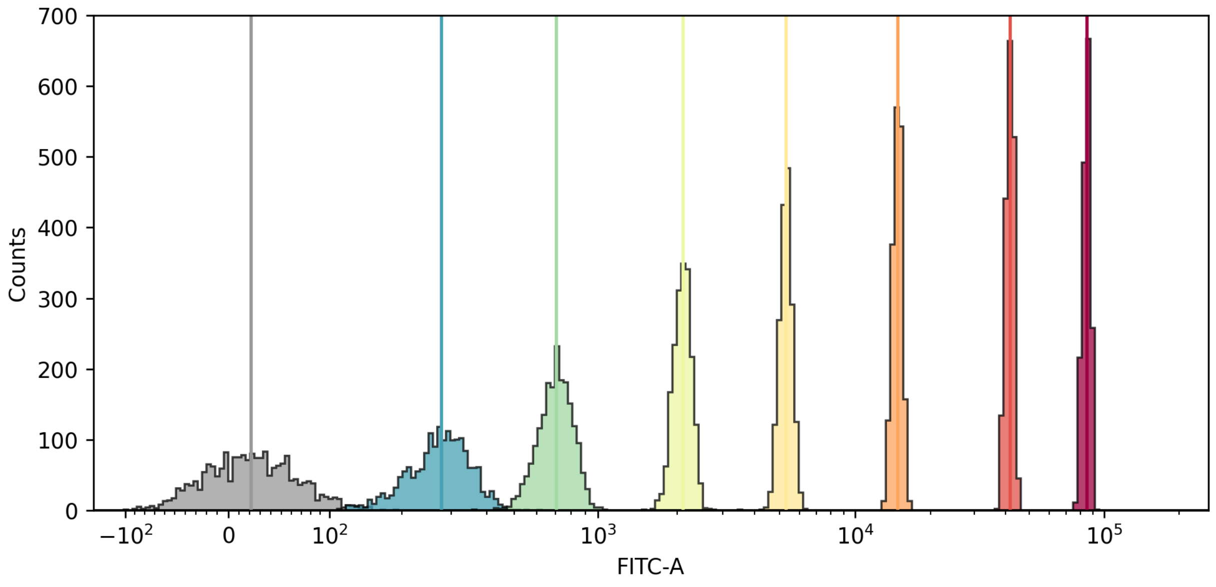
3. Identification of Bead Subpopulations

In order to calculate the average fluorescence of each subpopulation, the individual events corresponding to each must first be identified. The figure below shows one of the plots produced by `FlowCal` after an automated clustering algorithm has properly identified each subpopulation. Note how this can be achieved using information from several fluorescence channels at the same time.



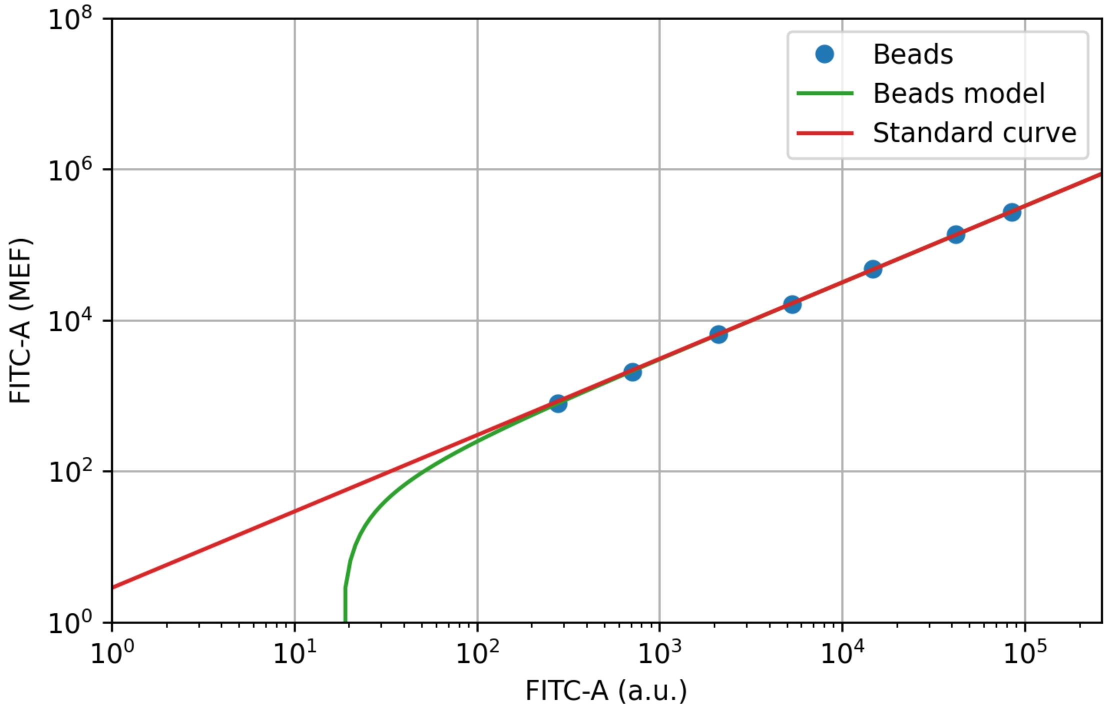
Next, the average fluorescence of each subpopulation is calculated. Some subpopulations, however, can have fluorescence values that are outside the limit of detection of the instrument, and therefore their events will show saturated fluorescence values. These subpopulations should not be considered further in the analysis. `FlowCal` discards these automatically.

The figure below shows the individual subpopulations with a vertical line representing their median fluorescence. In addition, subpopulations that were automatically discarded are shown colored in gray.



4. Calculation of a Standard Curve

Having the fluorescence of the individual populations, as measured by the flow cytometer, and the MEF values provided by the manufacturer, a standard curve can be calculated to transform fluorescence of any event to MEF. The figure below shows an example of such a standard curve. `FlowCal` uses the concept of a "bead fluorescence model", which is directly fitted to bead data but not immediately applicable to cells. However, some small mathematical manipulations turn this bead fluorescence model into a standard curve that is readily applicable to cells.



5. Conversion of Cell Fluorescence to MEF

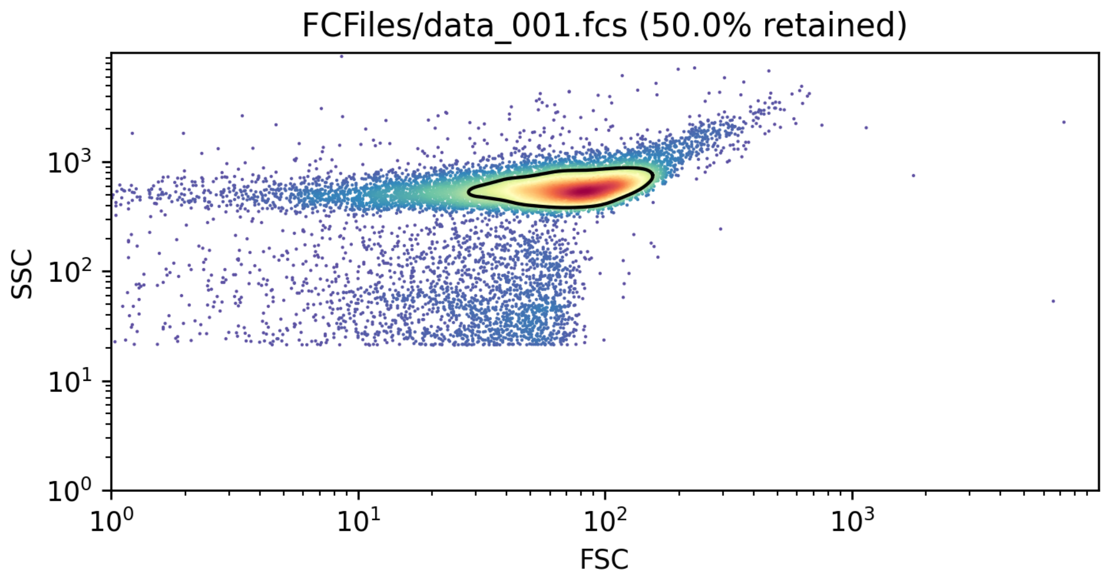
Finally, the fluorescence of any cell sample can be turned into MEF by using the standard curve obtained above.

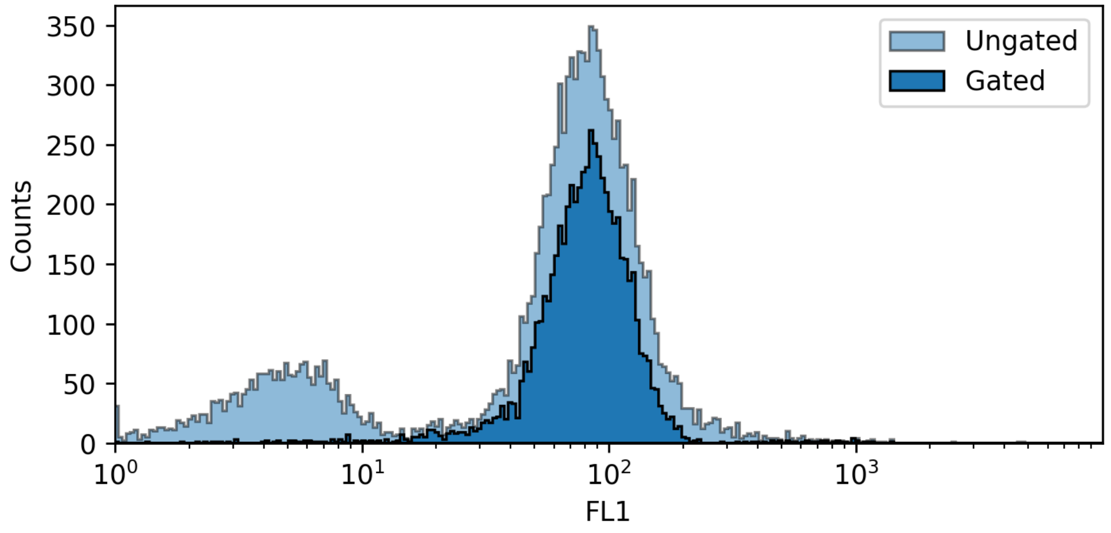
2.2.2 Density Gating

Description

Density gating looks at two channels of flow cytometry data, and discards events that are clearly different from the main population in the sample. Density gating is applied mostly to the forward/side scatter channels in `FlowCal`. When doing this, single microbeads or cells can be separated from aggregates and non-bead or non-biological debris, even when these events are a substantial fraction of the total count.

In the figure below, a sample was acquired with an intentionally low side-scatter threshold to allow a significant number of events corresponding to non-biological debris. Density gating was then applied to retain 50% of the events in the densest region. Because cells have a more uniform size than the observed debris, density gating retains mostly cells, which is reflected in the fact that FL1 fluorescence is bimodal before gating, but not after.





Note: The sample shown above was intentionally acquired with a low threshold value in `SSC` to show the capabilities of density gating. Normally, a lot of the debris can be eliminated by simply selecting a higher `SSC` threshold. However, density gating is still an excellent method to clean the data and eliminate all the debris that a simple threshold cannot filter. In our experience, this can still be a significant fraction of the total event count, especially if the cell culture has low density.

Algorithm

Density gating is implemented in the function *`FlowCal.gate.density2d()`*. In short, this function:

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.
5. Returns the gated event list.

2.3 FlowCal's Excel UI

`FlowCal`'s Excel UI allows for easy processing of flow cytometry data from a set of FCS files without having to write any code. The user simply writes an *Excel file* listing the samples to be analyzed, along with some options. FlowCal then *processes* those samples and produces *plots* and *statistics*, which can then be used in subsequent analyses. Calibration beads data can be included to report results in *calibrated MEF units*.

2.3.1 How to use FlowCal's Excel UI

To use the `FlowCal`'s Excel UI, follow these steps:

1. Make and save an Excel file indicating the FCS files to process. Click *here* for information on how to make a properly formatted input Excel file.
2. Launch `FlowCal`'s Excel UI by double clicking on `Run FlowCal (Windows).bat` or `Run FlowCal (macOS)`.
3. A window will appear requesting an input Excel file. Locate the Excel file made in step 1 and click on "Open".
4. `FlowCal` will start *processing* the indicated calibration beads and cell samples. A terminal window will appear indicating the progress of the analysis.
5. When the analysis finishes, the message "Press Enter to finish…" will appear. Press Enter and close the terminal window. A set of *plots* and *an Excel file* with statistics will appear in the same directory in which the input Excel file was located.

2.3.2 Format of the Input Excel File

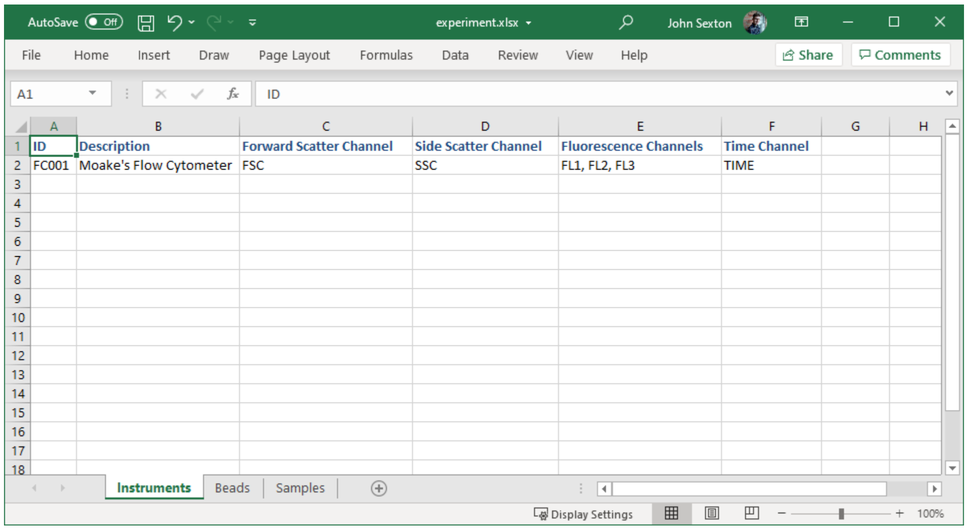
`FlowCal`'s Excel interface requires a properly formatted Excel file that depicts the samples to be analyzed and the data processing parameters. The Excel input file must have at least three sheets, named **Instruments**, **Beads**, and **Samples**. Other sheets may be present, but `FlowCal` will ignore them.

Warning: Sheet and column names are case-sensitive.

An example of a properly formatted Excel input file is provided in the `examples` folder of `FlowCal`. The following sections describe the format of the input Excel file, while using the example file as a guide.

Instruments sheet

This sheet must be filled with basic information about the flow cytometer used to acquire the samples. Each row represents an instrument. Typically, the user would only need to specify one instrument. However, `FlowCal` allows the simultaneous processing of samples taken with different instruments. The figure below shows an example of an **Instruments** sheet.



ID	Description	Forward Scatter Channel	Side Scatter Channel	Fluorescence Channels	Time Channel
FC001	Moake's Flow Cytometer	FSC	SSC	FL1, FL2, FL3	TIME

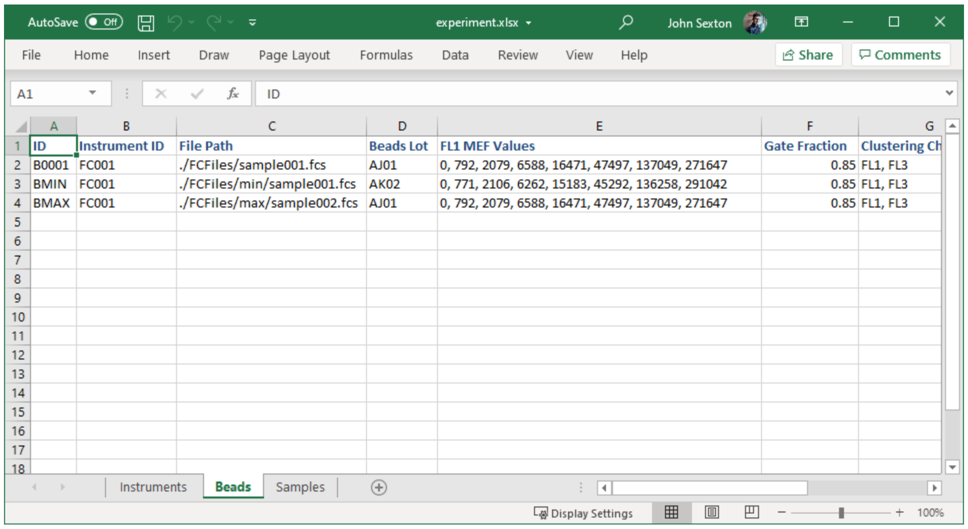
For each row, the following columns must be filled.

1. **ID** (column A in the figure above) used to reference the instrument from the other sheets. Each row must have a unique ID.
2. **Forward Scatter Channel** (C) and **Side Scatter Channel** (D): the names of these channels exactly as they appear in the acquisition software.
3. **Fluorescence channels** (E): The names of the relevant fluorescence channels as a comma-separated list, exactly as they appear in the acquisition software.
4. **Time Channel** (F): The name of the channel registering the time of each event. The FCS standard dictates that this should be called "Time", but some non-standard files may use a different name. This can be found in the acquisition software.

Additional columns, like **Description** (B in the figure above), can be added in any place for the user's records, and will be copied unmodified to the output Excel file by `FlowCal`.

Beads sheet

This sheet contains details about calibration microbeads and how to process them. Each row represents a different sample of beads. The figure below shows an example of an **Beads** sheet.



	A	B	C	D	E	F	G
1	ID	Instrument ID	File Path	Beads Lot	FL1 MEF Values	Gate Fraction	Clustering Ch
2	B0001	FC001	./FCFiles/sample001.fcs	AJ01	0, 792, 2079, 6588, 16471, 47497, 137049, 271647	0.85	FL1, FL3
3	BMIN	FC001	./FCFiles/min/sample001.fcs	AK02	0, 771, 2106, 6262, 15183, 45292, 136258, 291042	0.85	FL1, FL3
4	BMAX	FC001	./FCFiles/max/sample002.fcs	AJ01	0, 792, 2079, 6588, 16471, 47497, 137049, 271647	0.85	FL1, FL3

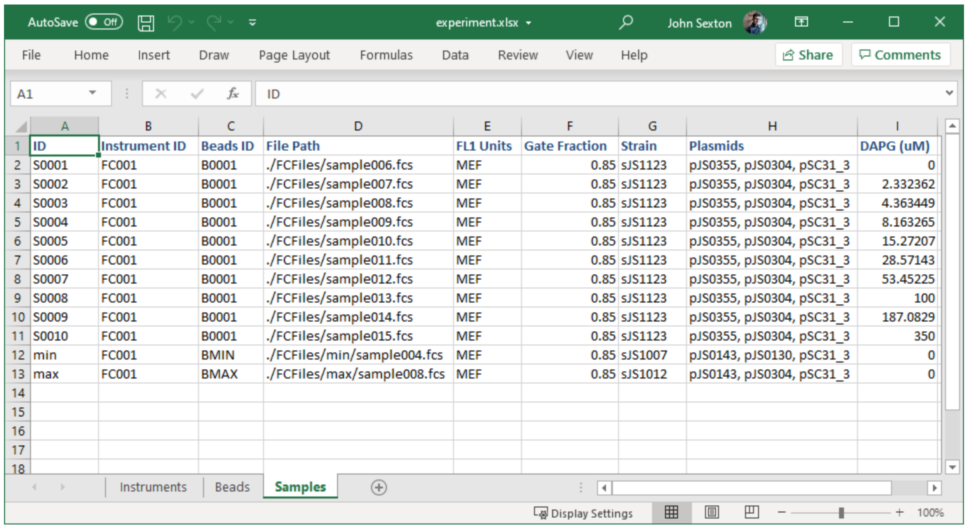
For each row, the following columns must be filled:

1. **ID** (column A in the figure above): used to reference the beads sample from the Samples sheet, and to name the figures produced by `FlowCal`. Each row must have a unique ID.
2. **Instrument ID** (B): The ID of the instrument used to take the sample.
3. **File Path** (C): the name of the corresponding FCS file.
4. **<Channel name> MEF Values** (E): MEF values provided by the manufacturer, for each channel in which a standard curve must be calculated. If MEF values are provided for a channel, the corresponding instrument should include this channel name in the **Fluorescence Channels** field. More **<Channel name> MEF Values** columns can be added if needed, or removed if not used.
5. **Gate Fraction** (F): a gate fraction parameter used for *density gating*.
6. **Clustering Channels** (G): the fluorescence channels used for clustering, as a comma separated list.

Additional columns, like **Beads Lot** (column D), can be added in any place for the user's records, and will be copied unmodified to the output Excel file by `FlowCal`.

Samples sheet

In this sheet, the user specifies cell samples and tells `FlowCal` how to process them. Each row contains the information used in the analysis of one FCS file. One file can be analyzed several times with different options (e.g. gating fractions or fluorescence units) by adding more rows that reference the same file. The figure below shows an example of a **Samples** sheet.



ID	Instrument ID	Beads ID	File Path	FL1 Units	Gate Fraction	Strain	Plasmids	DAPG (uM)
S0001	FC001	B0001	./FCFiles/sample006.fcs	MEF	0.85	sJS1123	pJS0355, pJS0304, pSC31_3	0
S0002	FC001	B0001	./FCFiles/sample007.fcs	MEF	0.85	sJS1123	pJS0355, pJS0304, pSC31_3	2.332362
S0003	FC001	B0001	./FCFiles/sample008.fcs	MEF	0.85	sJS1123	pJS0355, pJS0304, pSC31_3	4.363449
S0004	FC001	B0001	./FCFiles/sample009.fcs	MEF	0.85	sJS1123	pJS0355, pJS0304, pSC31_3	8.163265
S0005	FC001	B0001	./FCFiles/sample010.fcs	MEF	0.85	sJS1123	pJS0355, pJS0304, pSC31_3	15.27207
S0006	FC001	B0001	./FCFiles/sample011.fcs	MEF	0.85	sJS1123	pJS0355, pJS0304, pSC31_3	28.57143
S0007	FC001	B0001	./FCFiles/sample012.fcs	MEF	0.85	sJS1123	pJS0355, pJS0304, pSC31_3	53.45225
S0008	FC001	B0001	./FCFiles/sample013.fcs	MEF	0.85	sJS1123	pJS0355, pJS0304, pSC31_3	100
S0009	FC001	B0001	./FCFiles/sample014.fcs	MEF	0.85	sJS1123	pJS0355, pJS0304, pSC31_3	187.0829
S0010	FC001	B0001	./FCFiles/sample015.fcs	MEF	0.85	sJS1123	pJS0355, pJS0304, pSC31_3	350
min	FC001	BMIN	./FCFiles/min/sample004.fcs	MEF	0.85	sJS1007	pJS0143, pJS0130, pSC31_3	0
max	FC001	BMAX	./FCFiles/max/sample008.fcs	MEF	0.85	sJS1012	pJS0143, pJS0304, pSC31_3	0

For each row, the following columns must be filled:

1. **ID** (column A in the figure above): used to reference the sample while generating figures, and in the output Excel file. Each row must have a unique ID.
2. **Instrument ID** (B): The ID of the instrument used to take the sample.
3. **Beads ID** (C): The ID of the beads sample that will be used to perform the MEF transformation. Can be left blank if MEF units are not desired.
4. **File Path** (D): the name of the corresponding FCS file.
5. **<Channel name> Units** (E): The units in which to report statistics and make plots, for each fluorescence channel. If left blank, no statistics or plots will be made for that channel. More of these columns can be added or removed if necessary. If this field is specified for a channel, the corresponding instrument should include this channel in its **Fluorescence Channels** field. The available options are:
   a. **Channel**: Raw "Channel Number" units, exactly as they are stored in the FCS file.
   b. **RFI** or **a.u.**: Relative Fluorescence Intensity units, also known as Arbitrary Units.
   c. **MEF**: MEF units.
6. **Gate Fraction** (F): Fraction of samples to keep when performing *density gating*.

Additional columns, such as **Strain**, **Plasmid**, and **DAPG (uM)** (columns G, H, and I), can be added in any place for the user's records, and will be copied unmodified to the output Excel file by `FlowCal`.

Warning: If MEF units are requested for a fluorescence channel of a sample, an FCS file with calibration beads data should be specified in the **Beads ID** column. Both beads and samples should have been acquired at the same settings for the specified fluorescence channel, otherwise `FlowCal` will throw an error.

2.3.3 Analysis Performed by the Excel UI

The analysis that FlowCal's Excel UI performs is divided roughly in two phases: processing of calibration beads and processing of samples. We will now describe the steps involved in each.

Processing of Calibration Beads

The following steps are performed for each calibration beads sample specified in the **Beads** sheet of the *input Excel file*:

1. *Density gating* is applied in the forward/side scatter channels. This is an automated procedure that eliminates microbead aggregates and debris.
2. The individual microbead subpopulations are identified using automated clustering.
3. For each subpopulation, the median fluorescence is calculated.
4. Microbead subpopulations are discarded if they are found to be close to the saturation limits of the detector. Only populations that are not saturating are retained.
5. Using the fluorescence values of the retained populations in channel units and the corresponding MEF values provided by the user, a standard curve is generated. This standard curve is used to transform cell fluorescence from raw units to MEF.

Plots are generated for each one of these steps, and some intermediate results are saved to the *output Excel file*.

For an introductory discussion of flow cytometry calibration, go to the *fundamentals of calibration* section.

Processing of Cell Samples

The following steps are performed for each cell sample specified in the **Samples** sheet of the *input Excel file*:

1. *Density gating* is applied in the forward/side scatter channels.
2. Fluorescence data for each specified fluorescence channel is transformed to the units specified in the **Units** column of the *input Excel file*.
3. *Statistics* of the specified fluorescence channels are calculated, including mean, standard deviation, and others. A histogram of each fluorescence channel is also generated.

Statistics and histograms are saved to the *output Excel file*.

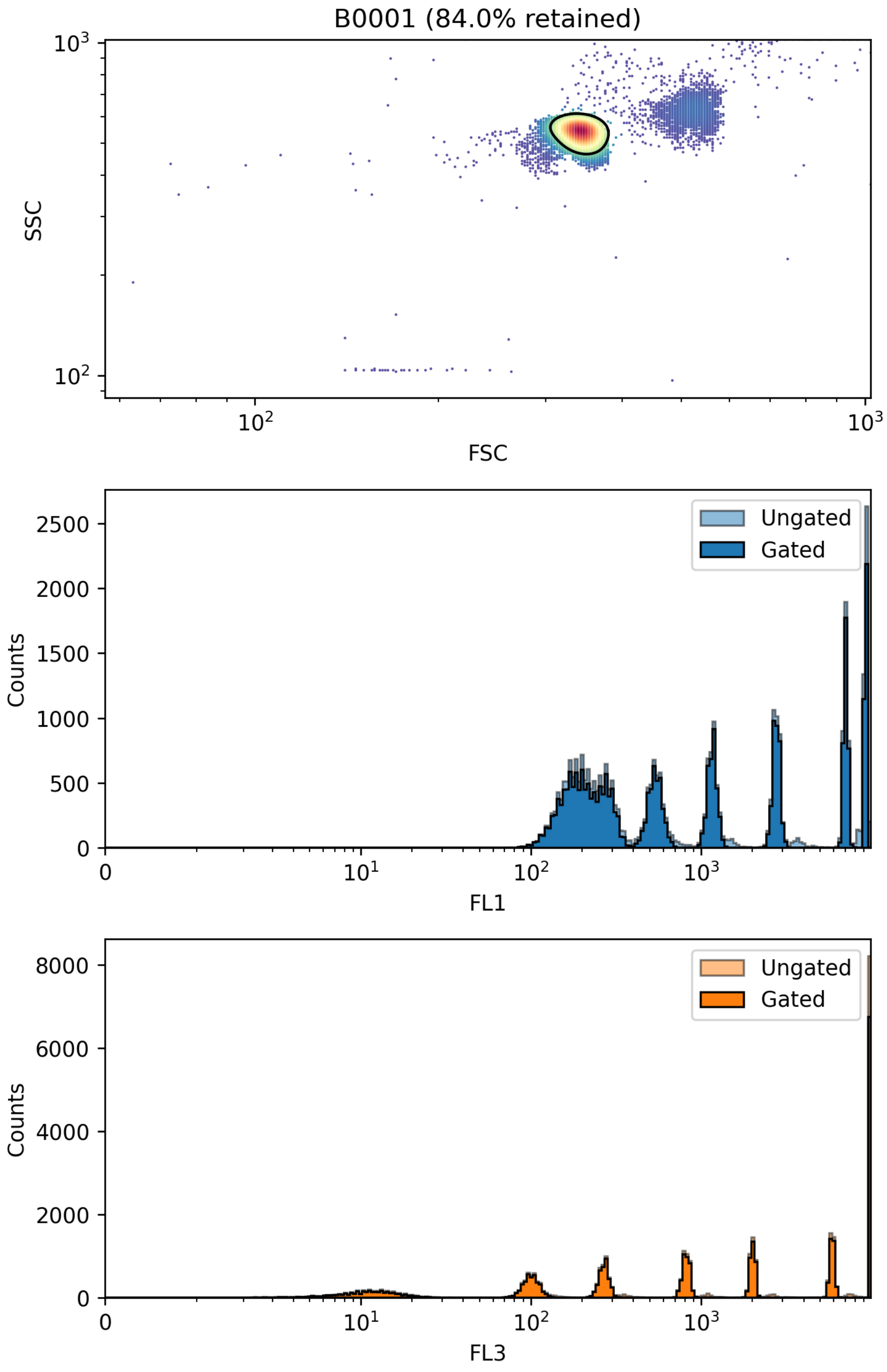
2.3.4 Outputs of the Excel UI

During processing of the calibration beads and cell samples, `FlowCal` creates two folders with images and an output Excel file in the same location as the *input Excel file*. Here we describe these. In what follows, <ID> refers to the value specified in the ID column of the input Excel file.

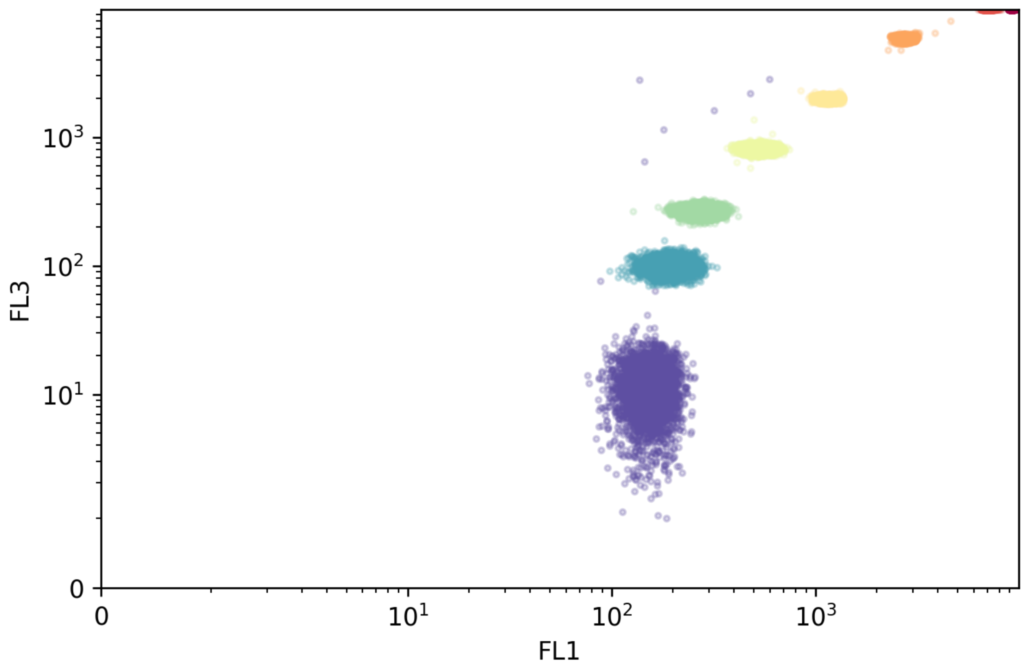
Plots

1. The folder `plot_beads` contains plots of the individual steps of processing of the calibration particle samples:

    a. `density_hist_<ID>.png`: A forward/side scatter 2D density diagram of the calibration particle sample, and a histogram for each relevant fluorescence channel.

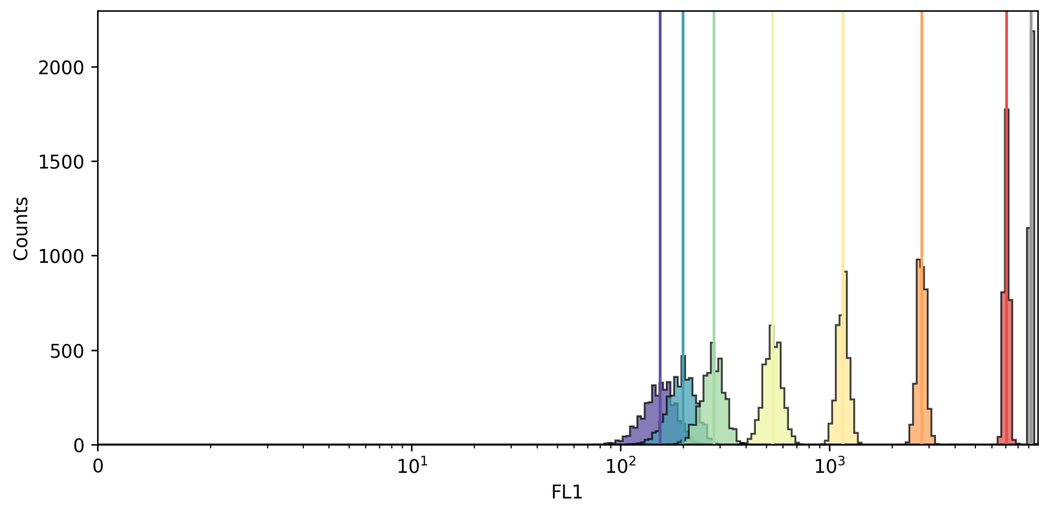


b. `clustering_<ID>.png`: A plot of the sub-populations identified during the clustering step, where the different sub-populations are shown in different colors. Depending on the number of channels used for clustering, this plot is a histogram (when using only one channel), a 2D scatter plot (when using two channels), or a 3D scatter plot with three 2D projections (when using three channels or more).



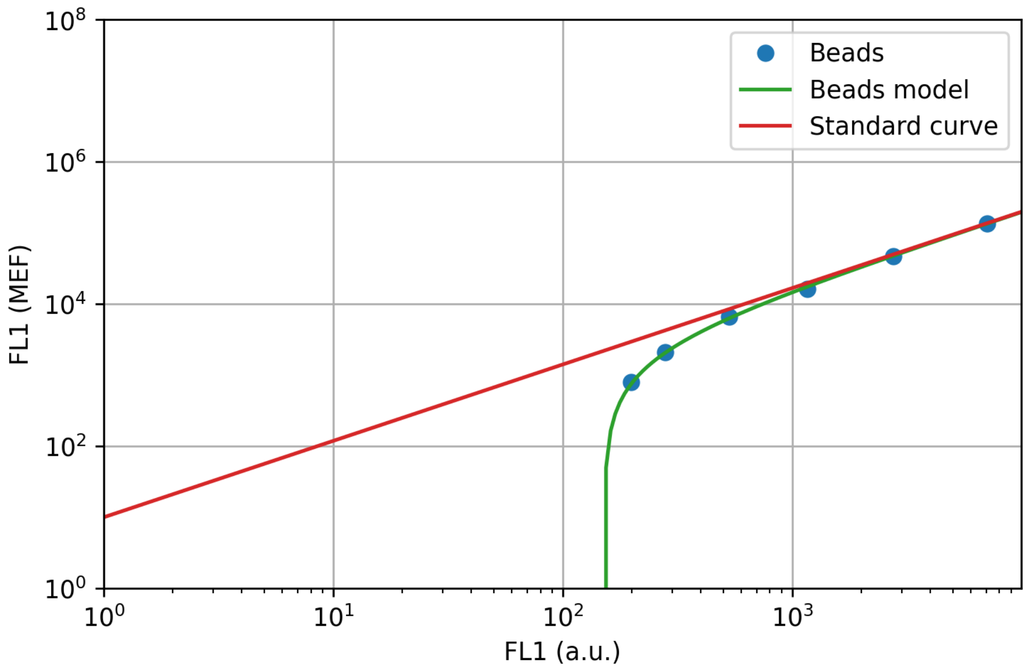
Note: It is normally easy to distinguish the different bead populations in this plot, and the different colors should correspond to this expectation. If the populations have been identified incorrectly, changing the number of channels used for clustering or the density gate fraction can improve the results. These two parameters can be changed in the **Beads** sheet of the input Excel file.

c. `populations_<channel>_<ID>.png`: A histogram showing the identified microbead sub-populations in different colors, for each fluorescence channel in which a MEF standard curve is to be calculated. In addition, a vertical line is shown representing the median of each population, which is later used to calculate the standard curve. Sub-populations that were not used to generate the standard curve are shown in gray.



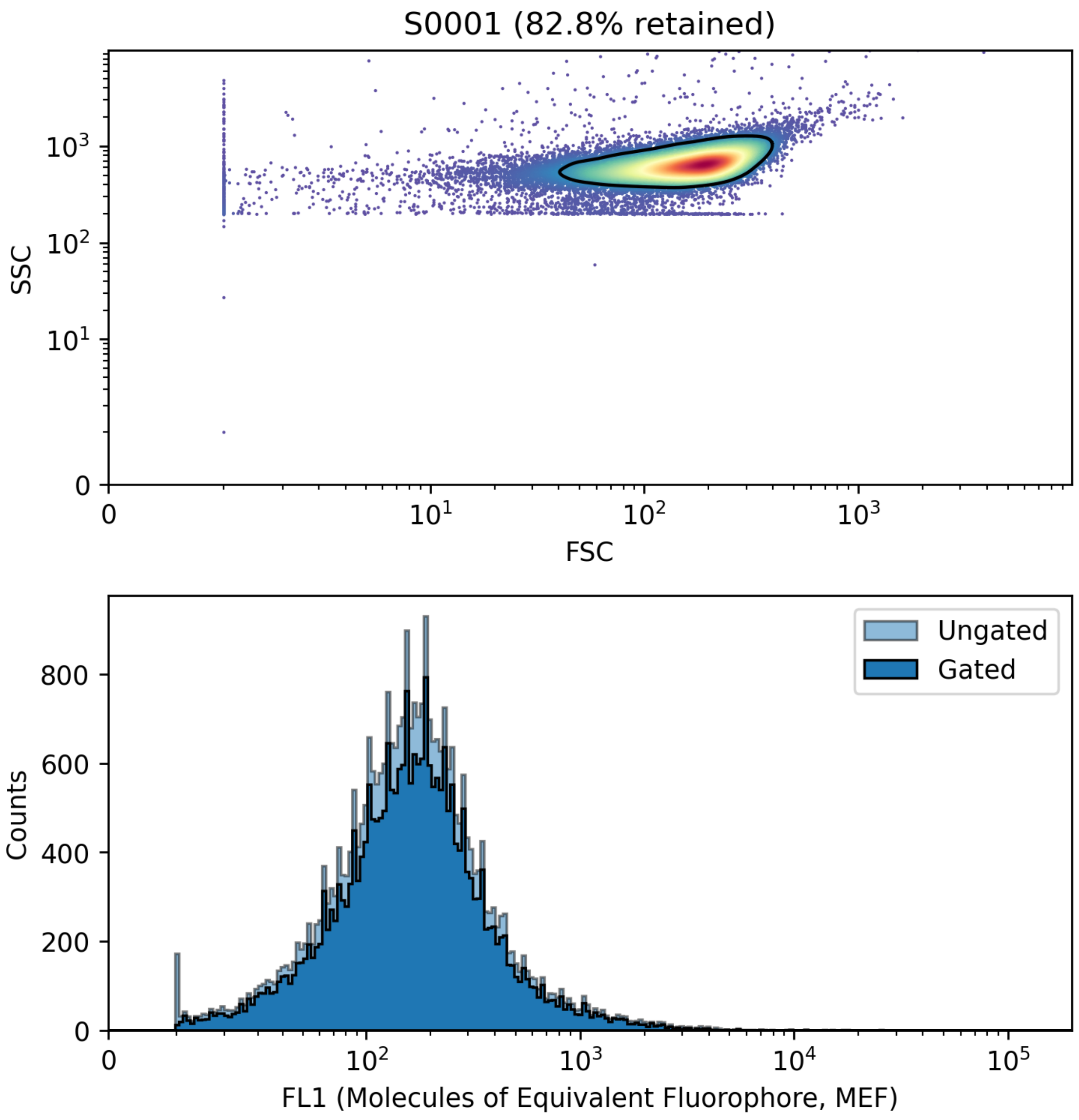
Note: All populations should be unimodal. Bimodal populations indicate incorrect clustering. This can be fixed by changing the number of channels used for clustering or the density gate fraction in the **Beads** sheet of the input Excel file.

d. `std_crv_<channel>_<ID>.png`: A plot of the fitted standard curve, for each channel in which MEF values were specified.



Note: All the blue dots should line almost perfectly on the green line, otherwise the estimation of the standard curve might not be good. If this is not the case, you should make sure that clustering is being performed correctly by looking at the previous plots. If one dot differs significantly from the curve despite perfect clustering, you might want to manually remove it. This can be done by replacing its MEF value with the word "None" in the **Beads** sheet of the input Excel file.

2. The folder `plot_samples` contains plots of the experimental cell samples. Each experimental sample of name "ID" as specified in the Excel input sheet results in a file named `<ID>.png`. This image contains a forward/side scatter 2D density diagram with the gated region indicated, and a histogram for each user-specified fluorescence channel.

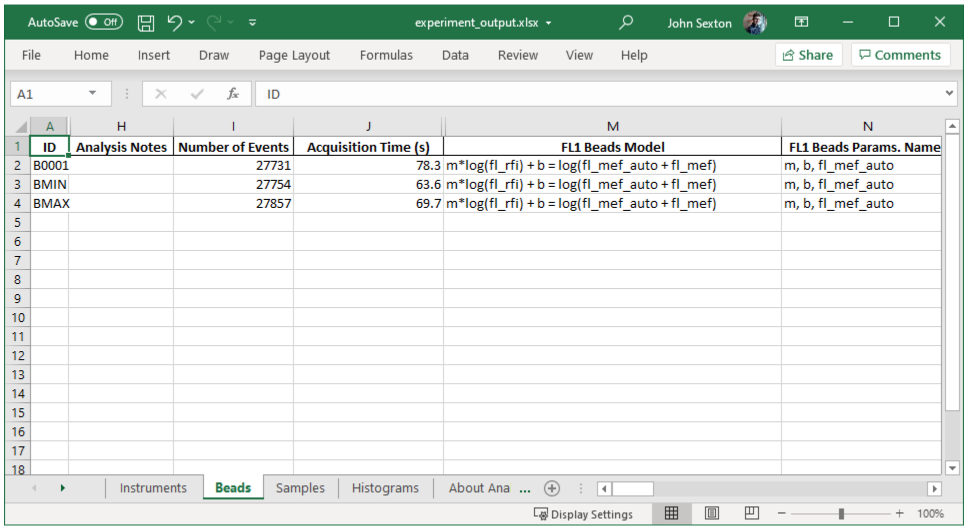


Output Excel File

The file `<Name of the input Excel file>_output.xlsx` contains calculated statistics for beads and samples. To produce this file, `FlowCal` copies the **Instruments**, **Beads**, and **Samples** sheets from the *input Excel* file, unmodified, to the output file, and adds columns to the **Beads** and **Samples** sheet with statistics.

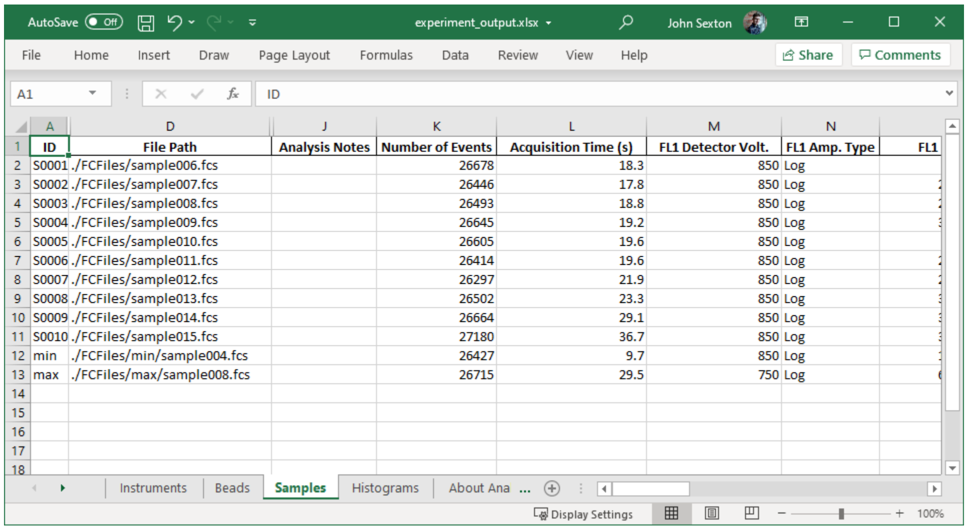
In both sheets, the number of events after gating and the acquisition time are reported for each sample. In addition, a column named **Analysis Notes** indicates the user about any errors that occurred during processing.

Statistics per beads file, per fluorescence channel include: the channel gain, the amplifier type, the equation of the beads fluorescence model used, and the values of the fitted parameters.



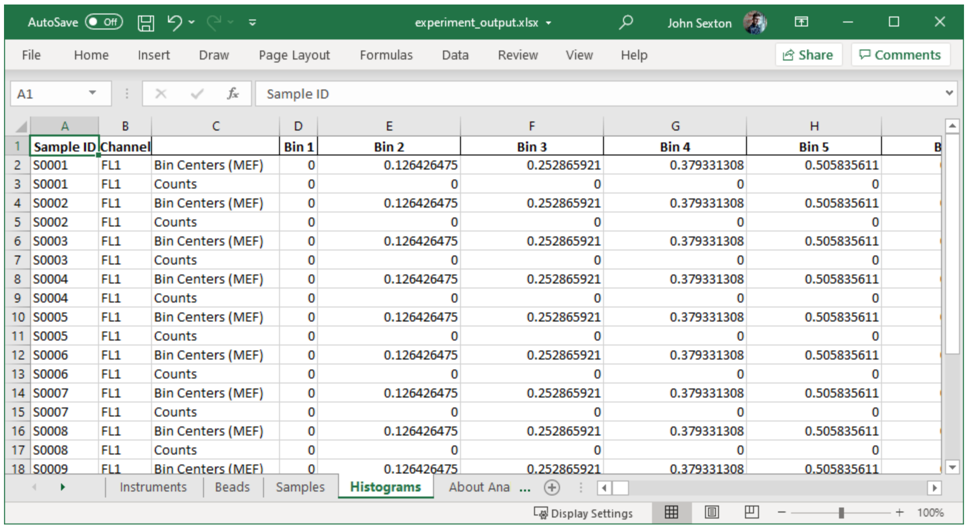
ID	Analysis Notes	Number of Events	Acquisition Time (s)	FL1 Beads Model	FL1 Beads Params. Name
B0001		27731	78.3	m*log(fl_rfi) + b = log(fl_mef_auto + fl_mef)	m, b, fl_mef_auto
BMIN		27754	63.6	m*log(fl_rfi) + b = log(fl_mef_auto + fl_mef)	m, b, fl_mef_auto
BMAX		27857	69.7	m*log(fl_rfi) + b = log(fl_mef_auto + fl_mef)	m, b, fl_mef_auto

Statistics per cell sample, per fluorescence channel include: channel gain, mean, geometric mean, median, mode, arithmetic and geometric standard deviation, arithmetic and geometric coefficient of variation (CV), interquartile range (IQR), and robust coefficient of variation (RCV). Note that if an error has been found, the **Analysis Notes** field will be populated, and statistics and plots will not be reported.



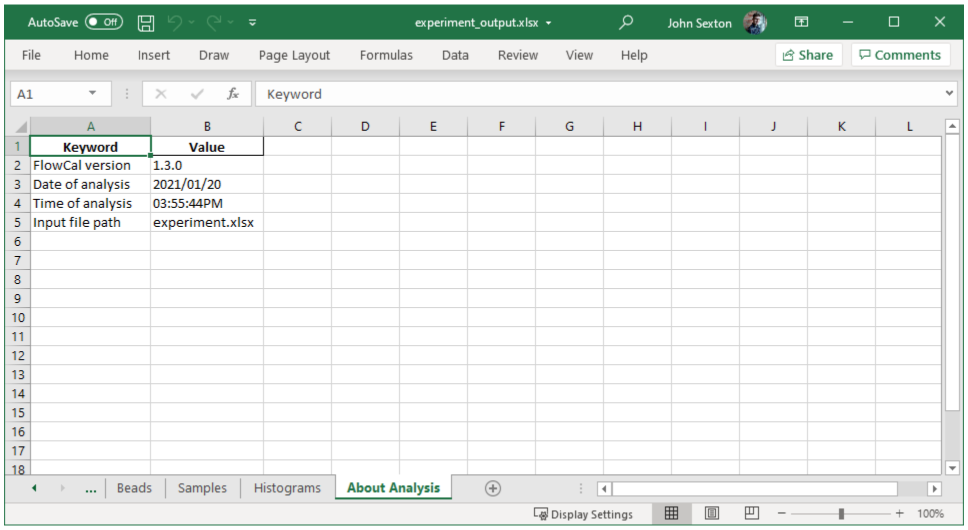
ID	File Path	Analysis Notes	Number of Events	Acquisition Time (s)	FL1 Detector Volt.	FL1 Amp. Type
S0001	./FCFiles/sample006.fcs		26678	18.3	850	Log
S0002	./FCFiles/sample007.fcs		26446	17.8	850	Log
S0003	./FCFiles/sample008.fcs		26493	18.8	850	Log
S0004	./FCFiles/sample009.fcs		26645	19.2	850	Log
S0005	./FCFiles/sample010.fcs		26605	19.6	850	Log
S0006	./FCFiles/sample011.fcs		26414	19.6	850	Log
S0007	./FCFiles/sample012.fcs		26297	21.9	850	Log
S0008	./FCFiles/sample013.fcs		26502	23.3	850	Log
S0009	./FCFiles/sample014.fcs		26664	29.1	850	Log
S0010	./FCFiles/sample015.fcs		27180	36.7	850	Log
min	./FCFiles/min/sample004.fcs		26427	9.7	850	Log
max	./FCFiles/max/sample008.fcs		26715	29.5	750	Log

In addition, a **Histograms** tab is generated, with bin/counts pairs for each sample and relevant fluorescence channel in the specified units.



Sample ID	Channel		Bin 1	Bin 2	Bin 3	Bin 4	Bin 5	B
S0001	FL1	Bin Centers (MEF)	0	0.126426475	0.252865921	0.379331308	0.505835611	
S0001	FL1	Counts	0	0	0	0	0	
S0002	FL1	Bin Centers (MEF)	0	0.126426475	0.252865921	0.379331308	0.505835611	
S0002	FL1	Counts	0	0	0	0	0	
S0003	FL1	Bin Centers (MEF)	0	0.126426475	0.252865921	0.379331308	0.505835611	
S0003	FL1	Counts	0	0	0	0	0	
S0004	FL1	Bin Centers (MEF)	0	0.126426475	0.252865921	0.379331308	0.505835611	
S0004	FL1	Counts	0	0	0	0	0	
S0005	FL1	Bin Centers (MEF)	0	0.126426475	0.252865921	0.379331308	0.505835611	
S0005	FL1	Counts	0	0	0	0	0	
S0006	FL1	Bin Centers (MEF)	0	0.126426475	0.252865921	0.379331308	0.505835611	
S0006	FL1	Counts	0	0	0	0	0	
S0007	FL1	Bin Centers (MEF)	0	0.126426475	0.252865921	0.379331308	0.505835611	
S0007	FL1	Counts	0	0	0	0	0	
S0008	FL1	Bin Centers (MEF)	0	0.126426475	0.252865921	0.379331308	0.505835611	
S0008	FL1	Counts	0	0	0	0	0	
S0009	FL1	Bin Centers (MEF)	0	0.126426475	0.252865921	0.379331308	0.505835611	

One last tab named **About Analysis** is added with information about the corresponding input Excel file, the date and time of the run, and the FlowCal version used.



Keyword	Value
FlowCal version	1.3.0
Date of analysis	2021/01/20
Time of analysis	03:55:44PM
Input file path	experiment.xlsx

2.3.5 Command Line Interface (Advanced)

The Excel UI can be run from a command line interpreter with one of the following equivalent statements:

```
flowcal [-h] [-i [INPUTPATH]] [-o [OUTPUTPATH]] [-v] [-p] [-H]
python -m FlowCal.excel_ui [-h] [-i [INPUTPATH]] [-o [OUTPUTPATH]] [-v] [-p] [-H]
```

Where the flags are:

```
-h, --help            show this help message and exit
-i [INPUTPATH], --inputpath [INPUTPATH]
                      input Excel file name. If not specified, show open
                      file window
-o [OUTPUTPATH], --outputpath [OUTPUTPATH]
                      output Excel file name. If not specified, use
                      [INPUTPATH]_output
```

(continues on next page)

(continued from previous page)

```
-v, --verbose          print information about individual processing steps
-p, --plot             generate and save density plots/histograms of beads
                       and samples
-H, --histogram-sheet
                       generate sheet in output Excel file specifying
                       histogram bins
```

Running `FlowCal`'s Excel UI without any flags will show the open file dialog to select an *input Excel file*. Once a file is selected, FlowCal will generate an *output Excel file*. In contrast to using `Run FlowCal (macOS)` or `Run FlowCal (Windows).bat`, the statement above with no flags will not display any messages during processing or generate any plots. To display messages and generate plots, use:

```
flowcal -v -p
```

`Run FlowCal (macOS)` and `Run FlowCal (Windows).bat` use, in fact, the following equivalent statement:

```
python -m FlowCal.excel_ui -v -p
```

Note: In macOS, a critical error may appear when trying to run the Excel UI from the command line. The error message is quite long, but one of the last lines reads similarly to this:

```
libc++abi.dylib: terminating with uncaught exception of type NSException
```

This is due to the `macosx` matplotlib backend conflicting with the `TkInter` library used to show the open file window. To solve this, you need to change matplotlib's backend to `TkAgg`. A few ways to do so can be found here.

We recommend changing the matplotlib's backend temporarily by setting the `MPLBACKEND` environment variable. If you follow this method, you should run the following before calling `FlowCal`:

```
export MPLBACKEND="TkAgg"
```

This is actually the solution implemented in `Run FlowCal (macOS)`.

Using the command line arguments, one can create a batch script to process several Excel files at once, each pointing to a different set of FCS files. Such script would have the form:

```
flowcal -i input_excel_file_1.xlsx -o output_excel_file_1.xlsx
flowcal -i input_excel_file_2.xlsx -o output_excel_file_2.xlsx
flowcal -i input_excel_file_3.xlsx -o output_excel_file_3.xlsx
...
```

2.4 FlowCal's Python API Tutorial

`FlowCal` is, at its core, a Python library that a programmer can use to analyze flow cytometry data in a more flexible way than with the *Excel UI*. This section gives an overview of the abilities of `FlowCal` from a programmer's perspective. The tutorials below are listed below in order of increasing complexity. We recommend the reader to go through them in order.

Note: The `FlowCal` Python API tutorial assumes that the reader is familiar with Python, `numpy` and `matplotlib`. It also assumes that the reader has Python 3.8 installed, as well as `FlowCal` and all its dependencies. For more information on installation, refer to the *Getting started* section.

2.4.1 Reading Flow Cytometry Data

This tutorial focuses on how to open FCS files and manipulate the data therein using `FlowCal`.

To start, navigate to the `examples` directory included with FlowCal, and open a `python` session therein. Then, import `FlowCal` as with any other python module.

```
>>> import FlowCal
```

FCS files are standard files in which flow cytometry data is stored. Normally, one FCS file corresponds to one sample.

The object *`FlowCal.io.FCSData`* allows a user to open an FCS file. The following instruction opens the file `sample006.fcs` from the `FCFiles` folder, loads the information into an `FCSData` object, and assigns it to a variable `s`.

```
>>> s = FlowCal.io.FCSData('FCFiles/sample006.fcs')
```

An `FCSData` object is a 2D `numpy` array with a few additional features. The first dimension indexes the event number, and the second dimension indexes the flow cytometry channel (or "parameter", as called by the FCS standard). We can see the number of events and channels using the standard `numpy`'s `shape` property:

```
>>> print(s.shape)
(32224, 8)
```

As with any `numpy` array, we can slice an `FCSData` object. For example, let's obtain the first 100 events.

```
>>> s_sub = s[:100]
>>> print(s_sub.shape)
(100, 8)
```

Note that the product of slicing an FCSData object is also an FCSData object. We can also get all the events in a subset of channels by slicing in the second dimension.

```
>>> s_sub_ch = s[:, [3, 4, 5]]
>>> print(s_sub_ch.shape)
(32224, 3)
```

However, it is not immediately obvious what channels we are getting. Fortunately, the `FCSData` object contains some additional information about the acquisition settings. In particular, we can check the name of the channels with the `channels` property.

```
>>> print(s.channels)
('TIME', 'FSC', 'SSC', 'FL1', 'FL2', 'FL3', 'FSCW', 'FSCA')
>>> print(s_sub_ch.channels)
('FL1', 'FL2', 'FL3')
```

It turns out that `s_sub_ch` contains the fluorescence channels `FL1`, `FL2`, and `FL3`.

One of the most practical features of an `FCSData` object is the ability to slice channels using their name. For example, if we want the fluorescence channels we can use the following.

```
>>> s_sub_ch_2 = s[:, ['FL1', 'FL2', 'FL3']]
>>> print(s_sub_ch_2.channels)
('FL1', 'FL2', 'FL3')
```

This is completely equivalent to indexing with integers.

```
>>> import numpy as np
>>> np.all(s_sub_ch == s_sub_ch_2)
True
```

`FCSData` contains more acquisition information, such as the acquisition time, amplifier type, and the detector voltage of each channel. For more information, consult the documentation of *`FlowCal.io.FCSData`*.

2.4.2 Transforming Flow Cytometry Data

This tutorial focuses on how to perform basic transformations to flow cytometry data using `FlowCal`, particularly by using the module *`FlowCal.transform`*

To start, navigate to the `examples` directory included with FlowCal, and open a `python` session therein. Then, import `FlowCal` as with any other python module.

```
>>> import FlowCal
```

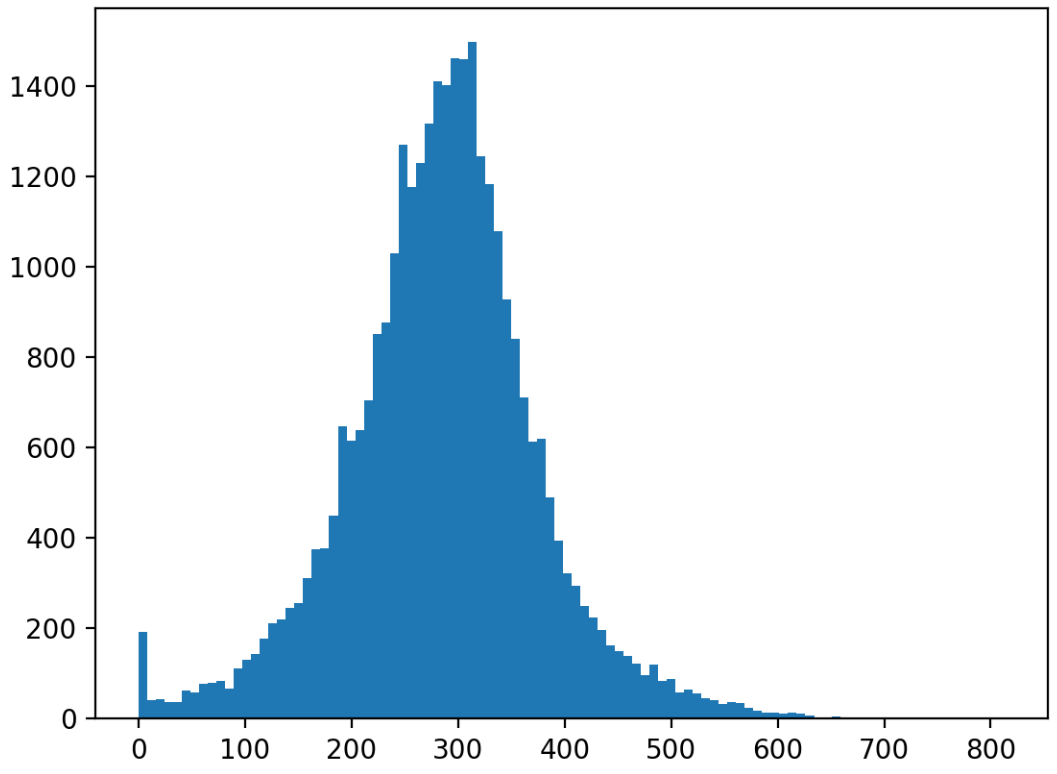
Transforming to Arbitrary Fluorescence Units (a.u.)

Start by loading file `sample006.fcs` into an `FCSData` object called `s`.

```
>>> s = FlowCal.io.FCSData('FCFiles/sample006.fcs')
```

Let's now visualize the contents of the `FL1` channel. We will explore `FlowCal`'s plotting functions in the *plotting tutorial*, but for now let's just use `matplotlib`'s `hist` function.

```
>>> import matplotlib.pyplot as plt
>>> plt.hist(s[:, 'FL1'], bins=100)
>>> plt.show()
```



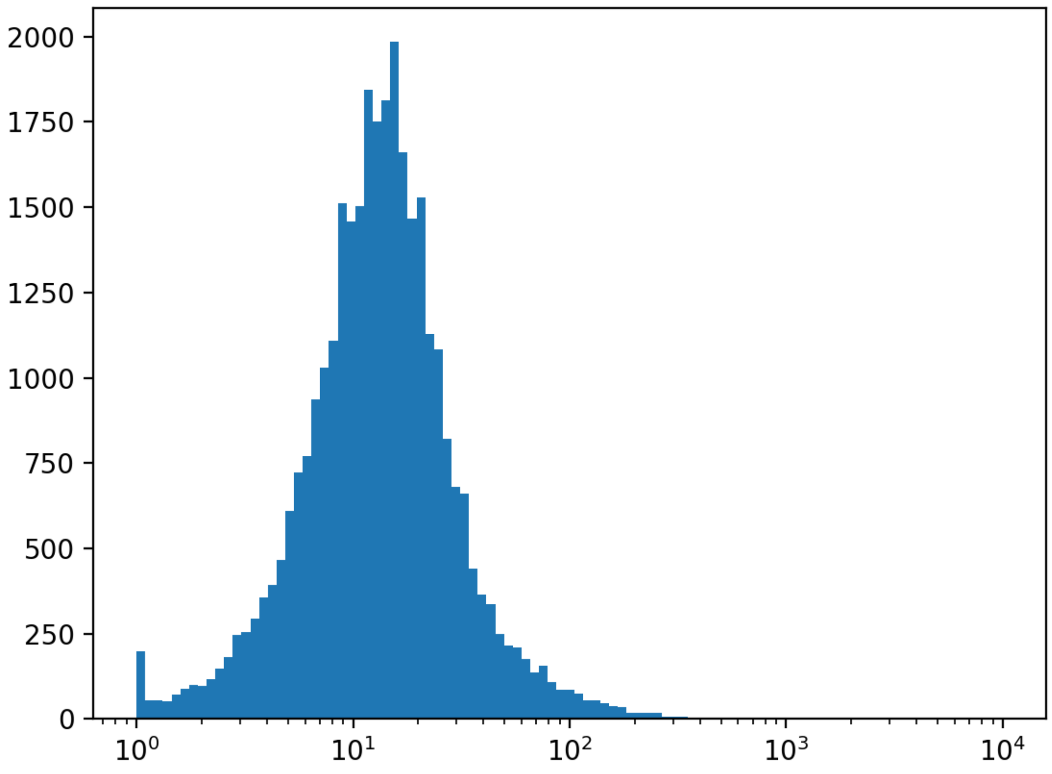
Note that the range of the x axis is from 0 to around 800. However, our acquisition software showed fluorescence values from 1 to 10000. Where does the difference come from? An FCS file normally stores raw numbers as they are are reported by the instrument sensors. These are referred to as "channel numbers". The FCS file also contains enough information to transform these numbers back to proper fluorescence units, called Relative Fluorescence Intensities (RFI), or more commonly, arbitrary fluorescence units (a.u.). Depending on the instrument used, this conversion sometimes involves a simple scaling factor, but other times requires a non-straigthforward exponential transformation. The latter is our case.

Fortunately, `FlowCal` includes *`FlowCal.transform.to_rfi()`*, a function that reads all the necessary paremeters from the FCS file and figures out how to convert data back to a.u.

```
>>> s_transformed = FlowCal.transform.to_rfi(s, channels='FL1')
```

`s_transformed` now contains the same data as `s`, except that the `FL1` channel has been transformed to a.u. Let's now look at the transformed data.

```
>>> import numpy as np
>>> bins = np.logspace(0, 4, 100)
>>> plt.hist(s_transformed[:, 'FL1'], bins=bins)
>>> plt.xscale('log')
>>> plt.show()
```



We will explore a more convenient way to plot transformed data in the *plotting tutorial*.

`FlowCal.transform.to_rfi()` can transform several channels at the same time. In fact, all channels will be transformed if no channel is specified.

```
>>> s_transformed = FlowCal.transform.to_rfi(s)
```

We will use this throughout the whole tutorial right after loading an FCSData object.

Transforming to Molecules of Equivalent Fluorophore (MEF)

`FlowCal` includes the ability to transform flow cytometry data to *Molecules of Equivalent Fluorophore (MEF)*, a unit independent of the acquisition settings. However, doing so is slightly more complex. We will see how to do this in the *MEF tutorial*.

2.4.3 Plotting Flow Cytometry Data

This tutorial focuses on how to plot flow cytometry data using `FlowCal`, particularly by using the module *`FlowCal.plot`*

To start, navigate to the `examples` directory included with FlowCal, and open a `python` session therein. Then, import `FlowCal` as with any other python module.

```
>>> import FlowCal
```

Also, import `numpy` and `pyplot` from `matplotlib`

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
```

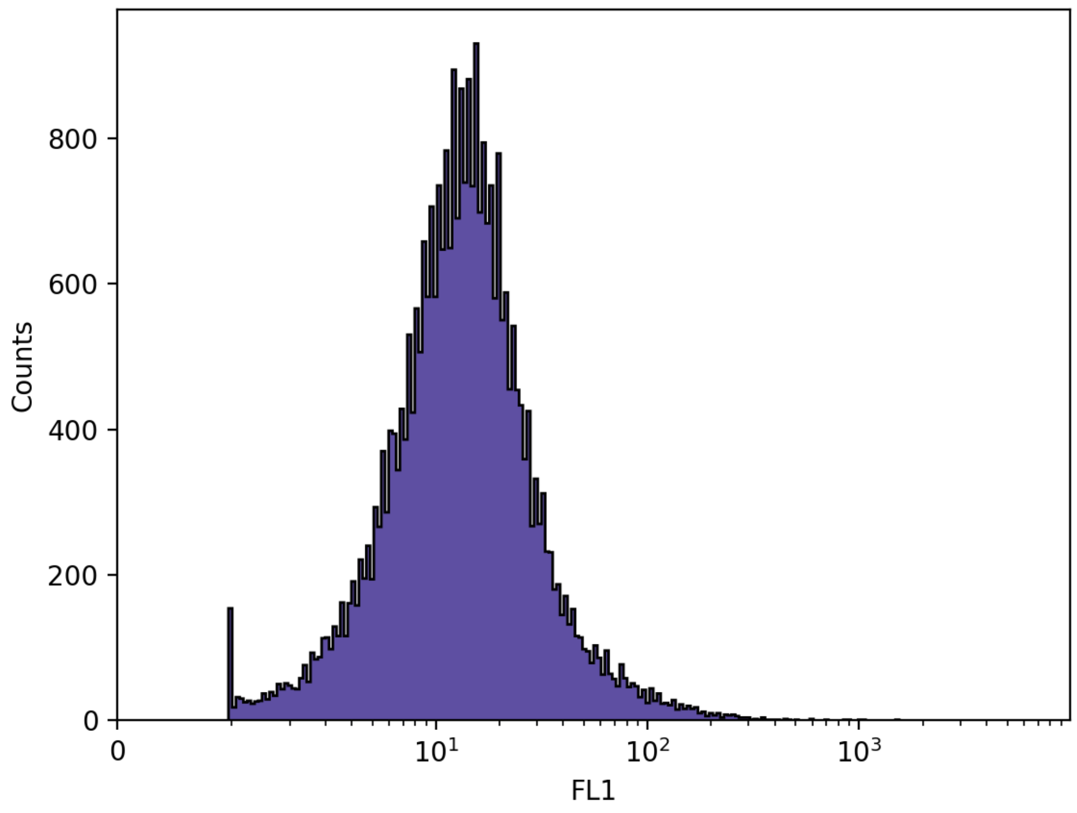
Histograms

Let's load the data from file `sample006.fcs` into an `FCSData` object called `s`, and tranform all channels to arbitrary units.

```
>>> s = FlowCal.io.FCSData('FCFiles/sample006.fcs')
>>> s = FlowCal.transform.to_rfi(s)
```

One is often interested in the fluorescence distribution across a population of cells. This is represented in a histogram. Since `FCSData` is a numpy array, one could use the standard `hist` function included in matplotlib. Alternatively, `FlowCal` includes its own histogram function specifically tailored to work with `FCSData` objects. For example, one can plot the contents of the `FL1` channel with a single call to *`FlowCal.plot.hist1d()`*.

```
>>> FlowCal.plot.hist1d(s, channel='FL1')
>>> plt.show()
```

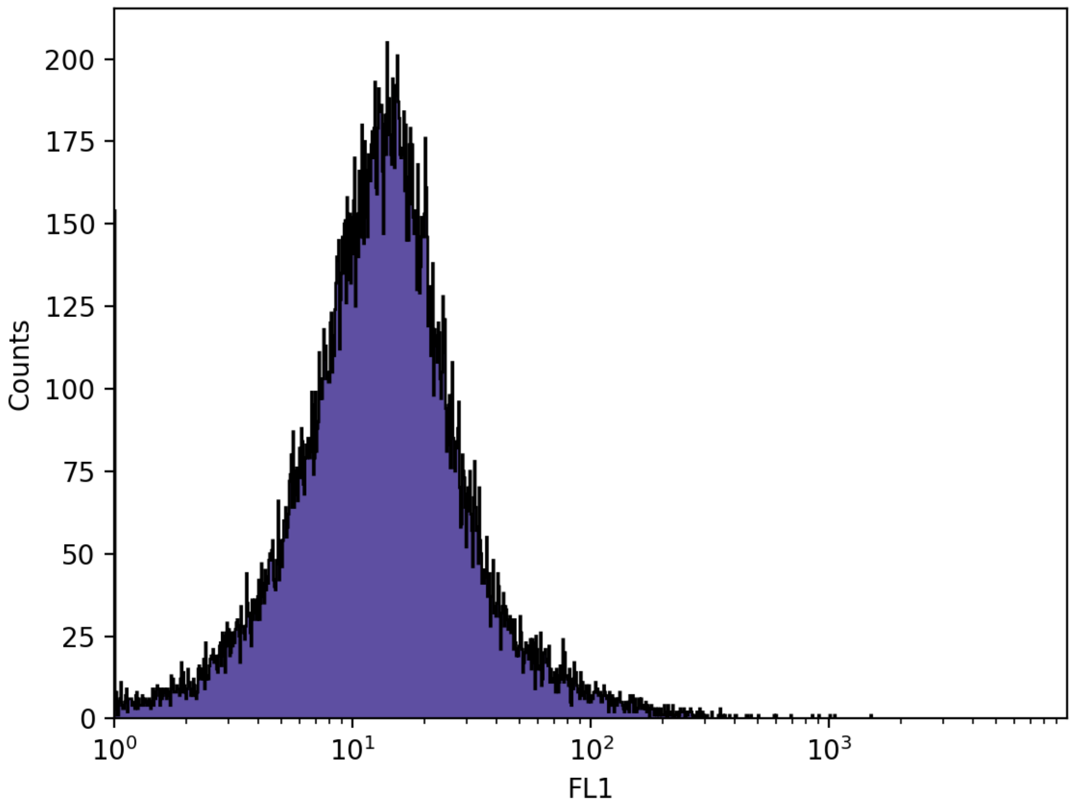


`FlowCal.plot.hist1d()` behaves mostly like a regular matplotlib plotting function: it will plot in the current

figure and axis. The axes labels are populated by default, but one can still use `plt.xlabel` and `plt.ylabel` to change them.

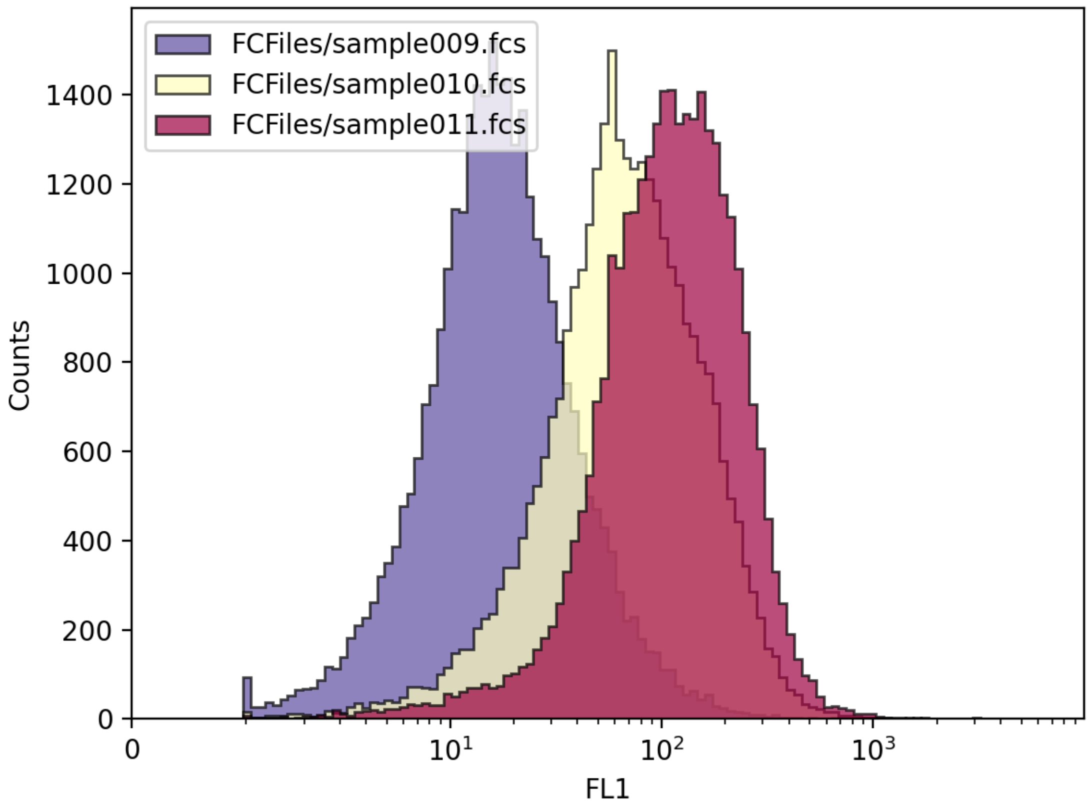
By default, *`FlowCal.plot.hist1d()`* uses something called *logicle* scaling for the x axis. This scaling allows visualization of high fluorescence values with logarithmic spacing, and low fluorescence values with a more linear spacing. In some modern flow cytometers, negative events may be present, and logicle scaling allows visualization of those as well. This can be changed to a more conventional linear or logarithmic scale by using the `xscale` argument. In addition, *`FlowCal.plot.hist1d()`* uses 256 uniformly spaced bins by default. We can override the default bins using the `bins` argument. Let's try using 1024 logarithmically-spaced bins.

```
>>> FlowCal.plot.hist1d(s, channel='FL1', xscale='log', bins=1024)
>>> plt.show()
```



Finally, *`FlowCal.plot.hist1d()`* can plot several FCSData objects at the same time. Let's now load 3 FCSData objects, transform all channels to a.u., and plot the `FL1` channel of all three with transparency.

```
>>> filenames = ['FCFiles/sample{:03d}.fcs'.format(i + 9) for i in range(3)]
>>> d = [FlowCal.io.FCSData(filename) for filename in filenames]
>>> d = [FlowCal.transform.to_rfi(di) for di in d]
>>> FlowCal.plot.hist1d(d, channel='FL1', alpha=0.7, bins=128)
>>> plt.legend(filenames, loc='upper left')
>>> plt.show()
```

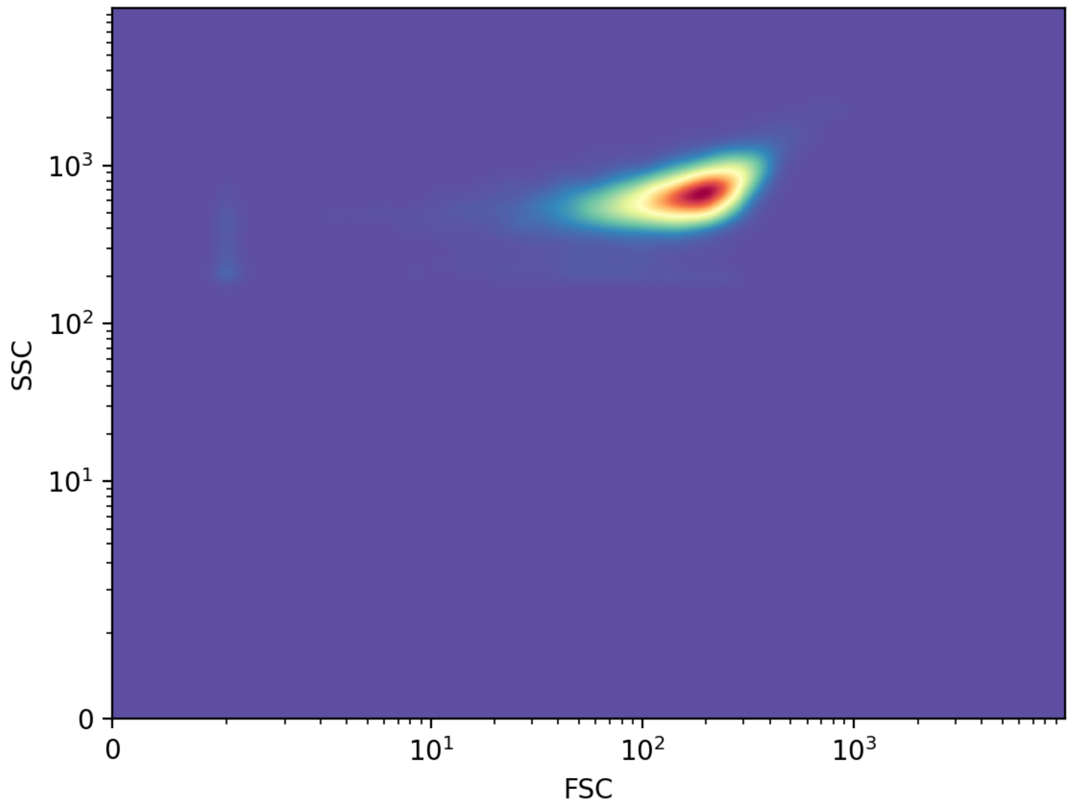


Density Plots

It is also common to look at the forward scatter and side scatter values in a 2D histogram, scatter plot, or density diagram. From those, the user can extract size and shape information that would allow him to differentiate between cells and debris. `FlowCal` includes the function *`FlowCal.plot.density2d()`* for this purpose.

Let's look at the `FSC` and `SSC` channels in our sample `s`.

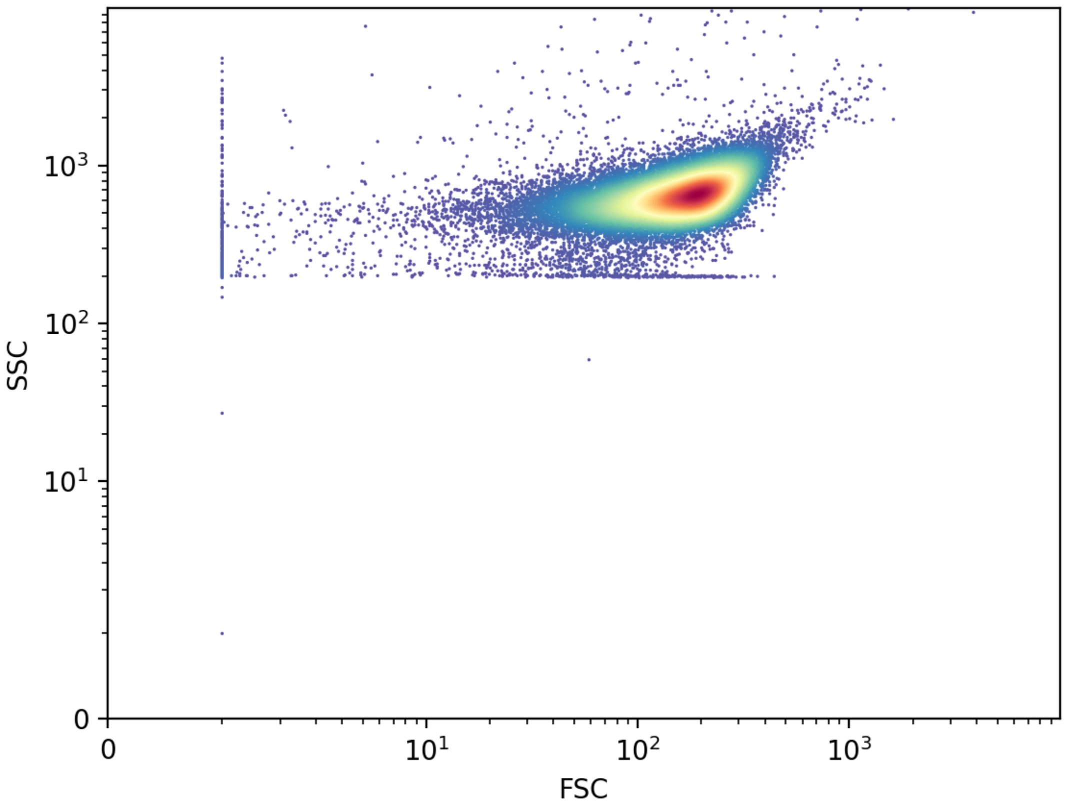
```
>>> FlowCal.plot.density2d(s, channels=['FSC', 'SSC'])
>>> plt.show()
```



The color indicates the number of events in the region, with red indicating a bigger number than yellow and blue, in that order, by default. Similarly to *FlowCal.plot.hist1d()*, *FlowCal.plot.density2d()* uses logicle scaling by default. In addition, *FlowCal.plot.density2d()* applies, by default, gaussian smoothing to the density plot.

FlowCal.plot.density2d() includes two visualization modes: `mesh` (seen above), and `scatter`. The last one is good for distinguishing regions with few events.

```
>>> FlowCal.plot.density2d(s, channels=['FSC', 'SSC'], mode='scatter')
>>> plt.show()
```



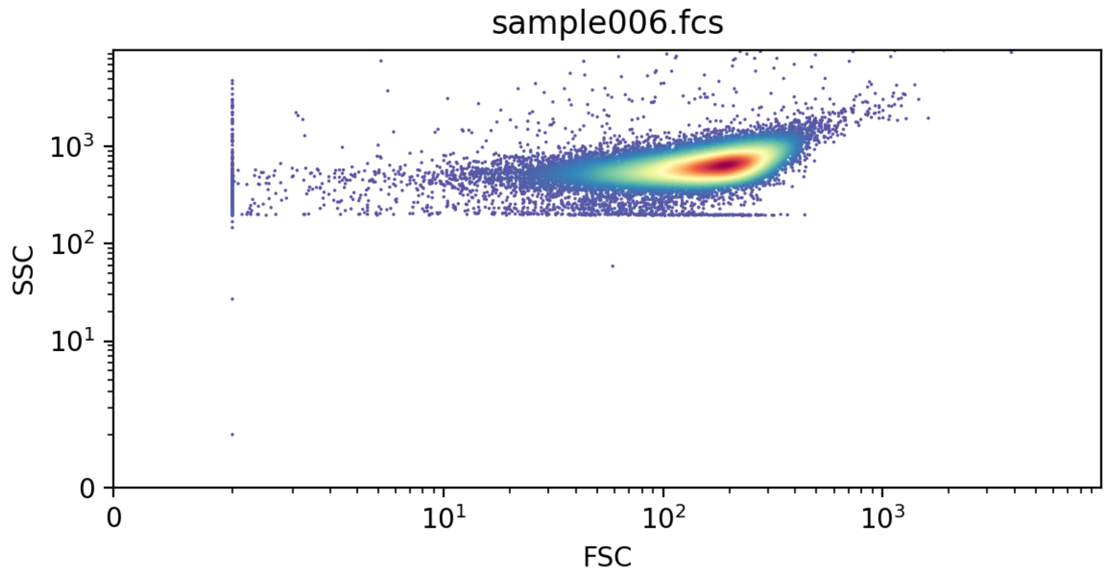
The last plot shows three distinct populations. The large one in the middle corresponds to cells, whereas the ones at the left and below correspond to non-biological debris. We will see how to “gate”, or select only one population, in the *gating tutorial*.

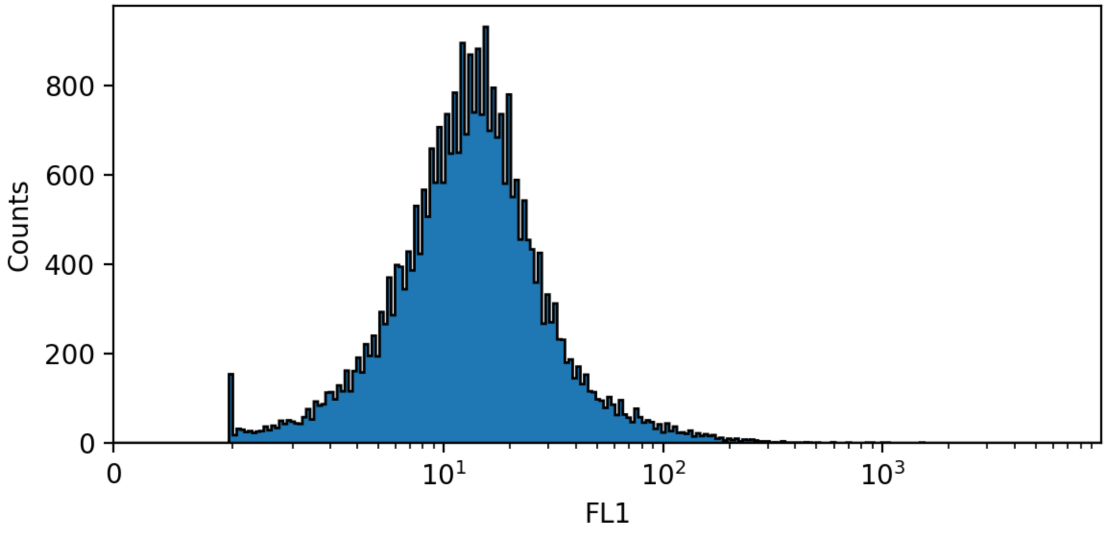
Combined Histogram and Density Plots

FlowCal also includes “complex plot” functions, which produce their own figure and a set of axes, and use simple `matplotlib` or `FlowCal` plotting functions to populate them.

In particular, *`FlowCal.plot.density_and_hist()`* uses *`FlowCal.plot.hist1d()`* and *`FlowCal.plot.density2d()`* to produce a combined density plot/histogram that allow the user to quickly see information about one sample. For example, let’s plot the `FSC` and `SSC` channels in a density plot, and the `FL1` channel in a histogram. In the following, `density_params` and `hist_params` are dictionaries that are directly passed to *`FlowCal.plot.hist1d()`* and *`FlowCal.plot.density2d()`* as keyword arguments.

```
>>> FlowCal.plot.density_and_hist(s,
...                               density_channels=['FSC', 'SSC'],
...                               density_params={'mode':'scatter'},
...                               hist_channels=['FL1'])
>>> plt.tight_layout()
>>> plt.show()
```





FlowCal.plot.density_and_hist() can also plot data before and after applying gates. We will see this in the *gating tutorial*.

Violin Plots

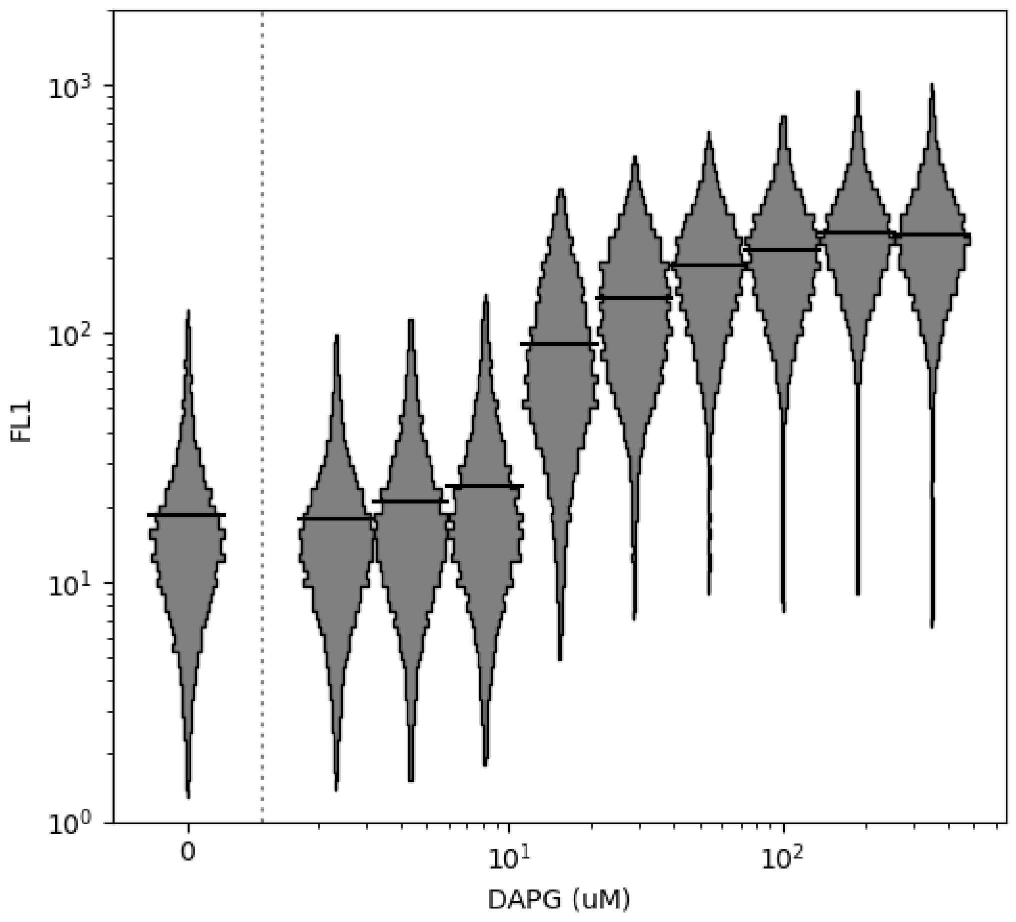
Histograms, as shown above, can be used to plot and compare data from multiple samples. However, they can easily get too crowded. A more compact way is to use a violin plot, wherein vertical, normalized, symmetrical histograms ("violins") are shown centered on corresponding x-axis values. We can do this with the *FlowCal.plot.violin()* function.

```
>>> filenames = ['FCFiles/sample{:03d}.fcs'.format(i+6) for i in range(10)]
>>> d = [FlowCal.io.FCSData(filename) for filename in filenames]
>>> d = [FlowCal.transform.to_rfi(di) for di in d]
>>> dapg = np.array([0, 2.33, 4.36, 8.16, 15.3, 28.6, 53.5, 100, 187, 350])
>>> FlowCal.plot.violin(data=d, channel='FL1', positions=dapg, xlabel='DAPG (uM)',␣
↪xscale='log', ylim=(1e0,2e3))
```

(continues on next page)

(continued from previous page)

```
>>> plt.show()
```



Note that the x axis has been plotted on a logarithmic scale using the `xscale` argument. Because data at position x=0 is specified, *FlowCal.plot.violin()* places it separately on the left side of the plot. In contrast, the y-axis is plotted on a `logicle` scale by default. However, it can be switched to `log` or `linear` using the argument `yscale`. Horizontal violin plots can also be generated by setting the `vert` argument to `False`. For more options, consult the function documentation.

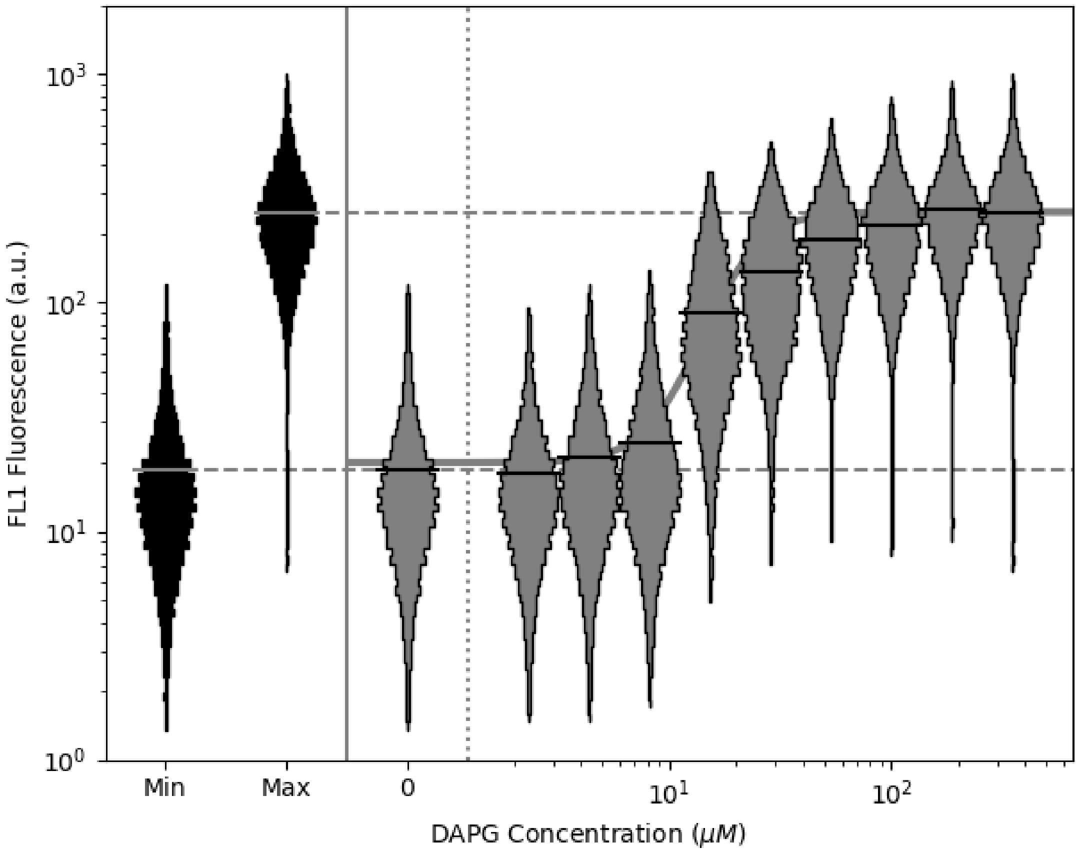
"Dose response" or "transfer" functions are common in biology. These sometimes include minimum (negative) and maximum (positive) controls, and are often approximated by mathematical models. The *FlowCal.plot.violin_dose_response()* function can be used to plot a full dose response dataset, including min data, max data, and a mathematical model. Min and max data are illustrated to the left of the plot, and the mathematical model is correctly illustrated even when a position=0 violin is illustrated separately when `xscale` is `log`.

```
>>> # Function specifying mathematical model
>>> def dapg_sensor_model(dapg_concentration):
>>>     mn = 20
>>>     mx = 250.
>>>     K  = 20.
```

(continues on next page)

(continued from previous page)

```
>>>     n  = 3.57
>>>     if dapg_concentration <= 0:
>>>         return mn
>>>     else:
>>>         return mn + ((mx-mn)/(1+((K/dapg_concentration)**n)))
>>>
>>> # Plot
>>> FlowCal.plot.violin_dose_response(
>>>     data=d,
>>>     channel='FL1',
>>>     positions=dapg,
>>>     min_data=d[0],
>>>     max_data=d[-1],
>>>     model_fxn=dapg_sensor_model,
>>>     xscale='log',
>>>     yscale='log',
>>>     ylim=(1e0,2e3),
>>>     draw_model_kwargs={'color':'gray',
>>>                        'linewidth':3,
>>>                        'zorder':-1,
>>>                        'solid_capstyle':'butt'})
>>> plt.xlabel('DAPG Concentration ($\mu M$)')
>>> plt.ylabel('FL1 Fluorescence (a.u.)')
>>> plt.show()
```



Other Plotting Functions

These are not the only functions in *`FlowCal.plot`*. For more information, consult the API reference.

2.4.4 Gating Flow Cytometry Data

This tutorial focuses on how to gate flow cytometry data using `FlowCal`, particularly by using the module *`FlowCal.gate`*. Gating is the process of retaining events that satisfy some criteria, and discarding the ones that do not.

To start, navigate to the `examples` directory included with FlowCal, and open a `python` session therein. Then, import `FlowCal` as with any other python module.

```
>>> import FlowCal
```

Also, import `numpy` and `pyplot` from `matplotlib`

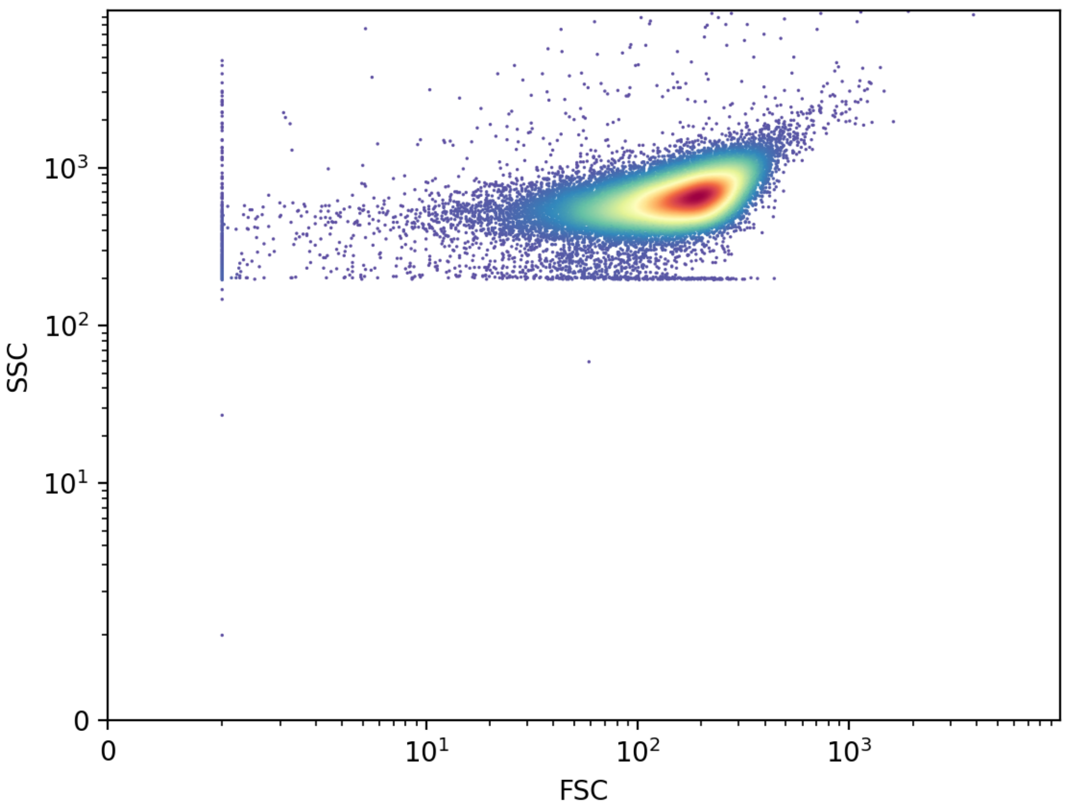
```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
```

Removing Saturated Events

We'll start by loading the data from file `sample006.fcs` into an `FCSData` object called `s`. Then, transform all channels into a.u.

```
>>> s = FlowCal.io.FCSData('FCFiles/sample006.fcs')
>>> s = FlowCal.transform.to_rfi(s)
```

In the *plotting tutorial* we looked at a density plot of the forward scatter/side scatter (`FSC`/`SSC`) channels and identified several clusters of particles (events). This density plot is repeated below for convenience.



From these subpopulations, the faint elongated one in the low-middle portion corresponds to non-cellular debris, and the large one in the middle corresponds to cells. One additional elongated subpopulation on the left corresponds to saturated events, with the lowest possible forward scatter value: 1 a.u..

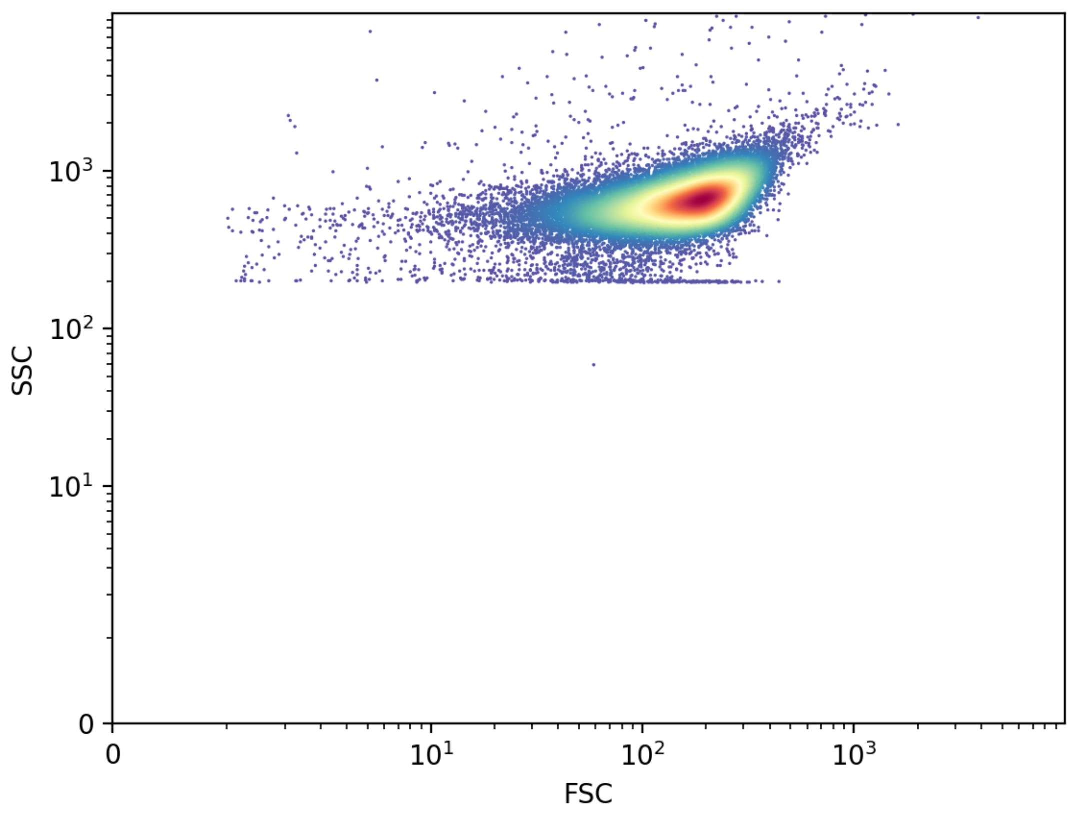
Some flow cytometers will capture events outside of their range and assign them either the lowest or highest possible values of a channel, depending on which side of the range they fall on. We call these events "saturated". Including them in the analysis results, most of the time, in distorted distribution shapes and incorrect statistics. Therefore, it is generally advised to remove saturated events. To do so, `FlowCal` incorporates the function *`FlowCal.gate.high_low()`*. This function retains all the events in the specified channels between two specified values: a high and a low threshold. If these values are not specified, however, the function uses the saturating values.

```
>>> s_g1 = FlowCal.gate.high_low(s, channels=['FSC', 'SSC'])
>>> FlowCal.plot.density2d(s_g1,
...                         channels=['FSC', 'SSC'],
```

(continues on next page)

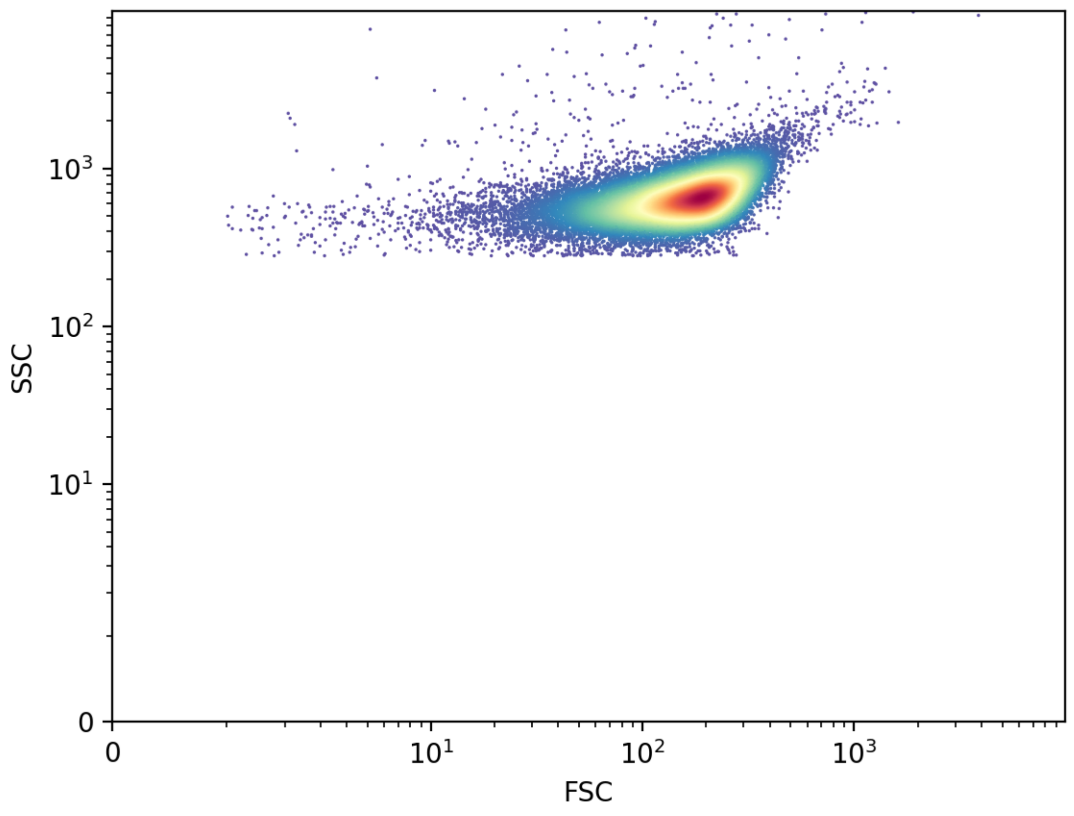
(continued from previous page)

```
...                             mode='scatter')
>>> plt.show()
```



We successfully removed the events on the left. We can go one step further and use *FlowCal.gate.high_low()* again to remove some of the events below the main event cluster, which as we said before corresponds to debris.

```
>>> s_g2 = FlowCal.gate.high_low(s_g1, channels='SSC', low=280)
>>> FlowCal.plot.density2d(s_g2,
...                             channels=['FSC', 'SSC'],
...                             mode='scatter')
>>> plt.show()
```

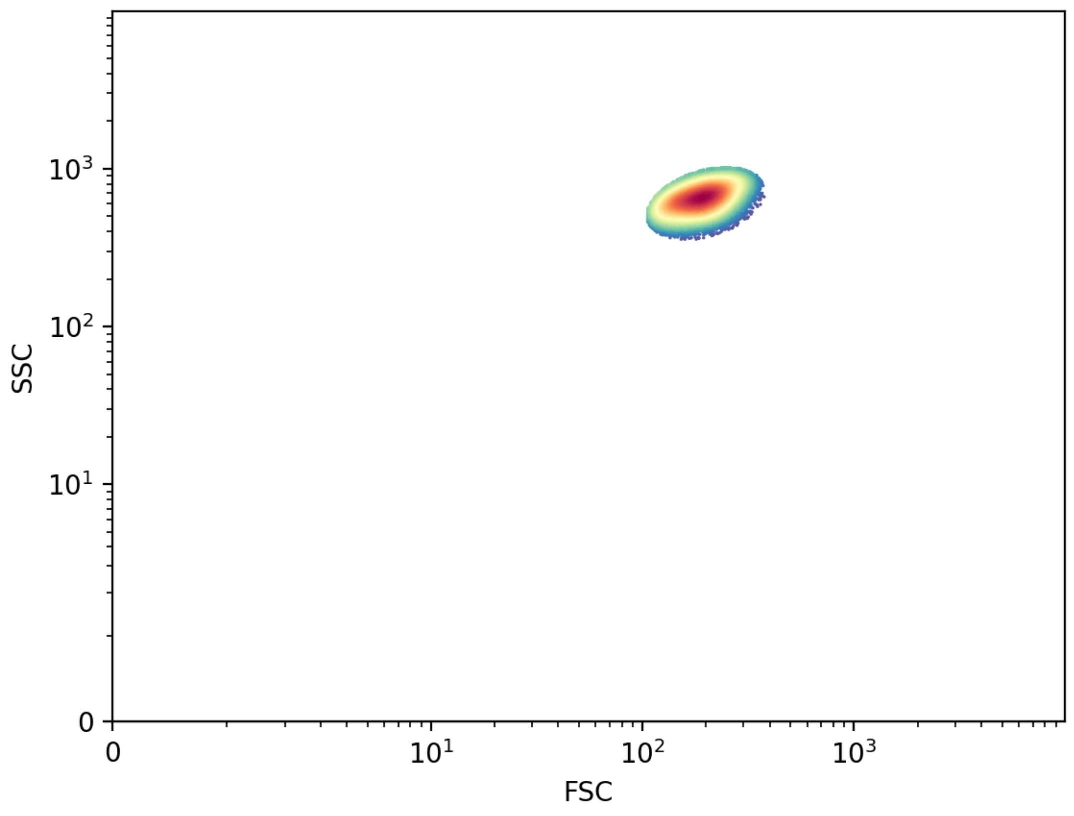


This approach, however, requires one to estimate a low threshold value for every sample manually. In addition, we usually want events in the densest forward scatter/side scatter region, which requires a more complex shape than a pair of thresholds. We will now explore better ways to achieve this.

Ellipse Gate

FlowCal includes an ellipse-shaped gate, in which events are retained if they fall inside an ellipse with a specified center and dimensions. Let's try to obtain the densest region of the cell cluster.

```
>>> s_g3 = FlowCal.gate.ellipse(s_g1,
...                             channels=['FSC', 'SSC'],
...                             log=True,
...                             center=(2.3, 2.78),
...                             a=0.3,
...                             b=0.2,
...                             theta=30/180.*np.pi)
>>> FlowCal.plot.density2d(s_g3,
...                        channels=['FSC', 'SSC'],
...                        mode='scatter')
>>> plt.show()
```



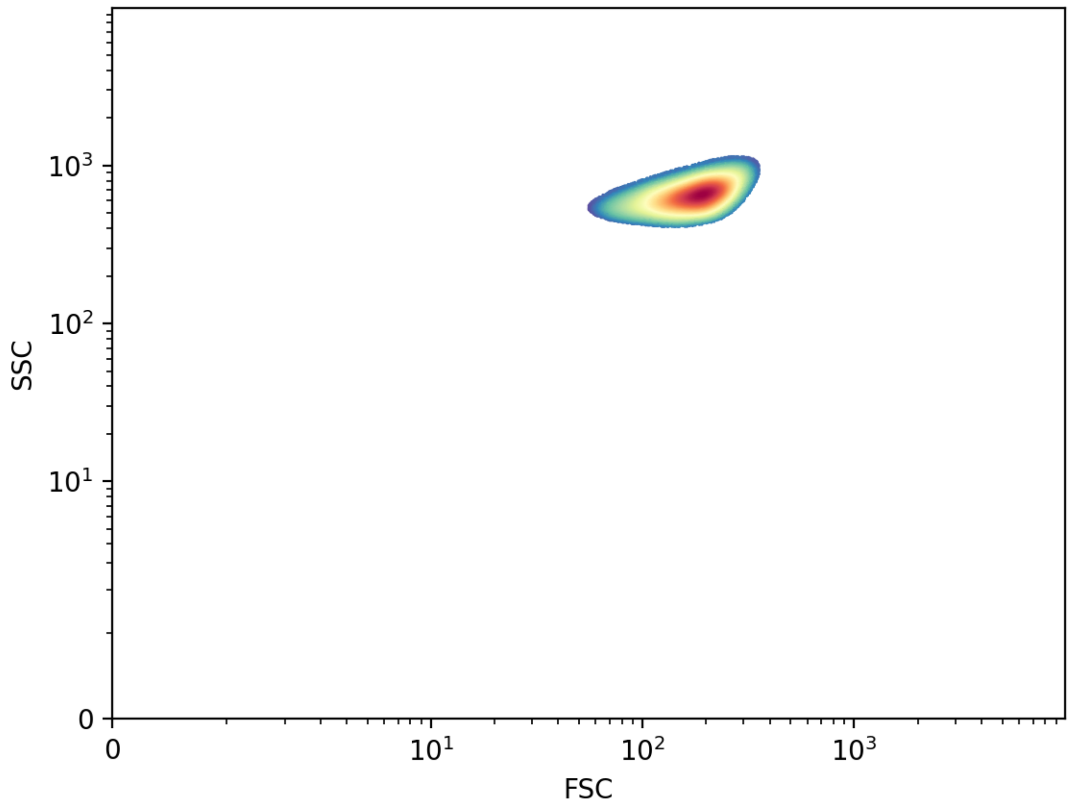
As shown above, the remaining events reside only inside an ellipse-shaped region. Note that we used the argument `log`, which indicates that the gated region should look like an ellipse in a logarithmic plot. This also requires that the center and the major and minor axes (`a` and `b`) be specified in log space.

The disadvantage of this gate is that several parameters need to be specified, which make the resulting gate arbitrary. In addition, it is questionable whether we're actually capturing the densest part of the distribution. Using the mean or median as centers results in similar issues because the original cell distribution is not symmetrical. The next gate solves these issues.

Density Gate

`FlowCal.gate.density2d()` automatically identifies the region with the highest density of events in a two-dimensional diagram, and calculates how big it should be to capture a certain percentage of the total event count. One advantage is that the number of user-defined parameters is reduced to one. Let's now try to separate cells from debris using this method.

```
>>> s_g4 = FlowCal.gate.density2d(s_g1,
...                               channels=['FSC', 'SSC'],
...                               gate_fraction=0.75)
>>> FlowCal.plot.density2d(s_g4,
...                        channels=['FSC', 'SSC'],
...                        mode='scatter')
>>> plt.show()
```



We can see that *FlowCal.gate.density2d()* automatically identified the region that contains cells, and defined a shape that more closely resembles what the ungated density map looks like. The parameter `gating_fraction` allows the user to control the fraction of events to retain, and it is the only parameter that the user is required to specify.

For more details on how *FlowCal.gate.density2d()* works, consult the section on *fundamentals of density gating*.

Plotting 2D Gates

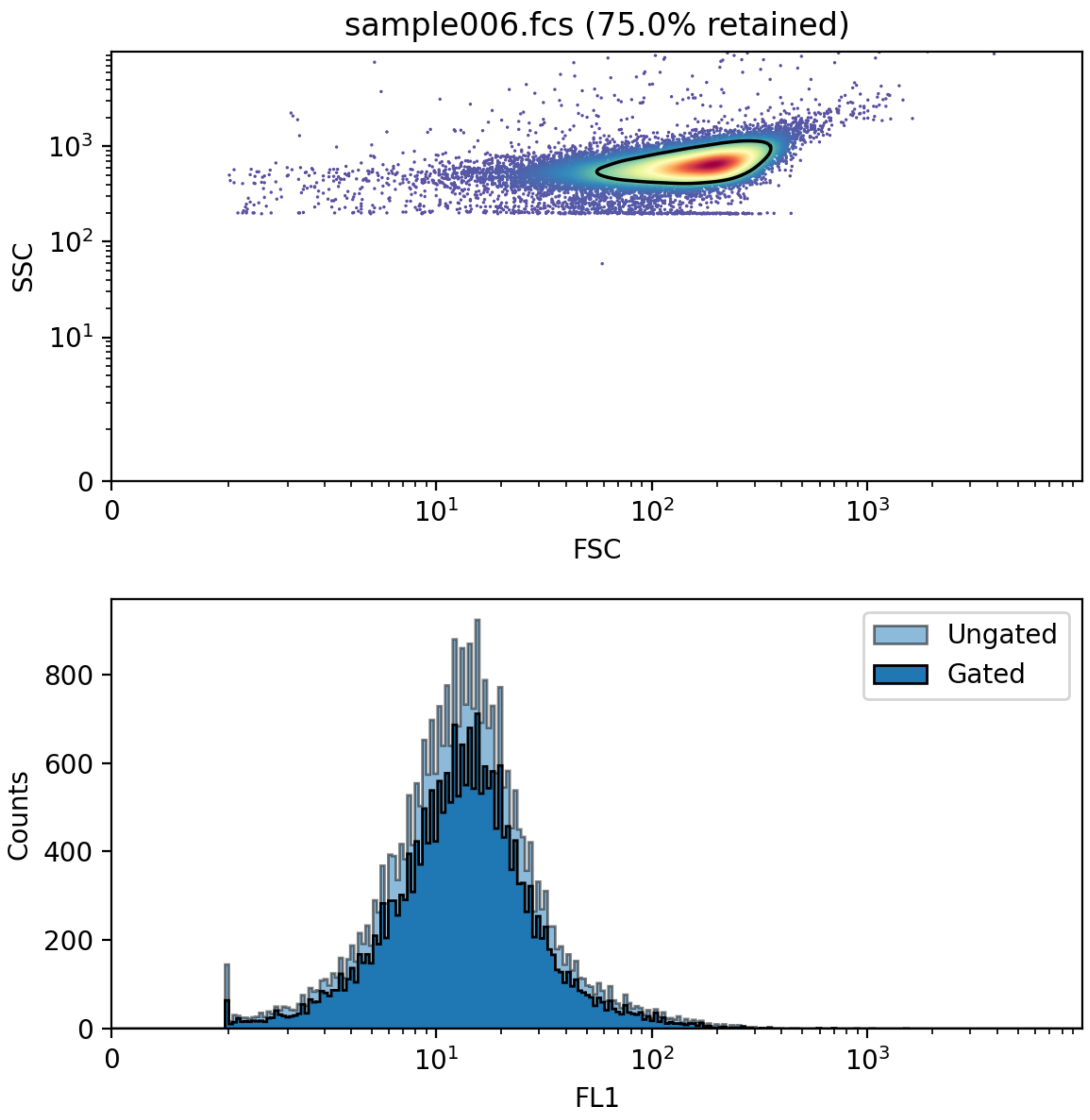
Finally, we will see a better way to visualize the result of applying a 2D gate. First, we will use density gating again, but this time we will do it a little differently.

```
>>> density_gate_output = FlowCal.gate.density2d(s_g1,
...                                               channels=['FSC', 'SSC'],
...                                               gate_fraction=0.75,
...                                               full_output=True)
>>> s_g5    = density_gate_output.gated_data
>>> m_g5    = density_gate_output.mask
>>> contour = density_gate_output.contour
```

The extra argument, `full_output`, is available in every function in *FlowCal.gate*. It instructs a gating function to return additional output arguments with information about the gating process. The second output argument is always a mask (extracted here from the `Density2dGateOutput namedtuple` using its field name), which is a boolean array that indicates which events from the original FCSData object are being retained by the gate. Two-dimensional gating functions have a third output argument: a contour surrounding the gated region, which we will now use for plotting.

The function `FlowCal.plot.density_and_hist()` was introduced in the *plotting tutorial* to produce plots of a single FCSData object. But it can also be used to plot the result of a gating step, showing the data before and after gating, and the gating contour. Let's use this ability to show the result of the density gating process.

```
>>> FlowCal.plot.density_and_hist(s_g1,
...                               gated_data=s_g5,
...                               gate_contour=contour,
...                               density_channels=['FSC', 'SSC'],
...                               density_params={'mode':'scatter'},
...                               hist_channels=['FL1'])
>>> plt.tight_layout()
>>> plt.show()
```



We can now observe the gating contour right on top of the ungated data, and see which events were kept and which ones were left out. In addition, we can visualize how gating affected the other channels.

2.4.5 Calibrating Flow Cytometry Data to MEF

This tutorial focuses on how to transform flow cytometry data to Molecules of Equivalent Fluorophore (MEF) using `FlowCal`, particularly by using the module *`FlowCal.mef`*. For more information on MEF calibration, see the section on *fundamentals of calibration*.

To start, navigate to the `examples` directory included with FlowCal, and open a `python` session therein. Then, import `FlowCal` as with any other python module.

```
>>> import FlowCal
```

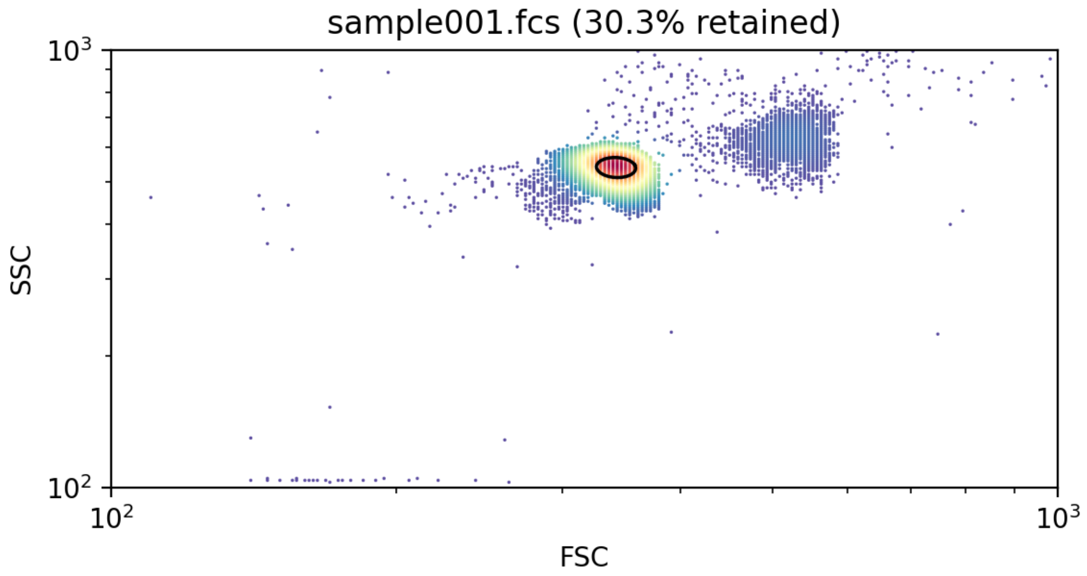
Also, import `numpy` and `pyplot` from `matplotlib`

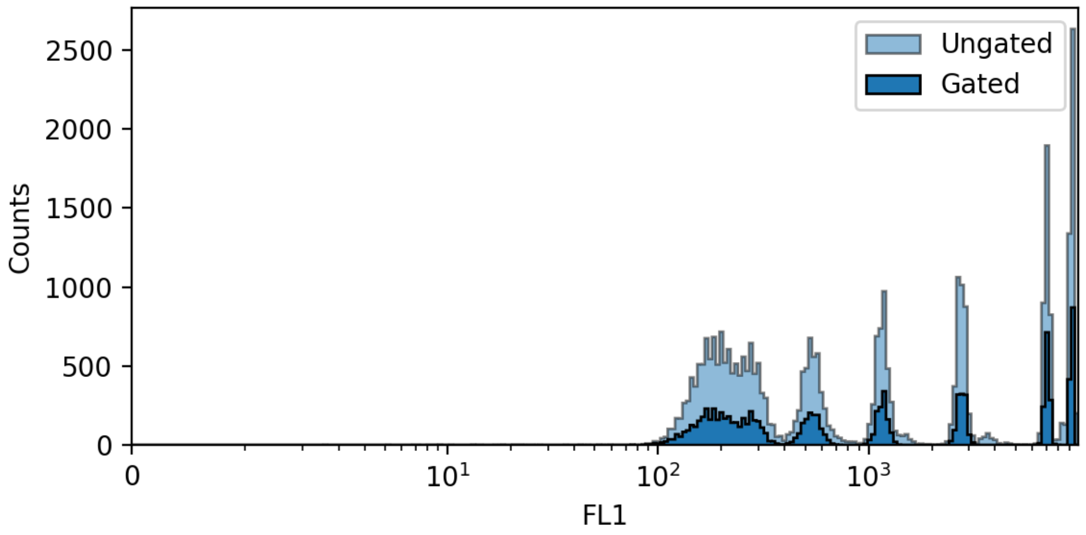
```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
```

Working with Calibration Beads

As mentioned in the *fundamentals* section, conversion to MEF requires measuring calibration beads. `sample001.fcs` in the `FCFiles` folder contains beads data. Let's examine it.

```
>>> b = FlowCal.io.FCSData('FCFiles/sample001.fcs')
>>> b = FlowCal.transform.to_rfi(b)
>>> density_gate_output = FlowCal.gate.density2d(b,
...                                              channels=['FSC', 'SSC'],
...                                              gate_fraction=0.3,
...                                              full_output=True)
>>> b_g = density_gate_output.gated_data
>>> c   = density_gate_output.contour
>>> FlowCal.plot.density_and_hist(b,
...                               gated_data=b_g,
...                               gate_contour=c,
...                               density_channels=['FSC', 'SSC'],
...                               density_params={'mode':'scatter',
...                                               'xlim': [1e2, 1e3],
...                                               'ylim': [1e2, 1e3],
...                                               'sigma': 5.},
...                               hist_channels=['FL1'])
>>> plt.tight_layout()
>>> plt.show()
```





The `FSC`/`SSC` density plot shows two groups of events: the dense group in the middle corresponds to single beads, whereas the fainter cluster on the upper right corresponds to bead agglomerations. Only single beads should be used, so *`FlowCal.gate.density2d()`* is used here to identify single beads automatically. Looking at the `FL1` histogram, we can clearly distinguish 8 subpopulations with different fluorescence levels. Note that the group with the highest fluorescence seems to be close to saturation.

MEF Transformation in `FlowCal`

We saw in the *transformation tutorial* that a transformation function is needed to convert flow cytometry data from raw sensor numbers, as stored in FCS files, to fluorescence values in a.u. Similarly, `FlowCal` uses transformation functions to convert these to MEF. However, these functions have to be generated during analysis using a calibration bead sample. Once a function is generated, though, it can be used to convert many cell samples to MEF, provided that beads and samples have been acquired using the same settings.

Generating a transformation function from calibration beads data is a complicated process, therefore `FlowCal` has an entire module dedicated to it: *`FlowCal.mef`*. The main function in this module, *`FlowCal.mef.`*

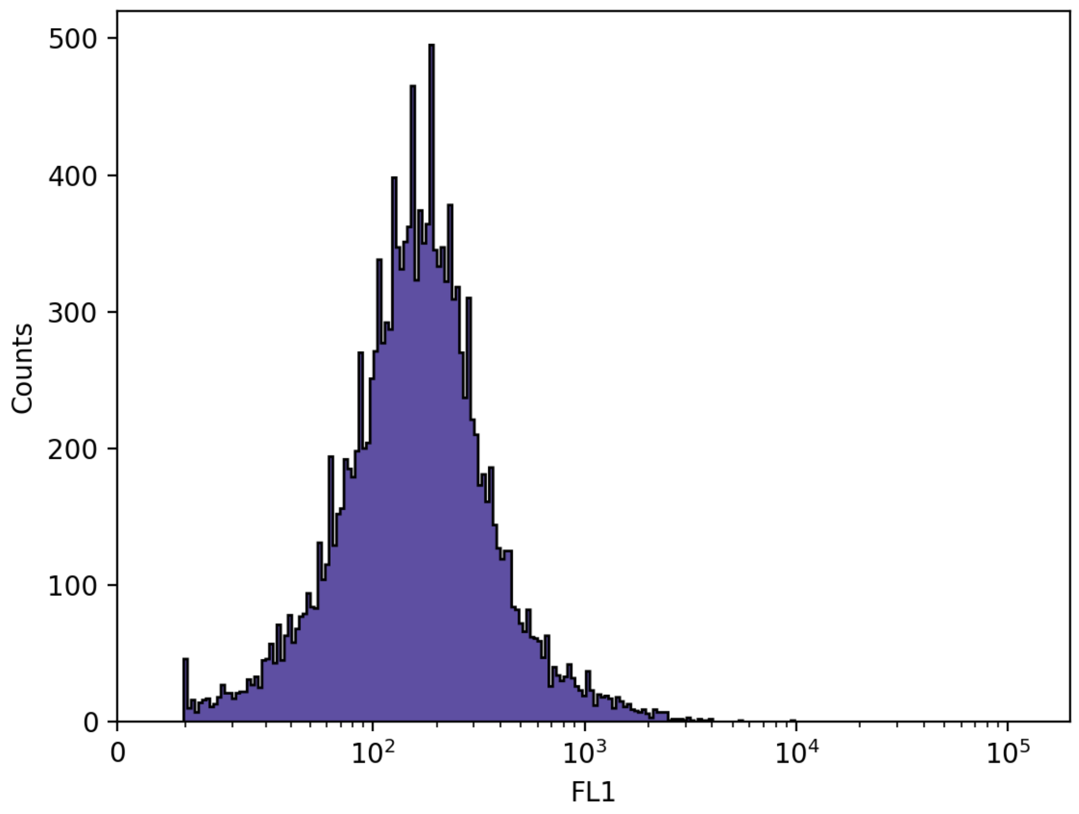
get_transform_fxn(), requires at least the following information: calibration beads data, the names of the channels for which a MEF transformation function should be generated, and manufacturer-provided MEF values of each subpopulation for each channel. Let's now use *FlowCal.mef.get_transform_fxn()* to obtain a transformation function.

```
>>> # Obtain transformation function
>>> # The following MEFL values were provided by the beads' manufacturer
>>> mefl_values = np.array([0, 792, 2079, 6588, 16471, 47497, 137049, 271647])
>>> to_mef = FlowCal.mef.get_transform_fxn(b_g,
...                                         mef_values=mefl_values,
...                                         mef_channels='FL1',
...                                         plot=True)
>>> plt.show()
```

The argument `plot` instructs *FlowCal.mef.get_transform_fxn()* to generate and save plots showing the individual steps of bead data analysis. We will look at these plots and how to interpret them in the next section. We recommend to always generate these plots to confirm that the standard curve was generated properly.

Let's now use `to_mef` to transform fluroescence data to MEF.

```
>>> # Load sample
>>> s = FlowCal.io.FCSData('FCFiles/sample006.fcs')
>>>
>>> # Transform all channels to a.u., and then FL1 to MEF.
>>> s = FlowCal.transform.to_rfi(s)
>>> s = to_mef(s, channels='FL1')
>>>
>>> # Gate
>>> s_g = FlowCal.gate.high_low(s, channels=['FSC', 'SSC'])
>>> s_g = FlowCal.gate.density2d(s_g,
...                              channels=['FSC', 'SSC'],
...                              gate_fraction=0.5)
>>>
>>> # Plot histogram of transformed channel
>>> FlowCal.plot.hist1d(s_g, channel='FL1')
>>> plt.show()
```



`s_g` now contains `FL1` fluorescence values in MEF units. Note that the values in the x axis of the histogram do not match the ones showed before in channel (raw) units or a.u.. This is always true in general, because fluorescence is now expressed in different units.

Generation of a MEF Transformation Function

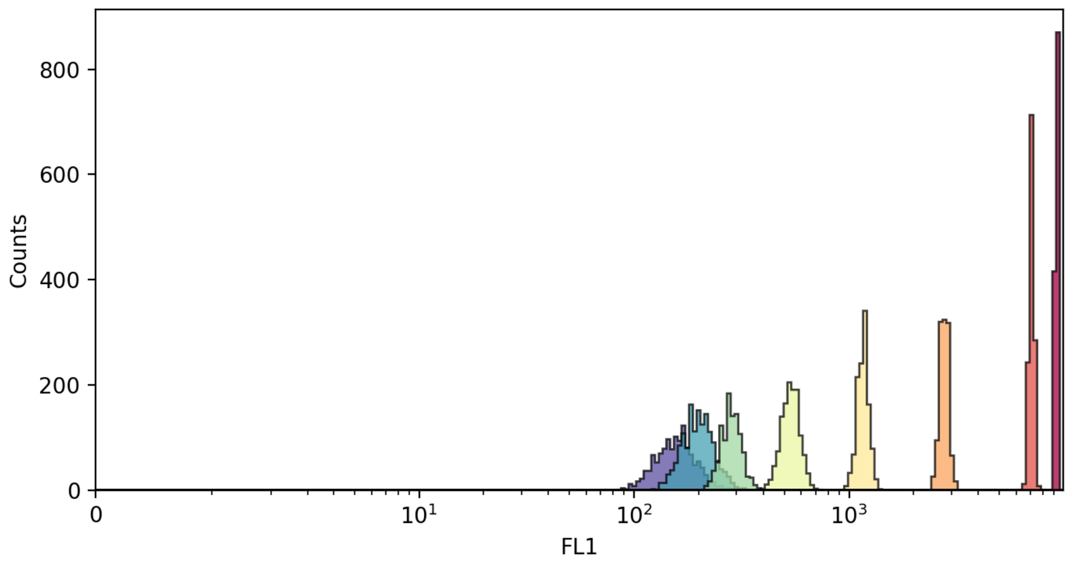
We will now give a short description of the process that *`FlowCal.mef.get_transform_fxn()`* uses to generate a transformation function from beads data. We will also examine the plots produced by *`FlowCal.mef.get_transform_fxn()`* and discuss how these plots can reveal problems with the analysis. In the following, `<beads_filename>` refers to the file name of the FSC cotaining beads data, which was provided to *`FlowCal.mef.get_transform_fxn()`*. This discussion is parallel to the one in the *fundamentals of calibration* document, but at a higher technical level.

Generating a MEF transformation function involves four steps:

1. Identification of Bead Subpopulations

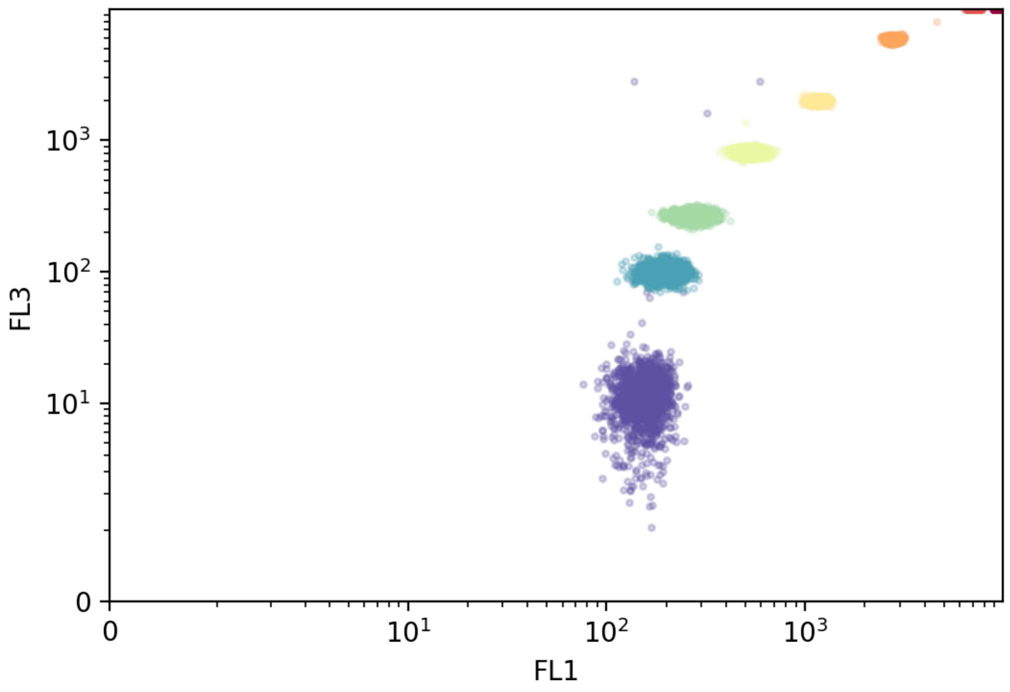
`FlowCal` uses a clustering algorithm to automatically identify the different subpopulations of beads. The algorithm will try to find as many populations as values are provided in `mef_values`.

A plot with a default filename of `clustering_<beads_filename>.png` is generated by *`FlowCal.mef.get_transform_fxn()`* after the completion of this step. This plot is a histogram or scatter plot in which different subpopulations are shown in a different colors. Such plot is shown below, for `sample001.fcs`.



It is always visually clear which events correspond to which groups, and the different colors should correspond to this expectation. If they don't, sometimes it helps to use a different set of fluorescence channels for clustering (see below), or to use a different gating fraction in the previous density gating step.

The default clustering algorithm is Gaussian Mixture Models, implemented in *`FlowCal.mef.clustering_gmm()`*. However, a function implementing another clustering algorithm can be provided to *`FlowCal.mef.get_transform_fxn()`* through the argument `clustering_fxn`. In addition, the argument `clustering_channels` specifies which channels to use for clustering. This can be different than `mef_channels`, the channels for which to generate a standard curve. A plot resulting from clustering with two fluroescence channels is shown below.



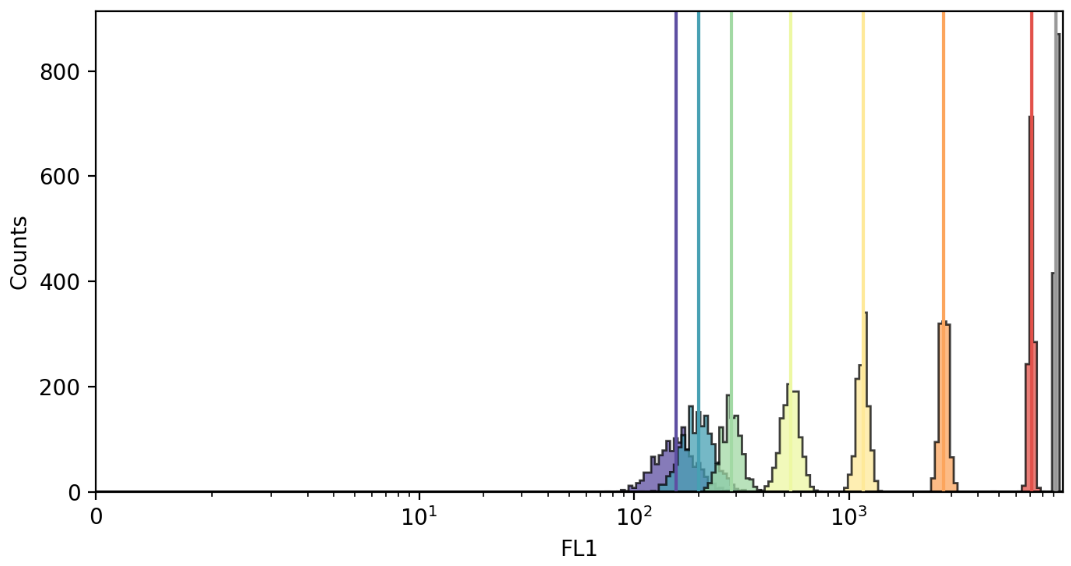
2. Calculation of Population Statistics

For each channel in `mef_channels`, a representative fluorescence value in a.u. is calculated for each subpopulation. By default, the median of each population is used, but this can be customized using the `statistic_fxn` parameter.

3. Population Selection

For each channel in `mef_channels`, subpopulations close to saturation are discarded.

A plot with a default filename of `populations_<channel>_<beads_filename>.png` is generated by *`FlowCal.mef.get_transform_fxn()`* for each channel in `mef_channels` after the completion of this step. This plot is a histogram showing each population, as identified in step one, with vertical lines showing their representative statistic as calculated from step 2, and with the discarded populations colored in grey. Such plot is shown below, for `sample001.fcs` and channel `FL1`.

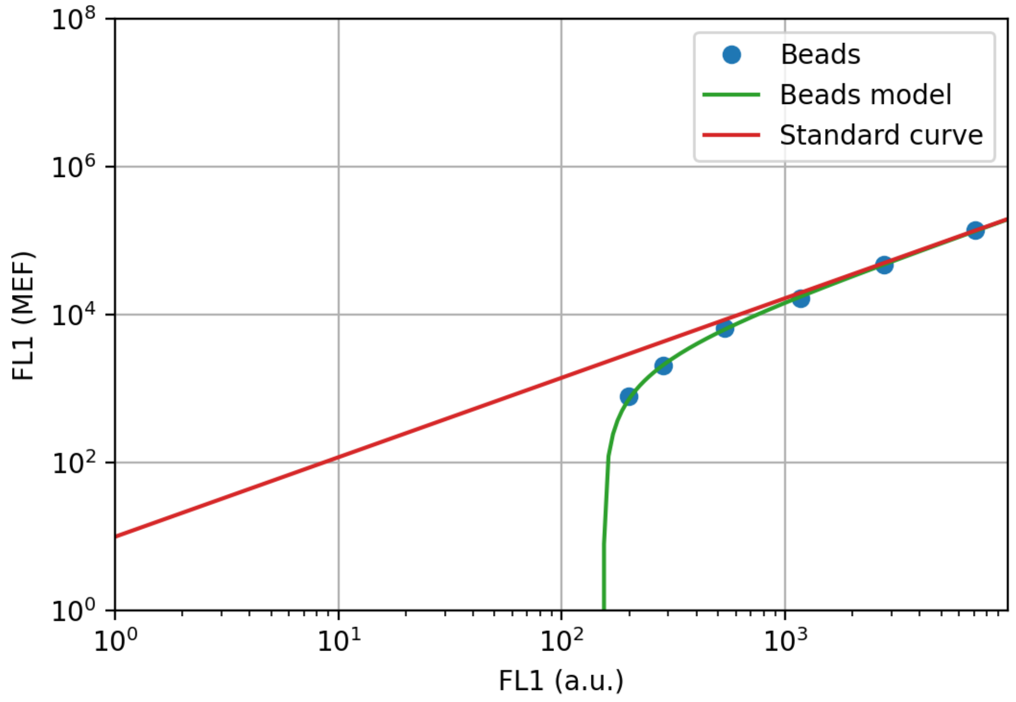


By default, populations whose mean is closer than a few standard deviations from one of the edge values are discarded. This is encoded in the function *`FlowCal.mef.selection_std()`*. A different method can be used by providing a different function to *`FlowCal.mef.get_transform_fxn()`* through the argument `selection_fxn`. This argument can even be None, in which case no populations are discarded. Finally, one can manually discard a population by using `None` as its MEF fluorescence value in `mef_values`. Discarding populations specified in this way is performed in addition to `selection_fxn`.

4. Standard Curve Calculation

A bead fluorescence model is fitted to the fluorescence values of each subpopulation in a.u., as calculated in step 2, and in MEF units, as provided in `mef_values`. A standard curve can then be calculated from the bead fluorescence model.

A plot with a default filename of `std_crv_<channel>_<beads_filename>.png` is generated by *`FlowCal.mef.get_transform_fxn()`* for each channel in `mef_channels` after the completion of this step. This plot shows the fluorescence values of each population in a.u. and MEF, the fitted bead fluorescence model, and the resulting standard curve. Such plot is shown below, for `sample001.fcs` and channel `FL1`.



It is worth noting that the bead fluorescence model and the standard curve are different, in that bead fluorescence is also affected by bead autofluorescence, its fluorescence when no fluorophore is present. To obtain the standard curve, autofluorescence is eliminated from the model. Such a model is fitted in *FlowCal.mef.fit_beads_autofluorescence()*, but a different model can be provided to *FlowCal.mef.get_transform_fxn()* using the argument `fitting_fxn`.

After these steps, a transformation function is generated using the standard curve, and returned.

FlowCal.mef.get_transform_fxn() has more customization options. For more information, consult the reference.

2.4.6 Processing FCS Files with the Excel UI

This tutorial focuses on how to obtain processed flow cytometry data from `FlowCal`'s Excel UI into python. This document assumes that the reader is familiar with `FlowCal`'s Excel UI. For more information, please refer to the *Excel UI documentation*.

To start, navigate to the `examples` directory included with FlowCal, and open a `python` session therein. Then, import `FlowCal` as with any other python module.

```
>>> import FlowCal
```

Introduction

`FlowCal` is a very flexible package that allows the user to perform different gating and transformation operations on flow cytometry data. As we saw in the *MEF tutorial*, the process of transformation to MEF units also allows for a lot of customization. However, for most experiments the user might simply want to follow a procedure similar to this:

1. Open calibration beads files

2. Perform density gating in forward/side scatter to eliminate bead aggregates
3. Obtain standard curves for each fluorescence channel of interest
4. Open cell sample files
5. Perform density gating in forward/side scatter to eliminate aggregates and non-cellular debris.
6. Transform the fluorescence of cell samples to MEF using the standard curves obtained in step 3.

After this, what follows is highly dependent on the type of experiment. Some might be interested, for example, in the geometric mean fluorescence and standard deviation of cell samples as a function of some inducer. For these cases, the Excel UI allows to easily specify a set of FCS files that will be processed as described above, and generate a set of statistics for each fluorescence channel of interest. This is performed through a convenient input Excel file, which can also document other information about the experiment, such as inducer level of each sample.

However, some applications demand more complicated downstream processing, such as n-dimensional fluorescence analysis, which will inevitably require programming. In these cases, one can still use `FlowCal`'s Excel UI to process files as above, and return transformed and gated `FCSData` objects for each specified FCS file to python, along with extra information contained in the input Excel file. This workflow combines the convenience of maintaining experimental information in an Excel file, the consistency of a standard FCS file processing pipeline, and the power of performing numerical analysis in python. We will now describe how to do this.

Processing Samples with the Excel UI

For this tutorial, we will analyze all the data in the `examples/FCFiles` folder using the input Excel file, `examples/experiment.xlsx`. This is the same file described in the *Excel UI documentation*.

First, load the necessary tables from this file.

```
>>> input_file = 'experiment.xlsx'
>>> instruments_table = FlowCal.excel_ui.read_table(input_file,
...                                                      sheetname='Instruments',
...                                                      index_col='ID')
>>> beads_table = FlowCal.excel_ui.read_table(input_file,
...                                              sheetname='Beads',
...                                              index_col='ID')
>>> samples_table = FlowCal.excel_ui.read_table(input_file,
...                                                sheetname='Samples',
...                                                index_col='ID')
```

`FlowCal.excel_ui.read_table()` returns the contents of a sheet from an Excel file as a `pandas DataFrame`. The file name is specified as the first argument, and the `index_col` argument specified which column to use as the `DataFrame`'s index. For more information about `DataFrames`, consult pandas' documentation.

From there, one can obtain the file name and analysis options of each beads file, and call all the necessary `FlowCal` functions to perform density gating and standard curve calculation. Or one could let the Excel UI do all that with the following instruction:

```
>>> beads_samples, mef_transform_fxns = FlowCal.excel_ui.process_beads_table(
...     beads_table,
...     instruments_table,
...     verbose=True,
...     plot=True)
```

`FlowCal.excel_ui.process_beads_table` uses the instruments table and the beads table to automatically open, density-gate, and transform the specified beads files, and generate MEF transformation functions as indicated by the Excel input file. The flags `verbose` and `plot` instruct the function to generate messages for each file being processed, and plots for each step of standard curve calculation, similar to what we saw in the *MEF tutorial*. The output arguments are `beads_samples`, a dictionary of transformed and gated FCSData objects, and `mef_transform_fxns`, a dictionary of MEF transformation functions, each indexed by the ID of the beads files.

In a similar way, `FlowCal`'s Excel UI can automatically density-gate and transform cell samples using a single instruction:

```
>>> samples = FlowCal.excel_ui.process_samples_table(
...     samples_table,
...     instruments_table,
...     mef_transform_fxns=mef_transform_fxns,
...     verbose=True,
...     plot=True)
```

`FlowCal.excel_ui.process_samples_table` uses the instruments and samples tables to open, density-gate, and transform cell samples as specified, and return the processed data as a dictionary of FCSData objects. If the input Excel file specifies that some samples should be transformed to MEF, `FlowCal.excel_ui.process_samples_table` also requires a dictionary with the respective MEF transformation functions (`mef_transform_fxns`), which was provided in the previous step by `FlowCal.excel_ui.process_beads_table`.

This is all the code required to obtain a set of processed cell samples. From here, one can perform any desired analysis on `samples`. Note that `samples_table` contains any other information in the input Excel file not directly used by `FlowCal`, such as inducer concentration, incubation time, etc. This can be used to build an induction curve, fluorescence vs. final optical density (OD), etc.

2.5 FlowCal (Python API) Reference

2.5.1 FlowCal.excel_ui module

`FlowCal`'s Microsoft Excel User Interface.

This module contains functions to read, gate, and transform data from a set of FCS files, as specified by an input Microsoft Excel file. This file should contain the following tables:

- **Instruments**: Describes the instruments used to acquire the samples listed in the other tables. Each instrument is specified by a row containing at least the following fields:
  - **ID**: Short string identifying the instrument. Will be referenced by samples in the other tables.
  - **Forward Scatter Channel**: Name of the forward scatter channel, as specified by the `$PnN` keyword in the associated FCS files.
  - **Side Scatter Channel**: Name of the side scatter channel, as specified by the `$PnN` keyword in the associated FCS files.
  - **Fluorescence Channels**: Name of the fluorescence channels in a comma-separated list, as specified by the `$PnN` keyword in the associated FCS files.
  - **Time Channel**: Name of the time channel, as specified by the `$PnN` keyword in the associated FCS files.
- **Beads**: Describes the calibration beads samples that will be used to calibrate cell samples in the **Samples** table. The following information should be available for each beads sample:
  - **ID**: Short string identifying the beads sample. Will be referenced by cell samples in the **Samples** table.
  - **Instrument ID**: ID of the instrument used to acquire the sample. Must match one of the rows in the **Instruments** table.

  - **File Path**: Path of the FCS file containing the sample's data.
  - **<Fluorescence Channel Name> MEF Values**: The fluorescence in MEF of each bead subpopulation, as given by the manufacturer, as a comma-separated list of numbers. Any element of this list can be replaced with the word `None`, in which case the corresponding subpopulation will not be used when fitting the beads fluorescence model. Note that the number of elements in this list (including the elements equal to `None`) are the number of subpopulations that `FlowCal` will try to find.
  - **Gate fraction**: The fraction of events to keep from the sample after density-gating in the forward/side scatter channels.
  - **Clustering Channels**: The fluorescence channels used to identify the different bead subpopulations.

- **Samples**: Describes the biological samples to be processed. The following information should be available for each sample:
  - **ID**: Short string identifying the sample. Will be used as part of the plot's filenames and in the **Histograms** table in the output Excel file.
  - **Instrument ID**: ID of the instrument used to acquire the sample. Must match one of the rows in the **Instruments** table.
  - **Beads ID**: ID of the beads sample used to convert data to calibrated MEF.
  - **File Path**: Path of the FCS file containing the sample's data.
  - **<Fluorescence Channel Name> Units**: Units to which the event list in the specified fluorescence channel should be converted, and all the subsequent plots and statistics should be reported. Should be one of the following: "Channel" (raw units), "a.u." or "RFI" (arbitrary units) or "MEF" (calibrated Molecules of Equivalent Fluorophore). If "MEF" is specified, the **Beads ID** should be populated, and should correspond to a beads sample with the **MEF Values** specified for the same channel.
  - **Gate fraction**: The fraction of events to keep from the sample after density-gating in the forward/side scatter channels.

Any columns other than the ones specified above can be present, but will be ignored by `FlowCal`.

exception `FlowCal.excel_ui.`**`ExcelUIException`**

Bases: `Exception`

FlowCal Excel UI Error.

`FlowCal.excel_ui.`**`add_beads_stats`**(*beads_table*, *beads_samples*, *mef_outputs=None*)

Add stats fields to beads table.

The following information is added to each row:

- Notes (warnings, errors) resulting from the analysis
- Number of Events
- Acquisition Time (s)

The following information is added for each row, for each channel in which MEF values have been specified:

- Detector voltage (gain)
- Amplification type
- Bead model fitted parameters

Parameters

beads_table [DataFrame] Table specifying bead samples to analyze. For more information about the fields required in this table, please consult the module's documentation.

beads_samples [dict or OrderedDict] FCSData objects from which to calculate statistics. `beads_samples[id]` should correspond to `beads_table.loc[id,:]`.

mef_outputs [dict or OrderedDict, optional] Intermediate results from the generation of the MEF transformation functions, as given by `mef.get_transform_fxn()`. This is used to populate the fields `<channel> Beads Model`, `<channel> Beads Params. Names`, and `<channel> Beads Params. Values`. If specified, `mef_outputs[id]` should correspond to `beads_table.loc[id,:]`.

`FlowCal.excel_ui.`**`add_samples_stats`**`(`*samples_table*, *samples*`)`

Add stats fields to samples table.

The following information is added to each row:

- Notes (warnings, errors) resulting from the analysis
- Number of Events
- Acquisition Time (s)

The following information is added for each row, for each channel in which fluorescence units have been specified:

- Detector voltage (gain)
- Amplification type
- Mean
- Geometric Mean
- Median
- Mode
- Standard Deviation
- Coefficient of Variation (CV)
- Geometric Standard Deviation
- Geometric Coefficient of Variation
- Inter-Quartile Range
- Robust Coefficient of Variation (RCV)

Parameters

samples_table [DataFrame] Table specifying samples to analyze. For more information about the fields required in this table, please consult the module's documentation.

samples [dict or OrderedDict] FCSData objects from which to calculate statistics. `samples[id]` should correspond to `samples_table.loc[id,:]`.

Notes

Geometric statistics (geometric mean, standard deviation, and geometric coefficient of variation) are defined only for positive data. If there are negative events in any relevant channel of any member of *samples*, geometric statistics will only be calculated on the positive events, and a warning message will be written to the "Analysis Notes" field.

`FlowCal.excel_ui.`**`generate_about_table`**`(`*extra_info={}*`)`

Make a table with information about FlowCal and the current analysis.

Parameters

extra_info [dict, optional] Additional keyword:value pairs to include in the table.

Returns

about_table [DataFrame] Table with information about FlowCal and the current analysis, as keyword:value pairs. The following keywords are included: FlowCal version, and date and time of analysis. Keywords and values from *extra_info* are also included.

`FlowCal.excel_ui.`**`generate_histograms_table`**`(`*samples_table*, *samples*, *max_bins=1024*`)`

Generate a table of histograms as a DataFrame.

Parameters

samples_table [DataFrame] Table specifying samples to analyze. For more information about the fields required in this table, please consult the module's documentation.

samples [dict or OrderedDict] FCSData objects from which to calculate statistics. `samples[id]` should correspond to `samples_table.loc[id,:]`.

max_bins [int, optional] Maximum number of bins to use.

Returns

hist_table [DataFrame] A multi-indexed DataFrame. Rows contain the histogram bins and counts for every sample and channel specified in samples_table. *hist_table* is indexed by the sample's ID, the channel name, and whether the row corresponds to bins or counts.

`FlowCal.excel_ui.`**`process_beads_table`**`(`*beads_table*, *instruments_table*, *base_dir='.'*, *verbose=False*, *plot=False*, *plot_dir=None*, *full_output=False*, *get_transform_fxn_kwargs={}*`)`

Process calibration bead samples, as specified by an input table.

This function processes the entries in *beads_table*. For each row, the function does the following:

- Load the FCS file specified in the field "File Path".
- Transform the forward scatter/side scatter and fluorescence channels to RFI
- Remove the 250 first and 100 last events.
- Remove saturated events in the forward scatter and side scatter channels.
- Apply density gating on the forward scatter/side scatter channels.
- Generate a standard curve transformation function, for each fluorescence channel in which the associated MEF values are specified.
- Generate forward/side scatter density plots and fluorescence histograms, and plots of the clustering and fitting steps of standard curve generation, if *plot* = True.

Names of forward/side scatter and fluorescence channels are taken from *instruments_table*.

Parameters

beads_table [DataFrame] Table specifying beads samples to be processed. For more information about the fields required in this table, please consult the module's documentation.

instruments_table [DataFrame] Table specifying instruments. For more information about the fields required in this table, please consult the module's documentation.

base_dir [str, optional] Directory from where all the other paths are specified.

verbose [bool, optional] Whether to print information messages during the execution of this function.

plot [bool, optional] Whether to generate and save density/histogram plots of each sample, and each beads sample.

plot_dir [str, optional] Directory relative to *base_dir* into which plots are saved. If *plot* is False, this parameter is ignored. If `plot==True` and `plot_dir is None`, plot without saving.

full_output [bool, optional] Flag indicating whether to include an additional output, containing intermediate results from the generation of the MEF transformation functions.

get_transform_fxn_kwargs [dict, optional] Additional parameters passed directly to internal `mef.get_transform_fxn()` function call.

Returns

beads_samples [OrderedDict] Processed, gated, and transformed samples, indexed by `beads_table.index`.

mef_transform_fxns [OrderedDict] MEF transformation functions, indexed by `beads_table.index`.

mef_outputs [OrderedDict, only if `full_output==True`] Intermediate results from the generation of the MEF transformation functions. For every entry in *beads_table*, *`FlowCal.mef.get_transform_fxn()`* is called on the corresponding processed and gated beads sample with `full_output=True`, and the full output (a *MEFOutput* `namedtuple`) is added to *mef_outputs*. *mef_outputs* is indexed by `beads_table.index`. Refer to the documentation for *`FlowCal.mef.get_transform_fxn()`* for more information.

`FlowCal.excel_ui.`**`process_samples_table`**`(`*samples_table*, *instruments_table*, *mef_transform_fxns=None*, *beads_table=None*, *base_dir='.'*, *verbose=False*, *plot=False*, *plot_dir=None*`)`

Process flow cytometry samples, as specified by an input table.

The function processes each entry in *samples_table*, and does the following:

- Load the FCS file specified in the field "File Path".
- Transform the forward scatter/side scatter to RFI.
- Transform the fluorescence channels to the units specified in the column "<Channel name> Units".
- Remove the 250 first and 100 last events.
- Remove saturated events in the forward scatter and side scatter channels.
- Apply density gating on the forward scatter/side scatter channels.
- Plot combined forward/side scatter density plots and fluorescence historgrams, if *plot* = True.

Names of forward/side scatter and fluorescence channels are taken from *instruments_table*.

Parameters

samples_table [DataFrame] Table specifying samples to be processed. For more information about the fields required in this table, please consult the module's documentation.

instruments_table [DataFrame] Table specifying instruments. For more information about the fields required in this table, please consult the module's documentation.

mef_transform_fxns [dict or OrderedDict, optional] Dictionary containing MEF transformation functions. If any entry in *samples_table* requires transformation to MEF, a key: value pair must exist in mef_transform_fxns, with the key being equal to the contents of field "Beads ID".

beads_table [DataFrame, optional] Table specifying beads samples used to generate *mef_transform_fxns*. This is used to check if a beads sample was taken at the same acquisition settings as a sample to be transformed to MEF. For any beads sample and channel for which a MEF transformation function has been generated, the following fields should be populated: `<channel> Amp. Type` and `<channel> Detector Volt`. If *beads_table* is not specified, no checking will be performed.

base_dir [str, optional] Directory from where all the other paths are specified.

verbose [bool, optional] Whether to print information messages during the execution of this function.

plot [bool, optional] Whether to generate and save density/histogram plots of each sample, and each beads sample.

plot_dir [str, optional] Directory relative to *base_dir* into which plots are saved. If *plot* is False, this parameter is ignored. If `plot==True` and `plot_dir is None`, plot without saving.

Returns

samples [OrderedDict] Processed, gated, and transformed samples, indexed by `samples_table.index`.

`FlowCal.excel_ui.`**`read_table`**(*filename*, *sheetname*, *index_col=None*, *engine=None*)

Return the contents of an Excel table as a pandas DataFrame.

Parameters

filename [str] Name of the Excel file to read.

sheetname [str or int] Name or index of the sheet inside the Excel file to read.

index_col [str, optional] Column name or index to be used as row labels of the DataFrame. If None, default index will be used.

engine [str, optional] Engine used by *pd.read_excel()* to read Excel file. If None, try 'openpyxl' then 'xlrd'.

Returns

table [DataFrame] A DataFrame containing the data in the specified Excel table. If *index_col* is not None, rows in which their *index_col* field is empty will not be present in *table*.

Raises

ValueError If *index_col* is specified and two rows contain the same *index_col* field.

`FlowCal.excel_ui.`**`run`**(*input_path=None*, *output_path=None*, *verbose=True*, *plot=True*, *hist_sheet=False*)

Run the MS Excel User Interface.

This function performs the following:

1. If *input_path* is not specified, show a dialog to choose an input Excel file.
2. Extract data from the Instruments, Beads, and Samples tables.
3. Process all the bead samples specified in the Beads table.
4. Generate statistics for each bead sample.
5. Process all the cell samples in the Samples table.
6. Generate statistics for each sample.
7. If requested, generate a histogram table for each fluorescent channel specified for each sample.

8. Generate a table with run time, date, FlowCal version, among others.
9. Save statistics and (if requested) histograms in an output Excel file.

Parameters

input_path [str] Path to the Excel file to use as input. If None, show a dialog to select an input file.

output_path [str] Path to which to save the output Excel file. If None, use "<input_path>_output".

verbose [bool, optional] Whether to print information messages during the execution of this function.

plot [bool, optional] Whether to generate and save density/histogram plots of each sample, and each beads sample.

hist_sheet [bool, optional] Whether to generate a sheet in the output Excel file specifying histogram bin information.

`FlowCal.excel_ui.`**`run_command_line`**(*args=None*)

Entry point for the FlowCal and flowcal console scripts.

Parameters

args: list of strings, optional Command line arguments. If None or not specified, get arguments from `sys.argv`.

See also:

`FlowCal.excel_ui.run`

References

http://amir.rachum.com/blog/2017/07/28/python-entry-points/

`FlowCal.excel_ui.`**`show_open_file_dialog`**(*filetypes*)

Show an open file dialog and return the path of the file selected.

Parameters

filetypes [list of tuples] Types of file to show on the dialog. Each tuple on the list must have two elements associated with a filetype: the first element is a description, and the second is the associated extension.

Returns

filename [str] The path of the filename selected, or an empty string if no file was chosen.

`FlowCal.excel_ui.`**`write_workbook`**(*filename*, *table_list*, *column_width=None*)

Write an Excel workbook from a list of tables.

Parameters

filename [str] Name of the Excel file to write.

table_list [list of `(str, DataFrame)` tuples] Tables to be saved as individual sheets in the Excel table. Each tuple contains two values: the name of the sheet to be saved as a string, and the contents of the table as a DataFrame.

column_width: int or float, optional The column width to use when saving the spreadsheet. If None, calculate width automatically from the maximum number of characters in each column.

2.5.2 FlowCal.gate module

Functions for gating flow cytometry data.

All gate functions are of the following form:

```
gated_data = gate(data, channels, *args, **kwargs)

(gated_data, mask, contour, ...) = gate(data, channels, *args,
                                        **kwargs, full_output=True)
```

where *data* is a NxD FCSData object or numpy array describing N cytometry events with D channels, *channels* specifies the channels in which to perform gating, and *args* and *kwargs* are gate-specific parameters. *gated_data* is the gated result, as an FCSData object or numpy array, *mask* is a bool array specifying the gate mask, and *contour* is an optional list of 2D numpy arrays containing the x-y coordinates of the contour surrounding the gated region, which can be used when plotting a 2D density diagram or scatter plot.

class `FlowCal.gate.`**`Density2dGateOutput`**(*gated_data*, *mask*, *contour*, *bin_edges*, *bin_mask*)
Bases: `tuple`

Attributes

bin_edges Alias for field number 3

bin_mask Alias for field number 4

contour Alias for field number 2

gated_data Alias for field number 0

mask Alias for field number 1

Methods

`count`(value, /)	Return number of occurrences of value.
`index`(value[, start, stop])	Return first index of value.

bin_edges
Alias for field number 3

bin_mask
Alias for field number 4

contour
Alias for field number 2

gated_data
Alias for field number 0

mask
Alias for field number 1

class `FlowCal.gate.`**`EllipseGateOutput`**(*gated_data*, *mask*, *contour*)
Bases: `tuple`

Attributes

`contour` Alias for field number 2

`gated_data` Alias for field number 0

`mask` Alias for field number 1

Methods

`count`(value, /)	Return number of occurrences of value.
`index`(value[, start, stop])	Return first index of value.

`contour`
Alias for field number 2

`gated_data`
Alias for field number 0

`mask`
Alias for field number 1

class `FlowCal.gate.`**`HighLowGateOutput`**(*gated_data*, *mask*)
Bases: `tuple`

Attributes

`gated_data` Alias for field number 0

`mask` Alias for field number 1

Methods

`count`(value, /)	Return number of occurrences of value.
`index`(value[, start, stop])	Return first index of value.

`gated_data`
Alias for field number 0

`mask`
Alias for field number 1

class `FlowCal.gate.`**`StartEndGateOutput`**(*gated_data*, *mask*)
Bases: `tuple`

Attributes

`gated_data` Alias for field number 0

`mask` Alias for field number 1

Methods

`count`(value, /)	Return number of occurrences of value.
`index`(value[, start, stop])	Return first index of value.

`gated_data`
Alias for field number 0

`mask`
Alias for field number 1

`FlowCal.gate.`**`density2d`**(*data, channels=[0, 1], bins=1024, gate_fraction=0.65, xscale='logicle', yscale='logicle', sigma=10.0, bin_mask=None, full_output=False*)

Gate that preserves events in the region with highest density.

Gate out all events in *data* but those near regions of highest density for the two specified channels.

Parameters

data [FCSData or numpy array] NxD flow cytometry data where N is the number of events and D is the number of parameters (aka channels).

channels [list of int, list of str, optional] Two channels on which to perform gating.

bins [int or array_like or [int, int] or [array, array], optional] Bins used for gating:

- If None, use `data.hist_bins` to obtain bin edges for both axes. None is not allowed if `data.hist_bins` is not available.
- If int, *bins* specifies the number of bins to use for both axes. If `data.hist_bins` exists, it will be used to generate a number *bins* of bins.
- If array_like, *bins* directly specifies the bin edges to use for both axes.
- If [int, int], each element of *bins* specifies the number of bins for each axis. If `data.hist_bins` exists, use it to generate `bins[0]` and `bins[1]` bin edges, respectively.
- If [array, array], each element of *bins* directly specifies the bin edges to use for each axis.
- Any combination of the above, such as [int, array], [None, int], or [array, int]. In this case, None indicates to generate bin edges using `data.hist_bins` as above, int indicates the number of bins to generate, and an array directly indicates the bin edges. Note that None is not allowed if `data.hist_bins` does not exist.

gate_fraction [float, optional] Fraction of events to retain after gating. Should be between 0 and 1, inclusive.

xscale [str, optional] Scale of the bins generated for the x axis, either `linear`, `log`, or `logicle`. *xscale* is ignored in *bins* is an array or a list of arrays.

yscale [str, optional] Scale of the bins generated for the y axis, either `linear`, `log`, or `logicle`. *yscale* is ignored in *bins* is an array or a list of arrays.

sigma [scalar or sequence of scalars, optional] Standard deviation for Gaussian kernel used by *scipy.ndimage.filters.gaussian_filter* to smooth 2D histogram into a density.

bin_mask [2D numpy array of bool, optional] A 2D mask array that selects the 2D histogram bins permitted by the gate. Corresponding bin edges should be specified via *bins*. If *bin_mask* is specified, *gate_fraction* and *sigma* are ignored.

full_output [bool, optional] Flag specifying to return additional outputs. If true, the outputs are given as a namedtuple.

Returns

gated_data [FCSData or numpy array] Gated flow cytometry data of the same format as *data*.

mask [numpy array of bool, only if `full_output==True`] Boolean gate mask used to gate data such that `gated_data = data[mask]`.

contour [list of 2D numpy arrays, only if `full_output==True`] List of 2D numpy array(s) of x-y coordinates tracing out the edge of the gated region. If *bin_mask* is specified, *contour* is None.

bin_edges [2-tuple of numpy arrays, only if `full_output==True`] X-axis and y-axis bin edges used by the np.histogram2d() command that bins events (bin_edges=(x_edges,y_edges)).

bin_mask [2D numpy array of bool, only if `full_output==True`] A 2D mask array that selects the 2D histogram bins permitted by the gate.

Raises

ValueError If more or less than 2 channels are specified.

ValueError If *data* has less than 2 dimensions or less than 2 events.

Exception If an unrecognized matplotlib Path code is encountered when attempting to generate contours.

Notes

The algorithm for gating based on density works as follows:

1) Calculate 2D histogram of *data* in the specified channels.
2) Map each event from *data* to its histogram bin (implicitly gating out any events which exist outside specified *bins*).
3) Use *gate_fraction* to determine number of events to retain (rounded up). Only events which are not implicitly gated out are considered.
4) Smooth 2D histogram using a 2D Gaussian filter.
5) Normalize smoothed histogram to obtain valid probability mass function (PMF).
6) Sort bins by probability.
7) Accumulate events (starting with events belonging to bin with highest probability ("densest") and proceeding to events belonging to bins with lowest probability) until at least the desired number of events is achieved. While the algorithm attempts to get as close to *gate_fraction* fraction of events as possible, more events may be retained based on how many events fall into each histogram bin (since entire bins are retained at a time, not individual events).

`FlowCal.gate.`**`ellipse`**(*data*, *channels*, *center*, *a*, *b*, *theta=0*, *log=False*, *full_output=False*)

Gate that preserves events inside an ellipse-shaped region.

Events are kept if they satisfy the following relationship:

```
(x/a)**2 + (y/b)**2 <= 1
```

where *x* and *y* are the coordinates of the event list, after substracting *center* and rotating by *-theta*. This is mathematically equivalent to maintaining the events inside an ellipse with major axis *a*, minor axis *b*, center at *center*, and tilted by *theta*.

Parameters

data [FCSData or numpy array] NxD flow cytometry data where N is the number of events and D is the number of parameters (aka channels).

channels [list of int, list of str] Two channels on which to perform gating.

center, a, b, theta (optional) [float] Ellipse parameters. *a* is the major axis, *b* is the minor axis.

log [bool, optional] Flag specifying that log10 transformation should be applied to *data* before gating.

full_output [bool, optional] Flag specifying to return additional outputs. If true, the outputs are given as a namedtuple.

Returns

gated_data [FCSData or numpy array] Gated flow cytometry data of the same format as *data*.

mask [numpy array of bool, only if `full_output==True`] Boolean gate mask used to gate data such that `gated_data = data[mask]`.

contour [list of 2D numpy arrays, only if `full_output==True`] List of 2D numpy array(s) of x-y coordinates tracing out the edge of the gated region.

Raises

ValueError If more or less than 2 channels are specified.

`FlowCal.gate.`**`high_low`**(*data*, *channels=None*, *high=None*, *low=None*, *full_output=False*)

Gate out high and low values across all specified channels.

Gate out events in *data* with values in the specified channels which are larger than or equal to *high* or less than or equal to *low*.

Parameters

data [FCSData or numpy array] NxD flow cytometry data where N is the number of events and D is the number of parameters (aka channels).

channels [int, str, list of int, list of str, optional] Channels on which to perform gating. If None, use all channels.

high, low [int, float, optional] High and low threshold values. If None, *high* and *low* will be taken from `data.range` if available, otherwise `np.inf` and `-np.inf` will be used.

full_output [bool, optional] Flag specifying to return additional outputs. If true, the outputs are given as a namedtuple.

Returns

gated_data [FCSData or numpy array] Gated flow cytometry data of the same format as *data*.

mask [numpy array of bool, only if `full_output==True`] Boolean gate mask used to gate data such that `gated_data = data[mask]`.

`FlowCal.gate.`**`start_end`**(*data*, *num_start=250*, *num_end=100*, *full_output=False*)

Gate out first and last events.

Parameters

data [FCSData or numpy array] NxD flow cytometry data where N is the number of events and D is the number of parameters (aka channels).

num_start, num_end [int, optional] Number of events to gate out from beginning and end of *data*. Ignored if less than 0.

full_output [bool, optional] Flag specifying to return additional outputs. If true, the outputs are given as a namedtuple.

Returns

gated_data [FCSData or numpy array] Gated flow cytometry data of the same format as *data*.

mask [numpy array of bool, only if `full_output==True`] Boolean gate mask used to gate data such that `gated_data = data[mask]`.

Raises

ValueError If the number of events to discard is greater than the total number of events in *data*.

2.5.3 FlowCal.io module

Classes and utiliy functions for reading FCS files.

class `FlowCal.io.FCSData`

Bases: `numpy.ndarray`

Object containing events data from a flow cytometry sample.

An *FCSData* object is an NxD numpy array representing N cytometry events with D dimensions (channels) extracted from the DATA segment of an FCS file. Indexing along the second axis can be performed by channel name, which allows to easily select data from one or several channels. Otherwise, an *FCSData* object can be treated as a numpy array for most purposes.

Information regarding the acquisition date, time, and information about the detector and the amplifiers are parsed from the TEXT segment of the FCS file and exposed as attributes. The TEXT and ANALYSIS segments are also exposed as attributes.

Parameters

infile [str or file-like] Reference to the associated FCS file.

Notes

FCSData uses *FCSFile* to parse an FCS file. All restrictions on the FCS file format and the Exceptions spcecified for FCSFile also apply to FCSData.

Parsing of some non-standard files is supported [4].

References

[1], [2], [3], [4]

Examples

Load an FCS file into an FCSData object

```
>>> import FlowCal
>>> d = FlowCal.io.FCSData('test/Data001.fcs')
```

Check channel names

```
>>> print d.channels
('FSC-H', 'SSC-H', 'FL1-H', 'FL2-H', 'FL3-H', 'Time')
```

Check the size of FCSData

```
>>> print d.shape
(20949, 6)
```

Get the first 100 events

```
>>> d_sub = d[:100]
>>> print d_sub.shape
(100, 6)
```

Retain only fluorescence channels

```
>>> d_fl = d[:, ['FL1-H', 'FL2-H', 'FL3-H']]
>>> d_fl.channels
('FL1-H', 'FL2-H', 'FL3-H')
```

Channel slicing can also be done with integer indices

```
>>> d_fl_2 = d[:, [2, 3, 4]]
>>> print d_fl_2.channels
('FL1-H', 'FL2-H', 'FL3-H')
>>> import numpy as np
>>> np.all(d_fl == d_fl_2)
True
```

Attributes

infile [str or file-like] Reference to the associated FCS file.

text [dict] Dictionary of key-value entries from the TEXT segment.

analysis [dict] Dictionary of key-value entries from the ANALYSIS segment.

data_type [str] Type of data in the FCS file's DATA segment.

time_step [float] Time step of the time channel.

acquisition_start_time [time or datetime] Acquisition start time, as a python time or datetime object.

acquisition_end_time [time or datetime] Acquisition end time, as a python time or datetime object.

acquisition_time [float] Acquisition time, in seconds.

channels [tuple] The name of the channels contained in *FCSData*.

Methods

amplification_type([channels])	Get the amplification type used for the specified channel(s).
detector_voltage([channels])	Get the detector voltage used for the specified channel(s).
amplifier_gain([channels])	Get the amplifier gain used for the specified channel(s).
channel_labels([channels])	Get the label of the specified channel(s).
range([channels])	Get the range of the specified channel(s).
resolution([channels])	Get the resolution of the specified channel(s).
hist_bins([channels, nbins, scale])	Get histogram bin edges for the specified channel(s).

acquisition_end_time
Acquisition end time, as a python time or datetime object.

acquisition_end_time is taken from the $ETIM keyword parameter in the TEXT segment of the FCS file. If date information is also found, *acquisition_end_time* is a datetime object with the acquisition date. If not, *acquisition_end_time* is a datetime.time object. If no end time is found in the FCS file, return None.

acquisition_start_time
Acquisition start time, as a python time or datetime object.

acquisition_start_time is taken from the $BTIM keyword parameter in the TEXT segment of the FCS file. If date information is also found, *acquisition_start_time* is a datetime object with the acquisition date. If not, *acquisition_start_time* is a datetime.time object. If no start time is found in the FCS file, return None.

acquisition_time
Acquisition time, in seconds.

The acquisition time is calculated using the 'time' channel by default (channel name is case independent). If the 'time' channel is not available, the acquisition_start_time and acquisition_end_time, extracted from the $BTIM and $ETIM keyword parameters will be used. If these are not found, None will be returned.

amplification_type(*channels=None*)
Get the amplification type used for the specified channel(s).

Each channel uses one of two amplification types: linear or logarithmic. This function returns, for each channel, a tuple of two numbers, in which the first number indicates the number of decades covered by the logarithmic amplifier, and the second indicates the linear value corresponding to the channel value zero. If the first value is zero, the amplifier used is linear

The amplification type for channel "n" is extracted from the required $PnE parameter.

Parameters

channels [int, str, list of int, list of str] Channel(s) for which to get the amplification type. If None, return a list with the amplification type of all channels, in the order of `FCSData.channels`.

amplifier_gain(*channels=None*)
Get the amplifier gain used for the specified channel(s).

The amplifier gain for channel "n" is extracted from the $PnG parameter, if available.

Parameters

channels [int, str, list of int, list of str] Channel(s) for which to get the amplifier gain. If None, return a list with the amplifier gain of all channels, in the order of `FCSData.channels`.

analysis
Dictionary of key-value entries from the ANALYSIS segment.

channel_labels(*channels=None*)
Get the label of the specified channel(s).

The label for channel "n" is extracted from the $PnS parameter, if available.

Parameters

channels [int, str, list of int, list of str] Channel(s) for which to get the label. If None, return a list with the label of all channels, in the order of `FCSData.channels`.

channels
The name of the channels contained in *FCSData*.

data_type
Type of data in the FCS file's DATA segment.

data_type is 'I' if the data type is integer, 'F' for floating point, and 'D' for double.

detector_voltage(*channels=None*)
Get the detector voltage used for the specified channel(s).

The detector voltage for channel "n" is extracted from the $PnV parameter, if available.

Parameters

channels [int, str, list of int, list of str] Channel(s) for which to get the detector voltage. If None, return a list with the detector voltage of all channels, in the order of `FCSData.channels`.

hist_bins(*channels=None*, *nbins=None*, *scale='logicle'*, ***kwargs*)
Get histogram bin edges for the specified channel(s).

These cover the range specified in `FCSData.range(channels)` with a number of bins *nbins*, with linear, logarithmic, or logicle spacing.

Parameters

channels [int, str, list of int, list of str] Channel(s) for which to generate histogram bins. If None, return a list with bins for all channels, in the order of `FCSData.channels`.

nbins [int or list of ints, optional] The number of bins to calculate. If *channels* specifies a list of channels, *nbins* should be a list of integers. If *nbins* is None, use `FCSData.resolution(channel)`.

scale [str, optional] Scale in which to generate bins. Can be either `linear`, `log`, or `logicle`.

kwargs [optional] Keyword arguments specific to the selected bin scaling. Linear and logarithmic scaling do not use additional arguments. For logicle scaling, the following parameters can be provided:

T [float, optional] Maximum range of data. If not provided, use `range[1]`.

M [float, optional] (Asymptotic) number of decades in scaled units. If not provided, calculate from the following:

```
max(4.5, 4.5 / np.log10(262144) * np.log10(T))
```

W [float, optional] Width of linear range in scaled units. If not provided, calculate using the following relationship:

```
W = (M - log10(T / abs(r))) / 2
```

Where `r` is the minimum negative event. If no negative events are present, W is set to zero.

Notes

If `range[0]` is equal or less than zero and *scale* is `log`, the lower limit of the range is replaced with one.

Logicle scaling uses the LogicleTransform class in the plot module.

References

Method Avoids Deceptive Effects of Logarithmic Scaling for Low Signals and Compensated Data," Cytometry Part A 69A:541-551, 2006, PMID 16604519.

[1]

infile
Reference to the associated FCS file.

range(*channels=None*)
Get the range of the specified channel(s).

The range is a two-element list specifying the smallest and largest values that an event in a channel should have. Note that with floating point data, some events could have values outside the range in either direction due to instrument compensation.

The range should be transformed along with the data when passed through a transformation function.

The range of channel "n" is extracted from the $PnR parameter as `[0, $PnR - 1]`.

Parameters

channels [int, str, list of int, list of str] Channel(s) for which to get the range. If None, return a list with the range of all channels, in the order of `FCSData.channels`.

resolution(*channels=None*)
Get the resolution of the specified channel(s).

The resolution specifies the number of different values that the events can take. The resolution is directly obtained from the $PnR parameter.

Parameters

channels [int, str, list of int, list of str] Channel(s) for which to get the resolution. If None, return a list with the resolution of all channels, in the order of `FCSData.channels`.

text
Dictionary of key-value entries from the TEXT segment.

text includes items from the TEXT segment and optional supplemental TEXT segment.

time_step
Time step of the time channel.

The time step is such that `self[:,'Time']*time_step` is in seconds. If no time step was found in the FCS file, *time_step* is None.

class FlowCal.io.**FCSFile**(*infile*)
Bases: `object`

Class representing an FCS flow cytometry data file.

This class parses a binary FCS file and exposes a read-only view of the HEADER, TEXT, DATA, and ANALYSIS segments via Python-friendly data structures.

Parameters

infile [str or file-like] Reference to the associated FCS file.

Raises

NotImplementedError If $MODE is not 'L'.

NotImplementedError If $DATATYPE is not 'I', 'F', or 'D'.

NotImplementedError If $DATATYPE is 'I' but data is not byte aligned.

NotImplementedError If $BYTEORD is not big endian ('4,3,2,1' or '2,1') or little endian ('1,2,3,4', '1,2').

ValueError If primary TEXT segment does not start with delimiter.

ValueError If TEXT-like segment has odd number of total extracted keys and values (indicating an unpaired key or value).

ValueError If calculated DATA segment size (as determined from the number of events, the number of parameters, and the number of bytes per data point) does not match size specified in HEADER segment offsets.

Warning If more than one data set is detected in the same file.

Warning If the ANALYSIS segment was not successfully parsed.

Notes

The Flow Cytometry Standard (FCS) describes the de facto standard file format used by flow cytometry acquisition and analysis software to record flow cytometry data to and load flow cytometry data from a file. The standard dictates that each file must have the following segments: HEADER, TEXT, and DATA. The HEADER segment contains version information and byte offset values of other segments, the TEXT segment contains delimited key-value pairs containing acquisition information, and the DATA segment contains the recorded flow cytometry data. The file may optionally have an ANALYSIS segment (structurally identicaly to the TEXT segment), a supplemental TEXT segment (according to more recent versions of the standard), and user-defined OTHER segments.

This class supports a subset of the FCS3.1 standard which should be backwards compatible with FCS3.0 and FCS2.0. The FCS file must be of the following form:

- $MODE = 'L' (list mode; histogram mode is not supported).
- $DATATYPE = 'I' (unsigned binary integers), 'F' (single precision floating point), or 'D' (double precision floating point). 'A' (ASCII) is not supported.
- If $DATATYPE = 'I', $PnB % 8 = 0 (byte aligned) for all parameters (aka channels).
- $BYTEORD = '4,3,2,1' (big endian) or '1,2,3,4' (little endian).
- One data set per file.

For more information on the TEXT segment keywords (e.g. $MODE, $DATATYPE, etc.), see [1], [2], and [3].

References

[1], [2], [3]

Attributes

infile [str or file-like] Reference to the associated FCS file.

header [namedtuple] `namedtuple` containing version information and byte offset

text [dict] Dictionary of key-value entries from TEXT segment and optional supplemental TEXT segment.

data [numpy array] Unwriteable NxD numpy array describing N cytometry events observing D data dimensions.

analysis [dict] Dictionary of key-value entries from ANALYSIS segment.

analysis
Dictionary of key-value entries from ANALYSIS segment.

data
Unwriteable NxD numpy array describing N cytometry events observing D data dimensions.

header
`namedtuple` containing version information and byte offset values of other FCS segments in the following order:

- version : str
- text_begin : int
- text_end : int
- data_begin : int
- data_end : int
- analysis_begin : int
- analysis_end : int

infile
Reference to the associated FCS file.

text
Dictionary of key-value entries from TEXT segment and optional supplemental TEXT segment.

`FlowCal.io.`**`read_fcs_data_segment`**(*buf*, *begin*, *end*, *datatype*, *num_events*, *param_bit_widths*, *big_endian*, *param_ranges=None*)

Read DATA segment of FCS file.

Parameters

buf [file-like object] Buffer containing data to interpret as DATA segment.

begin [int] Offset (in bytes) to first byte of DATA segment in *buf*.

end [int] Offset (in bytes) to last byte of DATA segment in *buf*.

datatype [{'I', 'F', 'D', 'A'}] String specifying FCS file datatype (see $DATATYPE keyword from FCS standards). Supported datatypes include 'I' (unsigned binary integer), 'F' (single precision floating point), and 'D' (double precision floating point). 'A' (ASCII) is recognized but not supported.

num_events [int] Total number of events (see $TOT keyword from FCS standards).

param_bit_widths [array-like] Array specifying parameter (aka channel) bit width for each parameter (see $PnB keywords from FCS standards). The length of *param_bit_widths* should match the $PAR keyword value from the FCS standards (which indicates the total number of parameters). If *datatype* is 'I', data must be byte aligned (i.e. all parameter bit widths should be divisible by 8), and data are upcast to the nearest uint8, uint16, uint32, or uint64 data type. Bit widths larger than 64 bits are not supported.

big_endian [bool] Endianness of computer used to acquire data (see $BYTEORD keyword from FCS standards). True implies big endian; False implies little endian.

param_ranges [array-like, optional] Array specifying parameter (aka channel) range for each parameter (see $PnR keywords from FCS standards). Used to ensure erroneous values are not read from DATA segment by applying a bit mask to remove unused bits. The length of *param_ranges* should match the $PAR keyword value from the FCS standards (which indicates the total number of parameters). If None, no masking is performed.

Returns

data [numpy array] NxD numpy array describing N cytometry events observing D data dimensions.

Raises

ValueError If lengths of *param_bit_widths* and *param_ranges* don't match.

ValueError If calculated DATA segment size (as determined from the number of events, the number of parameters, and the number of bytes per data point) does not match size specified by *begin* and *end*.

ValueError If *param_bit_widths* doesn't agree with *datatype* for single precision or double precision floating point (i.e. they should all be 32 or 64, respectively).

ValueError If *datatype* is unrecognized.

NotImplementedError If *datatype* is 'A'.

NotImplementedError If *datatype* is 'I' but data is not byte aligned.

References

[1], [2], [3]

`FlowCal.io.`**`read_fcs_header_segment`**(*buf*, *begin=0*)

Read HEADER segment of FCS file.

Parameters

buf [file-like object] Buffer containing data to interpret as HEADER segment.

begin [int] Offset (in bytes) to first byte of HEADER segment in *buf*.

Returns

header [namedtuple] Version information and byte offset values of other FCS segments (see FCS standards for more information) in the following order:

- version : str
- text_begin : int
- text_end : int
- data_begin : int
- data_end : int
- analysis_begin : int
- analysis_end : int

Notes

Blank ANALYSIS segment offsets are converted to zeros.

OTHER segment offsets are ignored (see [1], [2], and [3]).

References

[1], [2], [3]

`FlowCal.io.`**`read_fcs_text_segment`**(*buf*, *begin*, *end*, *delim=None*, *supplemental=False*)

Read TEXT segment of FCS file.

Parameters

buf [file-like object] Buffer containing data to interpret as TEXT segment.

begin [int] Offset (in bytes) to first byte of TEXT segment in *buf*.

end [int] Offset (in bytes) to last byte of TEXT segment in *buf*.

delim [str, optional] 1-byte delimiter character which delimits key-value entries of TEXT segment. If None and `supplemental==False`, will extract delimiter as first byte of TEXT segment.

supplemental [bool, optional] Flag specifying that segment is a supplemental TEXT segment (see FCS3.0 and FCS3.1), in which case a delimiter (`delim`) must be specified.

Returns

text [dict] Dictionary of key-value entries extracted from TEXT segment.

delim [str or None] String containing delimiter or None if TEXT segment is empty.

Raises

ValueError If supplemental TEXT segment (`supplemental==True`) but `delim` is not specified.

ValueError If primary TEXT segment (`supplemental==False`) does not start with delimiter.

ValueError If first keyword starts with delimiter (e.g. a primary TEXT segment with the following contents: ///k1/v1/k2/v2/).

ValueError If odd number of keys + values detected (indicating an unpaired key or value).

ValueError If TEXT segment is ill-formed (unable to be parsed according to the FCS standards).

Notes

ANALYSIS segments and supplemental TEXT segments are parsed the same way, so this function can also be used to parse ANALYSIS segments.

This function does *not* automatically parse and accumulate additional TEXT-like segments (e.g. supplemental TEXT segments or ANALYSIS segments) referenced in the originally specified TEXT segment.

References

[1], [2], [3]

2.5.4 FlowCal.mef module

Functions for transforming flow cytometer data to MEF units.

`FlowCal.mef.`**`clustering_gmm`**(*data*, *n_clusters*, *tol=1e-07*, *min_covar=None*, *scale='logicle'*)

Find clusters in an array using a Gaussian Mixture Model.

Before clustering, *data* can be automatically rescaled as specified by the *scale* argument.

Parameters

data [FCSData or array_like] Data to cluster.

n_clusters [int] Number of clusters to find.

tol [float, optional] Tolerance for convergence. Directly passed to either `GaussianMixture` or `GMM`, depending on `scikit-learn`'s version.

min_covar [float, optional] The minimum trace that the initial covariance matrix will have. If `scikit-learn`'s version is older than 0.18, *min_covar* is also passed directly to `GMM`.

scale [str, optional] Rescaling applied to *data* before performing clustering. Can be either `linear` (no rescaling), `log`, or `logicle`.

Returns

labels [array] Nx1 array with labels for each element in *data*, assigning `data[i]` to cluster `labels[i]`.

Notes

A Gaussian Mixture Model finds clusters by fitting a linear combination of *n_clusters* Gaussian probability density functions (pdf) to *data* using Expectation Maximization (EM).

This method can be fairly sensitive to the initial parameter choice. To generate a reasonable set of initial conditions, *clustering_gmm* first divides all points in *data* into *n_clusters* groups of the same size based on their Euclidean distance to the minimum value. Then, for each group, the 50% samples farther away from the mean are discarded. The mean and covariance are calculated from the remaining samples of each group, and used as initial conditions for the GMM EM algorithm.

clustering_gmm internally uses a *GaussianMixture* object from the `scikit-learn` library (`GMM` if `scikit-learn`'s version is lower than 0.18), with full covariance matrices for each cluster. For more information, consult `scikit-learn`'s documentation.

`FlowCal.mef.`**`fit_beads_autofluorescence`**(*fl_rfi*, *fl_mef*)

Fit a standard curve using a beads model with autofluorescence.

Parameters

fl_rfi [array] Fluorescence values of bead populations in units of Relative Fluorescence Intensity (RFI).

fl_mef [array] Fluorescence values of bead populations in MEF units.

Returns

std_crv [function] Standard curve that transforms fluorescence values from RFI to MEF units. This function has the signature `y = std_crv(x)`, where *x* is some fluorescence value in RFI and *y* is the same fluorescence expressed in MEF units.

beads_model [function] Fluorescence model of calibration beads. This function has the signature `y = beads_model(x)`, where *x* is the fluorescence of some bead population in RFI units and *y* is the same fluorescence expressed in MEF units, without autofluorescence.

beads_params [array] Fitted parameters of the bead fluorescence model: `[m, b, fl_mef_auto]`.

beads_model_str [str] String representation of the beads model used.

beads_params_names [list of str] Names of the parameters in a list, in the same order as they are given in *beads_params*.

Notes

The following model is used to describe bead fluorescence:

```
m*log(fl_rfi[i]) + b = log(fl_mef_auto + fl_mef[i])
```

where `fl_rfi[i]` is the fluorescence of bead subpopulation `i` in RFI units and `fl_mef[i]` is the corresponding fluorescence in MEF units. The model includes 3 parameters: `m` (slope), `b` (intercept), and `fl_mef_auto` (bead autofluorescence). The last term is constrained to be greater or equal to zero.

The bead fluorescence model is fit in log space using nonlinear least squares regression. In our experience, fitting in log space weights the residuals more evenly, whereas fitting in linear space vastly overvalues the brighter beads.

A standard curve is constructed by solving for `fl_mef`. As cell samples may not have the same autofluorescence as beads, the bead autofluorescence term (`fl_mef_auto`) is omitted from the standard curve; the user is expected to use an appropriate white cell sample to account for cellular autofluorescence if necessary. The returned standard curve mapping fluorescence in RFI units to MEF units is thus of the following form:

```
fl_mef = exp(m*log(fl_rfi) + b)
```

This is equivalent to:

```
fl_mef = exp(b) * (fl_rfi**m)
```

This works for positive `fl_rfi` values, but it is undefined for `fl_rfi < 0` and non-integer `m` (general case).

To extend this standard curve to negative values of `fl_rfi`, we define `s(fl_rfi)` to be equal to the standard curve above when `fl_rfi >= 0`. Next, we require this function to be odd, that is, `s(fl_rfi) = - s(-fl_rfi)`. This extends the domain to negative `fl_rfi` values and results in `s(fl_rfi) < 0` for any negative `fl_rfi`. Finally, we make `fl_mef = s(fl_rfi)` our new standard curve. In this way,:

```
s(fl_rfi) =   exp(b) * (  fl_rfi **m),     fl_rfi >= 0
            - exp(b) * ((-fl_rfi)**m),     fl_rfi <  0
```

This satisfies the definition of an odd function. In addition, `s(0) = 0`, and `s(fl_rfi)` converges to zero when `fl_rfi -> 0` from both sides. Therefore, the function is continuous at `fl_rfi = 0`. The definition of `s(fl_rfi)` can be expressed more conveniently as:

```
s(fl_rfi) = sign(fl_rfi) * exp(b) * (abs(fl_rfi)**m)
```

This is the equation implemented.

`FlowCal.mef.`**`get_transform_fxn`**(*data_beads*, *mef_values*, *mef_channels*, *clustering_fxn=<function clustering_gmm>*, *clustering_params={}*, *clustering_channels=None*, *statistic_fxn=<function median>*, *statistic_params={}*, *selection_fxn=<function selection_std>*, *selection_params={}*, *fitting_fxn=<function fit_beads_autofluorescence>*, *fitting_params={}*, *verbose=False*, *plot=False*, *plot_dir=None*, *plot_filename=None*, *full_output=False*)

Get a transformation function to convert flow cytometry data to MEF.

Parameters

data_beads [FCSData object] Flow cytometry data describing calibration beads.

mef_values [sequence of sequences] Known MEF values for the calibration bead subpopulations, for each channel specified in *mef_channels*. The innermost sequences must have the same length (the same number of bead subpopulations must exist for each channel). Values of np.nan or None specify that a subpopulation should be omitted from the fitting procedure.

mef_channels [int, or str, or list of int, or list of str] Channels for which to generate transformation functions.

verbose [bool, optional] Flag specifying whether to print information about step completion and warnings.

plot [bool, optional] Flag specifying whether to produce diagnostic plots.

plot_dir [str, optional] Directory where to save diagnostics plots. Ignored if *plot* is False. If `plot==True` and `plot_dir is None`, plot without saving.

plot_filename [str, optional] Name to use for plot files. If None, use `str(data_beads)`.

full_output [bool, optional] Flag specifying whether to include intermediate results in the output. If *full_output* is True, the function returns a *MEFOutput* `namedtuple` with fields as described below. If *full_output* is False, the function only returns the calculated transformation function.

Returns

transform_fxn [function] Transformation function to convert flow cytometry data from RFI units to MEF. This function has the following signature:

```
data_mef = transform_fxn(data_rfi, channels)
```

mef_channels [int, or str, or list, only if `full_output==True`] Channels on which the transformation function has been generated. Directly copied from the *mef_channels* argument.

clustering [dict, only if `full_output==True`] Results of the clustering step. The structure of this dictionary is:

```
clustering = {"labels": np.array}
```

A description of each `"key": value` is given below.

"labels" [array] Array of length `N`, where `N` is the number of events in *data_beads*. This array contains labels indicating which subpopulation each event has been assigned to by the clustering algorithm. Labels range from `0` to `M - 1`, where `M` is the number of MEF values specified, and therefore the number of subpopulations identified by the clustering algorithm.

statistic [dict, only if `full_output==True`] Results of the calculation of bead subpopulations' fluorescence. The structure of this dictionary is:

```
statistic = {"values": [np.array, ...]}
```

A description of each `"key": value` is given below.

"values" [list of arrays] Each array contains the representative fluorescence values of all subpopulations, for a specific fluorescence channel from *mef_channels*. Therefore, each array has a length equal to the number of subpopulations, and the outer list has as many arrays as the number of channels in *mef_channels*.

selection [dict, only if `full_output==True`] Results of the subpopulation selection step. The structure of this dictionary is:

```
selection = {"rfi": [np.array, ...],
             "mef": [np.array, ...]}
```

A description of each `"key": value` is given below.

"rfi" [list of arrays] Each array contains the fluorescence values of each selected subpopulation in RFI units, for a specific fluorescence channel from *mef_channels*. The outer list has as many arrays as the number of channels in *mef_channels*. Because the selection step may discard subpopulations, each array has a length less than or equal to the total number of subpopulations. Furthermore, different arrays in this list may not have the same length. However, the length of each array is consistent with the corresponding array in `selection["mef"]` (see below).

"mef" [list of arrays] Each array contains the fluorescence values of each selected subpopulation in MEF units, for a specific fluorescence channel from *mef_channels*. The outer list has as many arrays as the number of channels in *mef_channels*. Because the selection step may discard subpopulations, each array has a length less than or equal to the total number of subpopulations. Furthermore, different arrays in this list may not have the same length. However, the length of each array is consistent with the corresponding array in `selection["rfi"]` (see above).

fitting [dict, only if `full_output==True`] Results of the model fitting step. The structure of this dictionary is:

```
selection = {"std_crv": [func, ...],
             "beads_model": [func, ...],
             "beads_params": [np.array, ...],
             "beads_model_str": [str, ...],
             "beads_params_names": [[], ...]}
```

A description of each `"key": value` is given below.

"std_crv" [list of functions] Functions encoding the fitted standard curves, for each channel in *mef_channels*. Each element of this list is the `std_crv` output of the fitting function (see required signature of the `fitting_fxn` optional parameter), after applying it to the MEF and RFI fluorescence values of a specific channel from *mef_channels* .

"beads_model" [list of functions] Functions encoding the fluorescence model of the calibration beads, for each channel in *mef_channels*. Each element of this list is the `beads_model` output of the fitting function (see required signature of the `fitting_fxn` optional parameter), after applying it to the MEF and RFI fluorescence values of a specific channel from *mef_channels* .

"beads_params" [list of arrays] Fitted parameter values of the bead fluorescence model, for each channel in *mef_chanels*. Each element of this list is the `beads_params` output of the fitting function (see required signature of the `fitting_fxn` optional parameter), after applying it to the MEF and RFI fluorescence values of a specific channel from *mef_channels*.

"beads_model_str" [list of str] String representation of the bead models used, for each channel in *mef_channels*. Each element of this list is the `beads_model_str` output of the fitting function (see required signature of the `fitting_fxn` optional parameter), after applying it to the MEF and RFI fluorescence values of a specific channel from *mef_channels* .

"beads_params_names" [list of list] Names of the parameters given in *beads_params*, for each channel in *mef_channels*. Each element of this list is the `beads_params_names` output of the fitting function (see required signature of the `fitting_fxn` optional parameter), after applying it to the MEF and RFI fluorescence values of a specific channel from *mef_channels* .

Other Parameters

clustering_fxn [function, optional] Function used for clustering, or identification of subpopulations. Must have the following signature:

```
labels = clustering_fxn(data, n_clusters, **clustering_params)
```

where *data* is a NxD FCSData object or numpy array, *n_clusters* is the expected number of bead subpopulations, and *labels* is a 1D numpy array of length N, assigning each event in *data* to one subpopulation.

clustering_params [dict, optional] Additional keyword parameters to pass to *clustering_fxn*.

clustering_channels [list, optional] Channels used for clustering. If not specified, use *mef_channels*. If more than three channels are specified and *plot* is True, only a 3D scatter plot will be produced using the first three channels.

statistic_fxn [function, optional] Function used to calculate the representative fluorescence of each subpopulation. Must have the following signature:

```
s = statistic_fxn(data, **statistic_params)
```

where *data* is a 1D FCSData object or numpy array, and *s* is a float. Statistical functions from numpy, scipy, or FlowCal.stats are valid options.

statistic_params [dict, optional] Additional keyword parameters to pass to *statistic_fxn*.

selection_fxn [function, optional] Function to use for bead population selection. Must have the following signature:

```
selected_mask = selection_fxn(data_list, **selection_params)
```

where *data_list* is a list of FCSData objects, each one containing the events of one population, and *selected_mask* is a boolean array indicating whether the population has been selected (True) or discarded (False). If None, don't use a population selection procedure.

selection_params [dict, optional] Additional keyword parameters to pass to *selection_fxn*.

fitting_fxn [function, optional] Function used to fit the beads fluorescence model and obtain a standard curve. Must have the following signature:

```
std_crv, beads_model, beads_params, \
beads_model_str, beads_params_names = fitting_fxn(
    fl_rfi, fl_mef, **fitting_params)
```

where *std_crv* is a function implementing the standard curve, *beads_model* is a function implementing the beads fluorescence model, *beads_params* is an array containing the fitted parameters of the beads model, *beads_model_str* is a string representation of the beads model used, *beads_params_names* is a list with the parameter names in the same order as they are given in *beads_params*, and *fl_rfi* and *fl_mef* are the fluorescence values of the beads in RFI units and MEF units, respectively. Note that the standard curve and the fitted beads model are not necessarily the same.

fitting_params [dict, optional] Additional keyword parameters to pass to *fitting_fxn*.

Notes

The steps involved in generating the MEF transformation function are:

1. The individual subpopulations of beads are first identified using a clustering method of choice. Clustering is performed in all specified channels simultaneously.
2. The fluorescence of each subpopulation is calculated, for each channel in *mef_channels*.
3. Some subpopulations are then discarded if they are close to either the minimum or the maximum channel range limits. In addition, if the MEF value of some subpopulation is unknown (represented as a `np.nan` in *mef_values*), the whole subpopulation is also discarded.
4. The measured fluorescence of each subpopulation is compared with the known MEF values in *mef_values*, and a standard curve function is generated using the appropriate MEF model.

At the end, a transformation function is generated using the calculated standard curves, *mef_channels*, and `FlowCal.transform.to_mef()`.

Note that applying the resulting transformation function to other flow cytometry samples only yields correct results if they have been taken at the same settings as the calibration beads, for all channels in *mef_channels*.

Examples

Here is a simple application of this function:

```
>>> transform_fxn = FlowCal.mef.get_transform_fxn(
...     beads_data,
...     mef_channels=['FL1', 'FL3'],
...     mef_values=[np.array([    0,    646,    1704,    4827,
...                           15991, 47609, 135896, 273006],
...                 np.array([    0,   1614,    4035,    12025,
...                           31896, 95682, 353225, 1077421]],
...     )
>>> sample_mef = transform_fxn(data=sample_rfi,
...                            channels=['FL1', 'FL3'])
```

Here, we first generate `transform_fxn` from flow cytometry data contained in `FCSData` object `beads_data`, for channels FL1 and FL3, using provided MEF values for each one of these channels. In the next line, we use the resulting transformation function to transform cell sample data in RFI to MEF.

More data about intermediate steps can be obtained with the option `full_output=True`:

```
>>> get_transform_output = FlowCal.mef.get_transform_fxn(
...     beads_data,
...     mef_channels=['FL1', 'FL3'],
...     mef_values=[np.array([    0,    646,    1704,    4827,
...                           15991, 47609, 135896, 273006],
...                 np.array([    0,   1614,    4035,    12025,
...                           31896, 95682, 353225, 1077421]],
...     full_output=True)
```

In this case, the output `get_transform_output` will be a *MEFOutput* `namedtuple` similar to the following:

```
FlowCal.mef.MEFOutput(
    transform_fxn=<functools.partial object>,
    mef_channels=['FL1', 'FL3'],
```

(continues on next page)

(continued from previous page)

```
    clustering={
        'labels' : [7, 2, 2, ... 4, 3, 5]
    },
    statistic={
        'values' : [np.array([ 101,  150,  231,  433,
                             1241, 3106, 7774, 9306]),
                    np.array([   3,   30,   71,  204,
                              704, 2054, 6732, 9912])]
    },
    selection={
        'rfi' : [np.array([  101,    150,    231,    433,
                            1241,   3106,   7774]),
                 np.array([  30,      71,    204,    704,
                           2054,    6732])]
        'mef' : [np.array([     0,    646,   1704,   4827,
                           15991,  47609, 135896]),
                 np.array([ 1614,   4035,  12025,  31896,
                           95682, 353225])]
    },
    fitting={
        'std_crv' : [<function <lambda>>,
                     <function <lambda>>]
        'beads_model' : [<function <lambda>>,
                         <function <lambda>>]
        'beads_params' : [np.array([ 1.09e0, 2.02e0, 1.15e3]),
                          np.array([9.66e-1, 4.17e0, 6.63e1])]
        'beads_model_str' : ['m*log(fl_rfi) + b = log(fl_mef_auto + fl_mef)',
                             'm*log(fl_rfi) + b = log(fl_mef_auto + fl_mef)']
        'beads_params_names' : [['m', 'b', 'fl_mef_auto],
                                ['m', 'b', 'fl_mef_auto]]
    },
)
```

`FlowCal.mef.`**`plot_standard_curve`**(*fl_rfi*, *fl_mef*, *beads_model*, *std_crv*, *xscale='linear'*, *yscale='linear'*, *xlim=None*, *ylim=(1.0, 100000000.0)*)

Plot a standard curve with fluorescence of calibration beads.

Parameters

fl_rfi [array_like] Fluorescence of the calibration beads' subpopulations, in RFI units.

fl_mef [array_like] Fluorescence of the calibration beads' subpopulations, in MEF units.

beads_model [function] Fluorescence model of the calibration beads.

std_crv [function] The standard curve, mapping relative fluorescence (RFI) units to MEF units.

Other Parameters

xscale [str, optional] Scale of the x axis, either `linear` or `log`.

yscale [str, optional] Scale of the y axis, either `linear` or `log`.

xlim [tuple, optional] Limits for the x axis.

ylim [tuple, optional] Limits for the y axis.

`FlowCal.mef.`**`selection_std`**(*populations*, *low=None*, *high=None*, *n_std_low=2.5*, *n_std_high=2.5*, *scale='logicle'*)

Select populations if most of their elements are between two values.

This function selects populations from *populations* if their means are more than *n_std_low* standard deviations greater than *low* and *n_std_high* standard deviations lower than *high*.

Optionally, all elements in *populations* can be rescaled as specified by the *scale* argument before calculating means and standard deviations.

Parameters

populations [list of 1D arrays or 1-channel FCSData objects] Populations to select or discard.

low, high [int or float] Low and high thresholds. Required if the elements in *populations* are numpy arrays. If not specified, and the elements in *populations* are FCSData objects, use 1.5% and 98.5% of the range in `populations[0].range`.

n_std_low, n_std_high [float, optional] Number of standard deviations from *low* and *high*, respectively, that a population's mean has to be closer than to be discarded.

scale [str, optional] Rescaling applied to *populations* before calculating means and standard deviations. Can be either `linear` (no rescaling), `log`, or `logicle`.

Returns

selected_mask [boolean array] Flags indicating whether a population has been selected.

2.5.5 FlowCal.plot module

Functions for visualizing flow cytometry data.

Functions in this module are divided in two categories:

- Simple Plot Functions, with a signature similar to the following:

```
plot_fxn(data_list, channels, parameters, savefig)
```

  where *data_list* is a NxD FCSData object or numpy array, or a list of such, *channels* specifies the channel or channels to use for the plot, *parameters* are function-specific parameters, and *savefig* indicates whether to save the figure to an image file. Note that *hist1d*, *violin*, and *violin_dose_response* use *channel* instead of *channels*, since they use a single channel, and *density2d* only accepts one FCSData object or numpy array as its first argument.

  Simple Plot Functions do not create a new figure or axis, so they can be called directly to plot in a previously created axis if desired. If *savefig* is not specified, the plot is maintained in the current axis when the function returns. This allows for further modifications to the axis by direct calls to, for example, `plt.xlabel`, `plt.title`, etc. However, if *savefig* is specified, the figure is closed after being saved. In this case, the function may include keyword parameters *xlabel*, *ylabel*, *xlim*, *ylim*, *title*, and others related to legend or color, which allow the user to modify the axis prior to saving.

  The following functions in this module are Simple Plot Functions:

  - `hist1d`
  - `violin`
  - `violin_dose_response`
  - `density2d`
  - `scatter2d`
  - `scatter3d`

- Complex Plot Functions, which create a figure with several axes, and use one or more Simple Plot functions to populate the axes. They always include a *savefig* argument, which indicates whether to save the figure to a file. If *savefig* is not specified, the plot is maintained in the newly created figure when the function returns. However, if *savefig* is specified, the figure is closed after being saved.

  The following functions in this module are Complex Plot Functions:

  - `density_and_hist`
  - `scatter3d_and_projections`

`FlowCal.plot.`**`density2d`**`(`*data, channels=[0, 1], bins=1024, mode='mesh', normed=False, smooth=True, sigma=10.0, colorbar=False, xscale='logicle', yscale='logicle', xlabel=None, ylabel=None, xlim=None, ylim=None, title=None, savefig=None, **kwargs*`)`

Plot a 2D density plot from two channels of a flow cytometry data set.

density2d has two plotting modes which are selected using the *mode* argument. With `mode=='mesh'`, this function plots the data as a true 2D histogram, in which a plane is divided into bins and the color of each bin is directly related to the number of elements therein. With `mode=='scatter'`, this function also calculates a 2D histogram, but it plots a 2D scatter plot in which each dot corresponds to a bin, colored according to the number elements therein. The most important difference is that the `scatter` mode does not color regions corresponding to empty bins. This allows for easy identification of regions with low number of events. For both modes, the calculated histogram can be smoothed using a Gaussian kernel by specifying `smooth=True`. The width of the kernel is, in this case, given by *sigma*.

Parameters

data [FCSData or numpy array] Flow cytometry data to plot.

channels [list of int, list of str, optional] Two channels to use for the plot.

bins [int or array_like or [int, int] or [array, array], optional] Bins used for plotting:

- If None, use `data.hist_bins` to obtain bin edges for both axes. None is not allowed if `data.hist_bins` is not available.
- If int, *bins* specifies the number of bins to use for both axes. If `data.hist_bins` exists, it will be used to generate a number *bins* of bins.
- If array_like, *bins* directly specifies the bin edges to use for both axes.
- If [int, int], each element of *bins* specifies the number of bins for each axis. If `data.hist_bins` exists, use it to generate `bins[0]` and `bins[1]` bin edges, respectively.
- If [array, array], each element of *bins* directly specifies the bin edges to use for each axis.
- Any combination of the above, such as [int, array], [None, int], or [array, int]. In this case, None indicates to generate bin edges using `data.hist_bins` as above, int indicates the number of bins to generate, and an array directly indicates the bin edges. Note that None is not allowed if `data.hist_bins` does not exist.

mode [{'mesh', 'scatter'}, str, optional] Plotting mode. 'mesh' produces a 2D-histogram whereas 'scatter' produces a scatterplot colored by histogram bin value.

normed [bool, optional] Flag indicating whether to plot a normed histogram (probability mass function instead of a counts-based histogram).

smooth [bool, optional] Flag indicating whether to apply Gaussian smoothing to the histogram.

colorbar [bool, optional] Flag indicating whether to add a colorbar to the plot.

savefig [str, optional] The name of the file to save the figure to. If None, do not save.

Other Parameters

sigma [float, optional] The sigma parameter for the Gaussian kernel to use when smoothing.

xscale [str, optional] Scale of the x axis, either `linear`, `log`, or `logicle`.

yscale [str, optional] Scale of the y axis, either `linear`, `log`, or `logicle`

xlabel [str, optional] Label to use on the x axis. If None, attempts to extract channel name from *data*.

ylabel [str, optional] Label to use on the y axis. If None, attempts to extract channel name from *data*.

xlim [tuple, optional] Limits for the x axis. If not specified and *bins* exists, use the lowest and highest values of *bins*.

ylim [tuple, optional] Limits for the y axis. If not specified and *bins* exists, use the lowest and highest values of *bins*.

title [str, optional] Plot title.

kwargs [dict, optional] Additional parameters passed directly to the underlying matplotlib functions: `plt.scatter` if `mode==scatter`, and `plt.pcolormesh` if `mode==mesh`.

`FlowCal.plot.`**`density_and_hist`**(*data*, *gated_data=None*, *gate_contour=None*, *density_channels=None*, *density_params={}*, *hist_channels=None*, *hist_params={}*, *figsize=None*, *savefig=None*)

Make a combined density/histogram plot of a FCSData object.

This function calls *hist1d* and *density2d* to plot a density diagram and a number of histograms in different subplots of the same plot using one single function call. Setting *density_channels* to None will not produce a density diagram, and setting *hist_channels* to None will not produce any histograms. Setting both to None will raise an error. Additional parameters can be provided to *density2d* and *hist1d* by using *density_params* and *hist_params*.

If *gated_data* is provided, this function will plot the histograms corresponding to *gated_data* on top of *data*'s histograms, with some transparency on *data*. In addition, a legend will be added with the labels 'Ungated' and 'Gated'. If *gate_contour* is provided and it contains a valid list of 2D curves, these will be plotted on top of the density plot.

Parameters

data [FCSData object] Flow cytometry data object to plot.

gated_data [FCSData object, optional] Flow cytometry data object. If *gated_data* is specified, the histograms of *data* are plotted with an alpha value of 0.5, and the histograms of *gated_data* are plotted on top of those with an alpha value of 1.0.

gate_contour [list, optional] List of Nx2 curves, representing a gate contour to be plotted in the density diagram.

density_channels [list] Two channels to use for the density plot. If *density_channels* is None, do not plot a density plot.

density_params [dict, optional] Parameters to pass to *density2d*.

hist_channels [list] Channels to use for each histogram. If *hist_channels* is None, do not plot histograms.

hist_params [list, optional] List of dictionaries with the parameters to pass to each call of *hist1d*.

savefig [str, optional] The name of the file to save the figure to. If None, do not save.

Other Parameters

figsize [tuple, optional] Figure size. If None, calculate a default based on the number of subplots.

Raises

ValueError If both *density_channels* and *hist_channels* are None.

`FlowCal.plot.`**`hist1d`**(*data_list*, *channel=0*, *xscale='logicle'*, *bins=256*, *histtype='stepfilled'*, *normed_area=False*, *normed_height=False*, *xlabel=None*, *ylabel=None*, *xlim=None*, *ylim=None*, *title=None*, *legend=False*, *legend_loc='best'*, *legend_fontsize='medium'*, *legend_labels=None*, *facecolor=None*, *edgecolor=None*, *savefig=None*, ***kwargs*)

Plot one 1D histogram from one or more flow cytometry data sets.

Parameters

data_list [FCSData or numpy array or list of FCSData or numpy array] Flow cytometry data to plot.

channel [int or str, optional] Channel from where to take the events to plot. If ndim == 1, channel is ignored. String channel specifications are only supported for data types which support string-based indexing (e.g. FCSData).

xscale [str, optional] Scale of the x axis, either `linear`, `log`, or `logicle`.

bins [int or array_like, optional] If *bins* is an integer, it specifies the number of bins to use. If *bins* is an array, it specifies the bin edges to use. If *bins* is None or an integer, *hist1d* will attempt to use `data.hist_bins` to generate the bins automatically.

histtype [{'bar', 'barstacked', 'step', 'stepfilled'}, str, optional] Histogram type. Directly passed to `plt.hist`.

normed_area [bool, optional] Flag indicating whether to normalize the histogram such that the area under the curve is equal to one. The resulting plot is equivalent to a probability density function.

normed_height [bool, optional] Flag indicating whether to normalize the histogram such that the sum of all bins' heights is equal to one. The resulting plot is equivalent to a probability mass function. *normed_height* is ignored if *normed_area* is True.

savefig [str, optional] The name of the file to save the figure to. If None, do not save.

Other Parameters

xlabel [str, optional] Label to use on the x axis. If None, attempts to extract channel name from last data object.

ylabel [str, optional] Label to use on the y axis. If None and `normed_area==True`, use 'Probability'. If None, `normed_area==False`, and `normed_height==True`, use 'Counts (normalized)'. If None, `normed_area==False`, and `normed_height==False`, use 'Counts'.

xlim [tuple, optional] Limits for the x axis. If not specified and *bins* exists, use the lowest and highest values of *bins*.

ylim [tuple, optional] Limits for the y axis.

title [str, optional] Plot title.

legend [bool, optional] Flag specifying whether to include a legend. If *legend* is True, the legend labels will be taken from *legend_labels* if present, else they will be taken from `str(data_list[i])`.

legend_loc [str, optional] Location of the legend.

legend_fontsize [int or str, optional] Font size for the legend.

legend_labels [list, optional] Labels to use for the legend.

facecolor [matplotlib color or list of matplotlib colors, optional] The histogram's facecolor. It can be a list with the same length as *data_list*. If *edgecolor* and *facecolor* are not specified, and `histtype == 'stepfilled'`, the facecolor will be taken from the module-level variable *cmap_default*.

edgecolor [matplotlib color or list of matplotlib colors, optional] The histogram's edgecolor. It can be a list with the same length as *data_list*. If *edgecolor* and *facecolor* are not specified, and `histtype == 'step'`, the edgecolor will be taken from the module-level variable *cmap_default*.

kwargs [dict, optional] Additional parameters passed directly to matploblib's `hist`.

Notes

hist1d calls matplotlib's `hist` function for each object in *data_list*. *hist_type*, the type of histogram to draw, is directly passed to `plt.hist`. Additional keyword arguments provided to *hist1d* are passed directly to `plt.hist`.

If *normed_area* is set to True, *hist1d* calls `plt.hist` with `density` (or `normed`, if matplotlib's version is older than 2.2.0) set to True. There is a bug in matplotlib 2.1.0 that produces an incorrect plot in these conditions. We do not recommend using matplotlib 2.1.0 if *normed_area* is expected to be used.

`FlowCal.plot.`**`scatter2d`**(*data_list, channels=[0, 1], xscale='logicle', yscale='logicle', xlabel=None, ylabel=None, xlim=None, ylim=None, title=None, color=None, savefig=None, **kwargs*)

Plot 2D scatter plot from one or more FCSData objects or numpy arrays.

Parameters

data_list [array or FCSData or list of array or list of FCSData] Flow cytometry data to plot.

channels [list of int, list of str] Two channels to use for the plot.

savefig [str, optional] The name of the file to save the figure to. If None, do not save.

Other Parameters

xscale [str, optional] Scale of the x axis, either `linear`, `log`, or `logicle`.

yscale [str, optional] Scale of the y axis, either `linear`, `log`, or `logicle`.

xlabel [str, optional] Label to use on the x axis. If None, attempts to extract channel name from last data object.

ylabel [str, optional] Label to use on the y axis. If None, attempts to extract channel name from last data object.

xlim [tuple, optional] Limits for the x axis. If None, attempts to extract limits from the range of the last data object.

ylim [tuple, optional] Limits for the y axis. If None, attempts to extract limits from the range of the last data object.

title [str, optional] Plot title.

color [matplotlib color or list of matplotlib colors, optional] Color for the scatter plot. It can be a list with the same length as *data_list*. If *color* is not specified, elements from *data_list* are plotted with colors taken from the module-level variable *cmap_default*.

kwargs [dict, optional] Additional parameters passed directly to matploblib's `scatter`.

Notes

scatter2d calls matplotlib's `scatter` function for each object in data_list. Additional keyword arguments provided to *scatter2d* are passed directly to `plt.scatter`.

`FlowCal.plot.`**`scatter3d`**`(`*data_list, channels=[0, 1, 2], xscale='logicle', yscale='logicle', zscale='logicle', xlabel=None, ylabel=None, zlabel=None, xlim=None, ylim=None, zlim=None, title=None, color=None, savefig=None, **kwargs*`)`

Plot 3D scatter plot from one or more FCSData objects or numpy arrays.

Parameters

data_list [array or FCSData or list of array or list of FCSData] Flow cytometry data to plot.

channels [list of int, list of str] Three channels to use for the plot.

savefig [str, optional] The name of the file to save the figure to. If None, do not save.

Other Parameters

xscale [str, optional] Scale of the x axis, either `linear`, `log`, or `logicle`.

yscale [str, optional] Scale of the y axis, either `linear`, `log`, or `logicle`.

zscale [str, optional] Scale of the z axis, either `linear`, `log`, or `logicle`.

xlabel [str, optional] Label to use on the x axis. If None, attempts to extract channel name from last data object.

ylabel [str, optional] Label to use on the y axis. If None, attempts to extract channel name from last data object.

zlabel [str, optional] Label to use on the z axis. If None, attempts to extract channel name from last data object.

xlim [tuple, optional] Limits for the x axis. If None, attempts to extract limits from the range of the last data object.

ylim [tuple, optional] Limits for the y axis. If None, attempts to extract limits from the range of the last data object.

zlim [tuple, optional] Limits for the z axis. If None, attempts to extract limits from the range of the last data object.

title [str, optional] Plot title.

color [matplotlib color or list of matplotlib colors, optional] Color for the scatter plot. It can be a list with the same length as *data_list*. If *color* is not specified, elements from *data_list* are plotted with colors taken from the module-level variable *cmap_default*.

kwargs [dict, optional] Additional parameters passed directly to matploblib's `scatter`.

Notes

scatter3d uses matplotlib's `scatter` with a 3D projection. Additional keyword arguments provided to *scatter3d* are passed directly to `scatter`.

`FlowCal.plot.`**`scatter3d_and_projections`**(*data_list, channels=[0, 1, 2], xscale='logicle', yscale='logicle', zscale='logicle', xlabel=None, ylabel=None, zlabel=None, xlim=None, ylim=None, zlim=None, color=None, figsize=None, savefig=None, **kwargs*)

Plot a 3D scatter plot and 2D projections from FCSData objects.

scatter3d_and_projections creates a 3D scatter plot and three 2D projected scatter plots in four different axes for each FCSData object in *data_list*, in the same figure.

Parameters

data_list [FCSData object, or list of FCSData objects] Flow cytometry data to plot.

channels [list of int, list of str] Three channels to use for the plot.

savefig [str, optional] The name of the file to save the figure to. If None, do not save.

Other Parameters

xscale [str, optional] Scale of the x axis, either `linear`, `log`, or `logicle`.

yscale [str, optional] Scale of the y axis, either `linear`, `log`, or `logicle`.

zscale [str, optional] Scale of the z axis, either `linear`, `log`, or `logicle`.

xlabel [str, optional] Label to use on the x axis. If None, attempts to extract channel name from last data object.

ylabel [str, optional] Label to use on the y axis. If None, attempts to extract channel name from last data object.

zlabel [str, optional] Label to use on the z axis. If None, attempts to extract channel name from last data object.

xlim [tuple, optional] Limits for the x axis. If None, attempts to extract limits from the range of the last data object.

ylim [tuple, optional] Limits for the y axis. If None, attempts to extract limits from the range of the last data object.

zlim [tuple, optional] Limits for the z axis. If None, attempts to extract limits from the range of the last data object.

color [matplotlib color or list of matplotlib colors, optional] Color for the scatter plot. It can be a list with the same length as *data_list*. If *color* is not specified, elements from *data_list* are plotted with colors taken from the module-level variable *cmap_default*.

figsize [tuple, optional] Figure size. If None, use matplotlib's default.

kwargs [dict, optional] Additional parameters passed directly to matploblib's `scatter`.

Notes

scatter3d_and_projections uses matplotlib's `scatter`, with the 3D scatter plot using a 3D projection. Additional keyword arguments provided to *scatter3d_and_projections* are passed directly to `scatter`.

`FlowCal.plot.violin`(*data*, *channel=None*, *positions=None*, *violin_width=None*, *xscale=None*, *yscale=None*, *xlim=None*, *ylim=None*, *vert=True*, *num_bins=100*, *bin_edges=None*, *density=False*, *upper_trim_fraction=0.01*, *lower_trim_fraction=0.01*, *violin_width_to_span_fraction=0.1*, *violin_kwargs=None*, *draw_summary_stat=True*, *draw_summary_stat_fxn=<function mean>*, *draw_summary_stat_kwargs=None*, *log_zero_tick_label=None*, *draw_log_zero_divider=True*, *draw_log_zero_divider_kwargs=None*, *xlabel=None*, *ylabel=None*, *title=None*, *savefig=None*)

Plot violin plot.

Illustrate the relative frequency of members of different populations using normalized, symmetrical histograms ("violins") centered at corresponding positions. Wider regions of violins indicate regions that occur with greater frequency.

Parameters

data [1D or ND array or list of 1D or ND arrays] A population or collection of populations for which to plot violins. If ND arrays are used (e.g., FCSData), *channel* must be specified.

channel [int or str, optional] Channel from *data* to plot. If specified, data are assumed to be ND arrays. String channel specifications are only supported for data types that support string-based indexing (e.g., FCSData).

positions [scalar or array, optional] Positions at which to center violins.

violin_width [scalar, optional] Width of violin. If the scale of the position axis (*xscale* if *vert* is True, *yscale* if *vert* is False) is `log`, the units are decades. If not specified, *violin_width* is calculated from the limits of the position axis (*xlim* if *vert* is True, *ylim* if *vert* is False) and *violin_width_to_span_fraction*. If only one violin is specified in *data*, *violin_width* = 0.5.

savefig [str, optional] The name of the file to save the figure to. If None, do not save.

Other Parameters

xscale [{'linear', 'log', 'logicle'}, optional] Scale of the x-axis. `logicle` is only supported for horizontal violin plots (i.e., when *vert* is False). Default is `linear` if *vert* is True, `logicle` if *vert* is False.

yscale [{'logicle', 'linear', 'log'}, optional] Scale of the y-axis. If *vert* is False, `logicle` is not supported. Default is `logicle` if *vert* is True, `linear` if *vert* is False.

xlim, ylim [tuple, optional] Limits of the x-axis and y-axis views. If not specified, the view of the position axis (*xlim* if *vert* is True, *ylim* if *vert* if False) is calculated to pad the extreme violins with 0.5 * *violin_width*. If *violin_width* is also not specified, *violin_width* is calculated to satisfy the 0.5 * *violin_width* padding and *violin_width_to_span_fraction*. If not specified, the view of the data axis (*ylim* if *vert* is True, *xlim* if *vert* is False) is calculated to span all violins (before they are aesthetically trimmed).

vert [bool, optional] Flag specifying to illustrate a vertical violin plot. If False, a horizontal violin plot is illustrated.

num_bins [int, optional] Number of bins to bin population members. Ignored if *bin_edges* is specified.

bin_edges [array or list of arrays, optional] Bin edges used to bin population members. Bin edges can be specified for individual violins using a list of arrays of the same length as *data*. If not specified, *bin_edges* is calculated to span the data axis (*ylim* if *vert* is True, *xlim* if *vert* is False) logicly, linearly, or logarithmically (based on the scale of the data axis; *yscale* if *vert* is True, *xscale* if *vert* is False) using *num_bins*.

density [bool, optional] *density* parameter passed to the `np.histogram()` command that bins population members for each violin. If True, violin width represents relative frequency *density* instead of relative frequency (i.e., bins are normalized by their width).

upper_trim_fraction [float or list of floats, optional] Fraction of members to trim (discard) from the top of the violin (e.g., for aesthetic purposes). Upper trim fractions can be specified for individual violins using a list of floats of the same length as *data*.

lower_trim_fraction [float or list of floats, optional] Fraction of members to trim (discard) from the bottom of the violin (e.g., for aesthetic purposes). Lower trim fractions can be specified for individual violins using a list of floats of the same length as *data*.

violin_width_to_span_fraction [float, optional] Fraction of the position axis span (*xlim* if *vert* is True, *ylim* if *vert* is False) that a violin should span. Ignored if *violin_width* is specified.

violin_kwargs [dict or list of dicts, optional] Keyword arguments passed to the `plt.fill_between()` command that illustrates each violin. Keyword arguments can be specified for individual violins using a list of dicts of the same length as *data*. Default = {'facecolor':'gray', 'edgecolor':'black'}.

draw_summary_stat [bool, optional] Flag specifying to illustrate a summary statistic for each violin.

draw_summary_stat_fxn [function, optional] Function used to calculate the summary statistic for each violin. Summary statistics are calculated prior to aesthetic trimming.

draw_summary_stat_kwargs [dict or list of dicts, optional] Keyword arguments passed to the `plt.plot()` command that illustrates each violin's summary statistic. Keyword arguments can be specified for individual violins using a list of dicts of the same length as *data*. Default = {'color':'black'}.

log_zero_tick_label [str, optional] Label of position=0 violin tick if the position axis scale (*xscale* if *vert* is True, *yscale* if *vert* is False) is `log`. Default is generated by the default log tick formatter (`matplotlib.ticker.LogFormatterSciNotation`) with x=0.

draw_log_zero_divider [bool, optional] Flag specifying to illustrate a line separating the position=0 violin from the other violins if the position axis scale (*xscale* if *vert* is True, *yscale* if *vert* is False) is `log`.

draw_log_zero_divider_kwargs [dict, optional] Keyword arguments passed to the `plt.axvline()` or `plt.axhline()` command that illustrates the position=0 violin divider. Default = {'color':'gray','linestyle':':'}.

xlabel, ylabel [str, optional] Labels to use on the x and y axes. If a label for the data axis is not specified (*ylabel* if *vert* is True, *xlabel* if *vert* is False), the channel name will be used if possible (extracted from the last data object).

title [str, optional] Plot title.

`FlowCal.plot.`**`violin_dose_response`**(*data*, *channel=None*, *positions=None*, *min_data=None*, *max_data=None*, *violin_width=None*, *model_fxn=None*, *xscale='linear'*, *yscale='logicle'*, *xlim=None*, *ylim=None*, *violin_width_to_span_fraction=0.1*, *num_bins=100*, *bin_edges=None*, *density=False*, *upper_trim_fraction=0.01*, *lower_trim_fraction=0.01*, *violin_kwargs=None*, *draw_summary_stat=True*, *draw_summary_stat_fxn=<function mean>*, *draw_summary_stat_kwargs=None*, *log_zero_tick_label=None*, *min_bin_edges=None*, *min_upper_trim_fraction=0.01*, *min_lower_trim_fraction=0.01*, *min_violin_kwargs=None*, *min_draw_summary_stat_kwargs=None*, *draw_min_line=True*, *draw_min_line_kwargs=None*, *min_tick_label='Min'*, *max_bin_edges=None*, *max_upper_trim_fraction=0.01*, *max_lower_trim_fraction=0.01*, *max_violin_kwargs=None*, *max_draw_summary_stat_kwargs=None*, *draw_max_line=True*, *draw_max_line_kwargs=None*, *max_tick_label='Max'*, *draw_model_kwargs=None*, *draw_log_zero_divider=True*, *draw_log_zero_divider_kwargs=None*, *draw_minmax_divider=True*, *draw_minmax_divider_kwargs=None*, *xlabel=None*, *ylabel=None*, *title=None*, *savefig=None*)

Plot violin plot with min data, max data, and mathematical model.

Plot a violin plot (see `FlowCal.plot.violin()` description) with vertical violins and separately illustrate a min violin, a max violin, and a mathematical model. Useful for illustrating "dose response" or "transfer" functions, which benefit from the added context of minimum and maximum bounds and which are often described by mathematical models. Min and max violins are illustrated to the left of the plot, and the mathematical model is correctly illustrated even when a position=0 violin is illustrated separately when *xscale* is `log`.

Parameters

data [1D or ND array or list of 1D or ND arrays] A population or collection of populations for which to plot violins. If ND arrays are used (e.g., FCSData), *channel* must be specified.

channel [int or str, optional] Channel from *data* to plot. If specified, data are assumed to be ND arrays. String channel specifications are only supported for data types that support string-based indexing (e.g., FCSData).

positions [scalar or array, optional] Positions at which to center violins.

min_data [1D or ND array, optional] A population representing a minimum control. This violin is separately illustrated at the left of the plot.

max_data [1D or ND array, optional] A population representing a maximum control. This violin is separately illustrated at the left of the plot.

violin_width [scalar, optional] Width of violin. If *xscale* is `log`, the units are decades. If not specified, *violin_width* is calculated from *xlim* and *violin_width_to_span_fraction*. If only one violin is specified in *data*, *violin_width* = 0.5.

model_fxn [function, optional] Function used to calculate model y-values. 100 x-values are linearly (if *xscale* is `linear`) or logarithmically (if *xscale* is `log`) generated spanning *xlim*. If *xscale* is `log` and a position=0 violin is specified, the result of `model_fxn(0.0)` is illustrated as a horizontal line with the position=0 violin.

savefig [str, optional] The name of the file to save the figure to. If None, do not save.

Other Parameters

xscale [{'linear', 'log'}, optional] Scale of the x-axis.

yscale [{'logicle', 'linear', 'log'}, optional] Scale of the y-axis.

xlim [tuple, optional] Limits of the x-axis view. If not specified, *xlim* is calculated to pad leftmost and rightmost violins with 0.5 * *violin_width*. If *violin_width* is also not specified, *violin_width* is calculated to satisfy the 0.5 * *violin_width* padding and *violin_width_to_span_fraction*.

ylim [tuple, optional] Limits of the y-axis view. If not specified, *ylim* is calculated to span all violins (before they are aesthetically trimmed).

violin_width_to_span_fraction [float, optional] Fraction of the x-axis span that a violin should span. Ignored if *violin_width* is specified.

num_bins [int, optional] Number of bins to bin population members. Ignored if *bin_edges* is specified.

bin_edges [array or list of arrays, optional] Bin edges used to bin population members for *data* violins. Bin edges can be specified for individual violins using a list of arrays of the same length as *data*. If not specified, *bin_edges* is calculated to span *ylim* logicly (if *yscale* is `logicle`), linearly (if *yscale* is `linear`), or logarithmically (if *yscale* is `log`) using *num_bins*.

density [bool, optional] *density* parameter passed to the `np.histogram()` command that bins population members for each violin. If True, violin width represents relative frequency *density* instead of relative frequency (i.e., bins are normalized by their width).

upper_trim_fraction [float or list of floats, optional] Fraction of members to trim (discard) from the top of the *data* violins (e.g., for aesthetic purposes). Upper trim fractions can be specified for individual violins using a list of floats of the same length as *data*.

lower_trim_fraction [float or list of floats, optional] Fraction of members to trim (discard) from the bottom of the *data* violins (e.g., for aesthetic purposes). Lower trim fractions can be specified for individual violins using a list of floats of the same length as *data*.

violin_kwargs [dict or list of dicts, optional] Keyword arguments passed to the `plt.fill_betweenx()` command that illustrates the *data* violins. Keyword arguments can be specified for individual violins using a list of dicts of the same length as *data*. Default = {'facecolor':'gray', 'edgecolor':'black'}.

draw_summary_stat [bool, optional] Flag specifying to illustrate a summary statistic for each violin.

draw_summary_stat_fxn [function, optional] Function used to calculate the summary statistic for each violin. Summary statistics are calculated prior to aesthetic trimming.

draw_summary_stat_kwargs [dict or list of dicts, optional] Keyword arguments passed to the `plt.plot()` command that illustrates the *data* violin summary statistics. Keyword arguments can be specified for individual violins using a list of dicts of the same length as *data*. Default = {'color':'black'}.

log_zero_tick_label [str, optional] Label of position=0 violin tick if *xscale* is `log`. Default is generated by the default log tick formatter (`matplotlib.ticker.LogFormatterSciNotation`) with x=0.

min_bin_edges [array, optional] Bin edges used to bin population members for the min violin. If not specified, *min_bin_edges* is calculated to span *ylim* logicaly (if *yscale* is `logicle`), linearly (if *yscale* is `linear`), or logarithmically (if *yscale* is `log`) using *num_bins*.

min_upper_trim_fraction [float, optional] Fraction of members to trim (discard) from the top of the min violin.

min_lower_trim_fraction [float, optional] Fraction of members to trim (discard) from the bottom of the min violin.

min_violin_kwargs [dict, optional] Keyword arguments passed to the `plt.fill_betweenx()` command that illustrates the min violin. Default = {'facecolor':'black', 'edgecolor':'black'}.

min_draw_summary_stat_kwargs [dict, optional] Keyword arguments passed to the `plt.plot()` command that illustrates the min violin summary statistic. Default = {'color':'gray'}.

draw_min_line [bool, optional] Flag specifying to illustrate a line from the min violin summary statistic across the plot.

draw_min_line_kwargs [dict, optional] Keyword arguments passed to the `plt.plot()` command that illustrates the min violin line. Default = {'color':'gray', 'linestyle':'–', 'zorder':-2}.

min_tick_label [str, optional] Label of min violin tick. Default='Min'.

max_bin_edges [array, optional] Bin edges used to bin population members for the max violin. If not specified, *max_bin_edges* is calculated to span *ylim* logicaly (if *yscale* is `logicle`), linearly (if *yscale* is `linear`), or logarithmically (if *yscale* is `log`) using *num_bins*.

max_upper_trim_fraction [float, optional] Fraction of members to trim (discard) from the top of the max violin.

max_lower_trim_fraction [float, optional] Fraction of members to trim (discard) from the bottom of the max violin.

max_violin_kwargs [dict, optional] Keyword arguments passed to the `plt.fill_betweenx()` command that illustrates the max violin. Default = {'facecolor':'black', 'edgecolor':'black'}.

max_draw_summary_stat_kwargs [dict, optional] Keyword arguments passed to the `plt.plot()` command that illustrates the max violin summary statistic. Default = {'color':'gray'}.

draw_max_line [bool, optional] Flag specifying to illustrate a line from the max violin summary statistic across the plot.

draw_max_line_kwargs [dict, optional] Keyword arguments passed to the `plt.plot()` command that illustrates the max violin line. Default = {'color':'gray', 'linestyle':'–', 'zorder':-2}.

max_tick_label [str, optional] Label of max violin tick. Default='Max'.

draw_model_kwargs [dict, optional] Keyword arguments passed to the `plt.plot()` command that illustrates the model. Default = {'color':'gray', 'zorder':-1, 'solid_capstyle':'butt'}.

draw_log_zero_divider [bool, optional] Flag specifying to illustrate a line separating the position=0 violin from the *data* violins if *xscale* is `log`.

draw_log_zero_divider_kwargs [dict, optional] Keyword arguments passed to the `plt.axvline()` command that illustrates the position=0 violin divider. Default = {'color':'gray', 'linestyle':':'}.

draw_minmax_divider [bool, optional] Flag specifying to illustrate a vertical line separating the min and max violins from other violins.

draw_minmax_divider_kwargs [dict, optional] Keyword arguments passed to the `plt.axvline()` command that illustrates the min/max divider. Default = {'color':'gray', 'linestyle':'-'}.

xlabel [str, optional] Label to use on the x-axis.

ylabel [str, optional] Label to use on the y-axis. If None, channel name will be used if possible (extracted from the last data object).

title [str, optional] Plot title.

2.5.6 FlowCal.stats module

Functions to calculate statistics from the events in a FCSData object.

`FlowCal.stats.`**`cv`**`(`*data*, *channels=None*`)`

Calculate the Coeff. of Variation of the events in an FCSData object.

Parameters

data [FCSData or numpy array] NxD flow cytometry data where N is the number of events and D is the number of parameters (aka channels).

channels [int or str or list of int or list of str, optional] Channels on which to calculate the statistic. If None, use all channels.

Returns

float or numpy array The Coefficient of Variation of the events in the specified channels of *data*.

Notes

The Coefficient of Variation (CV) of a dataset is defined as the standard deviation divided by the mean of such dataset.

`FlowCal.stats.`**`gcv`**`(`*data*, *channels=None*`)`

Calculate the geometric CV of the events in an FCSData object.

Parameters

data [FCSData or numpy array] NxD flow cytometry data where N is the number of events and D is the number of parameters (aka channels).

channels [int or str or list of int or list of str, optional] Channels on which to calculate the statistic. If None, use all channels.

Returns

float or numpy array The geometric coefficient of variation of the events in the specified channels of *data*.

`FlowCal.stats.`**`gmean`**`(`*data*, *channels=None*`)`

Calculate the geometric mean of the events in an FCSData object.

Parameters

data [FCSData or numpy array] NxD flow cytometry data where N is the number of events and D is the number of parameters (aka channels).

channels [int or str or list of int or list of str, optional] Channels on which to calculate the statistic. If None, use all channels.

Returns

float or numpy array The geometric mean of the events in the specified channels of *data*.

`FlowCal.stats.`**`gstd`**(*data*, *channels=None*)

Calculate the geometric std. dev. of the events in an FCSData object.

Parameters

data [FCSData or numpy array] NxD flow cytometry data where N is the number of events and D is the number of parameters (aka channels).

channels [int or str or list of int or list of str, optional] Channels on which to calculate the statistic. If None, use all channels.

Returns

float or numpy array The geometric standard deviation of the events in the specified channels of *data*.

`FlowCal.stats.`**`iqr`**(*data*, *channels=None*)

Calculate the Interquartile Range of the events in an FCSData object.

Parameters

data [FCSData or numpy array] NxD flow cytometry data where N is the number of events and D is the number of parameters (aka channels).

channels [int or str or list of int or list of str, optional] Channels on which to calculate the statistic. If None, use all channels.

Returns

float or numpy array The Interquartile Range of the events in the specified channels of *data*.

Notes

The Interquartile Range (IQR) of a dataset is defined as the interval between the 25% and the 75% percentiles of such dataset.

`FlowCal.stats.`**`mean`**(*data*, *channels=None*)

Calculate the mean of the events in an FCSData object.

Parameters

data [FCSData or numpy array] NxD flow cytometry data where N is the number of events and D is the number of parameters (aka channels).

channels [int or str or list of int or list of str, optional] Channels on which to calculate the statistic. If None, use all channels.

Returns

float or numpy array The mean of the events in the specified channels of *data*.

`FlowCal.stats.`**`median`**(*data*, *channels=None*)

Calculate the median of the events in an FCSData object.

Parameters

data [FCSData or numpy array] NxD flow cytometry data where N is the number of events and D is the number of parameters (aka channels).

channels [int or str or list of int or list of str, optional] Channels on which to calculate the statistic. If None, use all channels.

Returns

float or numpy array The median of the events in the specified channels of *data*.

`FlowCal.stats.`**`mode`**`(`*data*, *channels=None*`)`

Calculate the mode of the events in an FCSData object.

Parameters

data [FCSData or numpy array] NxD flow cytometry data where N is the number of events and D is the number of parameters (aka channels).

channels [int or str or list of int or list of str, optional] Channels on which to calculate the statistic. If None, use all channels.

Returns

float or numpy array The mode of the events in the specified channels of *data*.

`FlowCal.stats.`**`rcv`**`(`*data*, *channels=None*`)`

Calculate the RCV of the events in an FCSData object.

Parameters

data [FCSData or numpy array] NxD flow cytometry data where N is the number of events and D is the number of parameters (aka channels).

channels [int or str or list of int or list of str, optional] Channels on which to calculate the statistic. If None, use all channels.

Returns

float or numpy array The Robust Coefficient of Variation of the events in the specified channels of *data*.

Notes

The Robust Coefficient of Variation (RCV) of a dataset is defined as the Interquartile Range (IQR) divided by the median of such dataset.

`FlowCal.stats.`**`std`**`(`*data*, *channels=None*`)`

Calculate the standard deviation of the events in an FCSData object.

Parameters

data [FCSData or numpy array] NxD flow cytometry data where N is the number of events and D is the number of parameters (aka channels).

channels [int or str or list of int or list of str, optional] Channels on which to calculate the statistic. If None, use all channels.

Returns

float or numpy array The standard deviation of the events in the specified channels of *data*.

2.5.7 FlowCal.transform module

Functions for transforming flow cytometry data

All transformations are of the following form:

```
data_t = transform(data, channels, *args, **kwargs):
```

where *data* and *data_t* are NxD FCSData objects or numpy arrays, representing N events with D channels, *channels* indicate the channels in which to apply the transformation, and *args* and *kwargs* are transformation-specific parameters. Each transformation function can apply its own restrictions or defaults on *channels*.

If *data* is an FCSData object, *transform* should rescale `data.range` if necessary.

`FlowCal.transform.`**`to_mef`**`(`*data*, *channels*, *sc_list*, *sc_channels=None*`)`

Transform flow cytometry data using a standard curve function.

This function accepts a list of standard curves (*sc_list*) and a list of channels to which those standard curves should be applied (*sc_channels*). *to_mef* automatically checks whether a standard curve is available for each channel specified in *channels*, and throws an error otherwise.

This function is intended to be reduced to the following signature:

```
to_mef_reduced(data, channels)
```

by using `functools.partial` once a list of standard curves and their respective channels is available.

Parameters

data [FCSData or numpy array] NxD flow cytometry data where N is the number of events and D is the number of parameters (aka channels).

channels [int, str, list of int, list of str] Channels on which to perform the transformation. If *channels* is None, perform transformation in all channels specified on *sc_channels*.

sc_list [list of functions] Functions implementing the standard curves for each channel in *sc_channels*.

sc_channels [list of int or list of str, optional] List of channels corresponding to each function in *sc_list*. If None, use all channels in *data*.

Returns

FCSData or numpy array NxD transformed flow cytometry data.

Raises

ValueError If any channel specified in *channels* is not in *sc_channels*.

`FlowCal.transform.`**`to_rfi`**`(`*data*, *channels=None*, *amplification_type=None*, *amplifier_gain=None*, *resolution=None*`)`

Transform flow cytometry data to Relative Fluorescence Units (RFI).

If `amplification_type[0]` is different from zero, data has been taken using a log amplifier. Therefore, to transform to RFI, the following operation is applied:

```
y = a[1]*10^(a[0] * (x/r))
```

Where `x` and `y` are the original and transformed data, respectively; `a` is *amplification_type* argument, and `r` is *resolution*. This will transform flow cytometry data taken with a log amplifier and an ADC of range `r` to linear RFIs, such that it covers `a[0]` decades of signal with a minimum value of `a[1]`.

If `amplification_type[0]==0`, however, a linear amplifier has been used and the following operation is applied instead:

```
y = x/g
```

Where `g` is *amplifier_gain*. This will transform flow cytometry data taken with a linear amplifier of gain `g` back to RFIs.

Parameters

data [FCSData or numpy array] NxD flow cytometry data where N is the number of events and D is the number of parameters (aka channels).

channels [int, str, list of int, list of str, optional] Channels on which to perform the transformation. If *channels* is None, perform transformation in all channels.

amplification_type [tuple or list of tuple] The amplification type of the specified channel(s). This should be reported as a tuple, in which the first element indicates how many decades the logarithmic amplifier covers, and the second indicates the linear value that corresponds to a channel value of zero. If the first element is zero, the amplification type is linear. This is similar to the $PnE keyword from the FCS standard. If None, take *amplification_type* from `data.amplification_type(channel)`.

amplifier_gain [float or list of floats, optional] The linear amplifier gain of the specified channel(s). Only used if `amplification_type[0]==0` (linear amplifier). If None, take *amplifier_gain* from `data.amplifier_gain(channel)`. If *data* does not contain `amplifier_gain()`, use 1.0.

resolution [int, float, or list of int or float, optional] Maximum range, for each specified channel. Only needed if `amplification_type[0]!=0` (log amplifier). If None, take *resolution* from `len(data.domain(channel))`.

Returns

FCSData or numpy array NxD transformed flow cytometry data.

`FlowCal.transform.`**`transform`**(*data*, *channels*, *transform_fxn*, *def_channels=None*)

Apply some transformation function to flow cytometry data.

This function is a template transformation function, intended to be used by other specific transformation functions. It performs basic checks on *channels* and *data*. It then applies *transform_fxn* to the specified channels. Finally, it rescales `data.range` and if necessary.

Parameters

data [FCSData or numpy array] NxD flow cytometry data where N is the number of events and D is the number of parameters (aka channels).

channels [int, str, list of int, list of str, optional] Channels on which to perform the transformation. If *channels* is None, use def_channels.

transform_fxn [function] Function that performs the actual transformation.

def_channels [int, str, list of int, list of str, optional] Default set of channels in which to perform the transformation. If *def_channels* is None, use all channels.

Returns

data_t [FCSData or numpy array] NxD transformed flow cytometry data.

2.6 Contribute

2.6.1 How to Contribute

If you are interested in contributing to this project, either by writing code, correcting a bug, or adding a new feature, we would love your help! Below we provide some guidelines on how to contribute.

`FlowCal` Installation for Developers

Regardless of your OS version, we recommend using `virtualenv` for development. A short primer on `virtualenv` can be found at http://docs.python-guide.org/en/latest/dev/virtualenvs/.

The recommended way to install `FlowCal` for development is to run `python setup.py develop`. This will install `FlowCal` in a special "developer" mode. In this mode, a link pointing to the `FlowCal` directory is made in the python installation directory, allowing you to import `FlowCal` from any python script, while at the same time being able to modify `FlowCal`'s code and immediately see the resulting effects.

Version Control

`FlowCal` uses `git` for version control. We try to follow the git-flow branching model. Please familiarize yourself with such model before contributing. A quick summary of relevant branches is given below.

- `master` is only used for final release versions. **Do not** directly commit to `master`, ever.
- `develop` holds unreleased features, which will eventually be released into `master`.
- *Feature branches* are branches derived from `develop`, in which new features are committed. When the feature is completed, a merge request towards `develop` should be made.

Recommended Workflow

A recommended workflow for contributing to `FlowCal` is as follows:

1. Report your intended change in the issue tracker on `github`. If reporting a bug, please be as detailed as possible and try to include the necessary steps to reproduce the problem. If suggesting a feature, indicate if you're willing to write the code for it.
2. Assuming that you decided to write code, clone the repo in your computer. You can use the command `git clone https://github.com/taborlab/FlowCal` if you are using the command-line version of `git`.
3. Switch to the develop branch, using `git checkout develop`.
4. Create a new feature branch, using `git checkout -b <feature_name>`.
5. Set up your virtual environment, if desired.
6. Install `FlowCal` in developer mode, using `python setup.py develop`.
7. Write/test code, commit. Repeat until feature is fully implemented.
8. Push and submit a merge request towards `develop`.

Version Policy

The version number in `FlowCal` is organized as follows: `MAJOR.MINOR.PATCH`. The following are guidelines on how to manage version numbers:

- The patch version number should only be increased after fixing a bug or an incompatibility issue, if the public API was not modified at all.
- The minor version number should be increased after a relatively minor API modification. For example:
  - After fixing a bug, when a minor API modification was required to do so.
  - After making a small adjustment to a function signature, such as adding a new argument or changing the data type of an existing one.
  - After adding one or more relatively minor new features (e.g. a new plotting function).
- The major version number should be increased after a fundamental modification to the API and/or the package, or the introduction of a major feature. For example:
  - After completely reorganizing the FCSData object or the functions in the package
  - After introducing a new Excel UI with a completely reorganized input file format.
  - After introducing a Graphical User Interface.

In general, new patch versions should not break a user's code, whereas minor versions should not require more than minor adjustments. Major versions could either require significant changes in the user's code or a complete change in the way they think about `FlowCal`'s API.

2.6.2 Report Bugs

The official way to report a bug is through the issue tracker on github (https://github.com/taborlab/FlowCal/issues). Try to be as explicit as possible when describing your issue. Ideally, a set of instructions to reproduce the error should be provided, together with the version of all the relevant packages you are using.

If you are interested in writing the code necessary to solve a bug, please visit *How to Contribute* first.

2.6.3 Request Features

The official way to request features is through the issue tracker on github (https://github.com/taborlab/FlowCal/issues). Try to be as descriptive as possible about the desired feature.

If you are interested in writing the code necessary to implement your feature, please visit *How to Contribute* first.

Bibliography

[1] P.N. Dean, C.B. Bagwell, T. Lindmo, R.F. Murphy, G.C. Salzman, “Data file standard for flow cytometry. Data File Standards Committee of the Society for Analytical Cytology,” Cytometry vol 11, pp 323-332, 1990, PMID 2340769.

[2] L.C. Seamer, C.B. Bagwell, L. Barden, D. Redelman, G.C. Salzman, J.C. Wood, R.F. Murphy, “Proposed new data file standard for flow cytometry, version FCS 3.0,” Cytometry vol 28, pp 118-122, 1997, PMID 9181300.

[3] J. Spidlen, et al, “Data File Standard for Flow Cytometry, version FCS 3.1,” Cytometry A vol 77A, pp 97-100, 2009, PMID 19937951.

[4] R. Hicks, “BD$WORD file header fields,” https://lists.purdue.edu/pipermail/cytometry/2001-October/020624.html

[1] D.R. Parks, M. Roederer, W.A. Moore, “A New Logicle Display

[1] P.N. Dean, C.B. Bagwell, T. Lindmo, R.F. Murphy, G.C. Salzman, “Data file standard for flow cytometry. Data File Standards Committee of the Society for Analytical Cytology,” Cytometry vol 11, pp 323-332, 1990, PMID 2340769.

[2] L.C. Seamer, C.B. Bagwell, L. Barden, D. Redelman, G.C. Salzman, J.C. Wood, R.F. Murphy, “Proposed new data file standard for flow cytometry, version FCS 3.0,” Cytometry vol 28, pp 118-122, 1997, PMID 9181300.

[3] J. Spidlen, et al, “Data File Standard for Flow Cytometry, version FCS 3.1,” Cytometry A vol 77A, pp 97-100, 2009, PMID 19937951.

[1] P.N. Dean, C.B. Bagwell, T. Lindmo, R.F. Murphy, G.C. Salzman, “Data file standard for flow cytometry. Data File Standards Committee of the Society for Analytical Cytology,” Cytometry vol 11, pp 323-332, 1990, PMID 2340769.

[2] L.C. Seamer, C.B. Bagwell, L. Barden, D. Redelman, G.C. Salzman, J.C. Wood, R.F. Murphy, “Proposed new data file standard for flow cytometry, version FCS 3.0,” Cytometry vol 28, pp 118-122, 1997, PMID 9181300.

[3] J. Spidlen, et al, “Data File Standard for Flow Cytometry, version FCS 3.1,” Cytometry A vol 77A, pp 97-100, 2009, PMID 19937951.

[1] P.N. Dean, C.B. Bagwell, T. Lindmo, R.F. Murphy, G.C. Salzman, “Data file standard for flow cytometry. Data File Standards Committee of the Society for Analytical Cytology,” Cytometry vol 11, pp 323-332, 1990, PMID 2340769.

[2] L.C. Seamer, C.B. Bagwell, L. Barden, D. Redelman, G.C. Salzman, J.C. Wood, R.F. Murphy, “Proposed new data file standard for flow cytometry, version FCS 3.0,” Cytometry vol 28, pp 118-122, 1997, PMID 9181300.

[3] J. Spidlen, et al, "Data File Standard for Flow Cytometry, version FCS 3.1," Cytometry A vol 77A, pp 97-100, 2009, PMID 19937951.

[1] P.N. Dean, C.B. Bagwell, T. Lindmo, R.F. Murphy, G.C. Salzman, "Data file standard for flow cytometry. Data File Standards Committee of the Society for Analytical Cytology," Cytometry vol 11, pp 323-332, 1990, PMID 2340769.

[2] L.C. Seamer, C.B. Bagwell, L. Barden, D. Redelman, G.C. Salzman, J.C. Wood, R.F. Murphy, "Proposed new data file standard for flow cytometry, version FCS 3.0," Cytometry vol 28, pp 118-122, 1997, PMID 9181300.

[3] J. Spidlen, et al, "Data File Standard for Flow Cytometry, version FCS 3.1," Cytometry A vol 77A, pp 97-100, 2009, PMID 19937951.

Python Module Index

Index