# FRETBursts Documentation

***Release 0.7.1+0.gc51b***

**Antonino Ingargiola**

**Sep 20, 2022**

# CONTENTS

**FRETBursts** is an open-source python package for burst analysis of freely-diffusing single-molecule FRET data for single and multi-spot experiments. FRETBursts supports both single-laser and dual-laser alternated excitation (ALEX and PAX) as well as ns-ALEX (or PIE).

We provide well-tested implementations of state-of-the-art algorithms for confocal smFRET analysis. We focus on computational reproducibility, by using Jupyter notebook based interfaces.

Please send questions or report issue on GitHub.

CHAPTER

ONE

# GETTING STARTED

## 1.1 Getting started for the absolute python beginner

Before running FRETBursts you need to install a python distribution that includes the Jupyter/IPython Notebook application.

You can find a quick guide for installing the software and running your first notebook here:

- 

Once you are able start Jupyter Notebook application and open a notebook you can move to the next section.

### 1.1.1 Installing FRETBursts

To install FRETBursts, make sure you close Jupyter Notebook, then type the following commands in a terminal (i.e. `cmd` on Windows or `Terminal` on OSX):

```
conda install fretbursts -c conda-forge
```

The installation should take a few seconds. If you notice any error please report it by opening a new issue on the FRETBursts GitHub Issues.

### 1.1.2 Running FRETBursts tutorial notebook

Download the ZIP file of FRETBursts notebooks and extract it inside a folder accessible by the Jupyter Notebook App.

Next, in the new Jupyter Notebook Dashboard click on the folder containing the FRETBursts notebooks.

For first time users, we recommend to start from the notebook:

- FRETBursts - us-ALEX smFRET burst analysis

and follow the instructions therein.

Remember, to run the notebooks step-by-step (one cell a time) keep pressing *shift + enter*. To run the entire notebook in a single step click on menu *Cell -> Run All*.

For more info how to run/edit a notebook see .

## 1.2 FRETBursts Installation

FRETBursts can be installed as a standard python package either via conda or PIP (see below). Being written in python, FRETBursts runs on OS X, Windows and Linux.

For updates on the latest FRETBursts version please refer to the *Release Notes (What's new?)*.

### 1.2.1 Installing latest stable version

The preferred way to to install and keep FRETBursts updated is through conda, a package manager used by Anaconda scientific python distribution. If you haven't done it already, please install the python3 version of Continuum Anaconda distribution (legacy python 2.7 works at the moment but it will be discontinued soon). Then, you can install or upgrade FRETBursts with:

```
conda install fretbursts -c conda-forge
```

After the installation, it is recommended that you download and run the FRETBursts notebooks to get familiar with the workflow. If you don't know what a Jupyter Notebooks is and how to launch it please see:

- Jupyter/IPython Notebook Quick Start Guide

See also the FRETBursts documentation section: *Running FRETBursts*.

### 1.2.2 Alternative methods: using PIP

Users that prefer using PIP, have to make sure that all the non-pure python dependencies are properly installed (i.e. numpy, scipy, pandas, matplotlib, pyqt, pytables), then use the usual:

```
pip install fretbursts --upgrade
```

The previous command installs or upgrades FRETBursts to the latest stable release.

### 1.2.3 Install FRETBursts in a stand-alone environment

For reproducibiltity, it is better to install FRETBursts in a dedicated environment. The instructions below create a new conda environment with python 3.7:

First, add the conda-forge channel containing the fretbursts (do it only once after installing Anaconda):

```
conda config --append channels conda-forge
```

Then create a new conda environment with python 3.7 and FRETbursts:

```
conda create -n py37-fb python=3.7 fretbursts
conda activate py37-fb
conda install pyqt   # optional
pip install pybroom  # optional
python -m ipykernel install --user --name py37-fb --display-name "Python 3.7 (FB)"
```

The last command installs the jupyter kernel so that you can use the new environment from jupyter notebooks.

This method allows to easily backup and reinstall a working environment, or install it on a different machine (with same OS). This is useful for replicating an environment on multiple machine, for recovering from a broken anaconda installation or for reproducibility of published results.

More info:

- Using conda environments
- Managing conda channels
- Installing a jupyter kernel

### 1.2.4 Install latest development version

As a rule, all new development takes place on separate "feature branches". The master branch should always be stable and releasable. The advantage of installing from the master branch is that you can get updates without waiting for a formal release. If there are some errors you can always roll back to the latest released version to get your job done. Since you have the full version down to the commit level printed in the notebook you will know which version works and which does not.

You can install the latest development version directly from GitHub with:

```
pip install git+git://github.com/OpenSMFS/FRETBursts.git
```

**Note:** Note that the previous command fails if git is not installed.

Alternatively you can do an "editable" installation, i.e. executing FRETBursts from the source folder. In this case, modifications in the source files are immediately available on the next FRETBursts import. To do so, clone FRETBursts and install it as follows:

```
git clone https://github.com/OpenSMFS/FRETBursts.git
cd FRETBursts
pip install -e .
```

It is recommended that you install cython before FRETBursts so that the optimized C routines are installed as well. Also, make sure you have `lmfit` and `seaborn` installed before running FRETBursts.

## 1.3 Running FRETBursts

After installation, FRETBursts can be imported with:

```
from fretbursts import *
```

that will also import numpy (as `np`) and matplolib.pyplot (as `plt`). This is the syntax used throughout the tutorials.

Alternatively, you can import FRETBursts in its own namespace (which is cleaner):

```
import fretbursts as fb
```

To get started with FRETBursts it is recommended that you download the FRETBursts notebooks that contains live tutorials ready to run and modify.

### 1.3.1 Why a notebook-based workflow

Jupyter Notebooks is the recommended environment to perform interactive analysis with FRETBursts.

The FRETBursts tutorials are Jupyter notebooks and, typically, a new analysis is performed by copying and modifying an existing notebook.

The FRETBursts notebooks display and store the exact FRETBursts version (including the revision) used in the execution. Saving the software revision together with analysis commands and results allows long term reproducibility and provides a lightweight approach for regression testing.

For more information on installing and first steps with Jupyter Notebook see:

- Jupyter/IPython Notebook Quick Start Guide

## 1.4 FRETBursts Dependencies

For documentation purposes, this is the list of dependencies to run FRETBursts:

- Python 3.6+
- Numpy 1.6+
- Scipy 0.17+
- Matplotlib 3+, with QT4 backend (either PyQT4 or PySide) or QT5.
- PyTables 3.x. To load/save the *Photon-HDF5*.
- lmfit 0.9.3+, used for flexible histogram fitting.
- Jupyter environment: notebook, ipython, ipywidgets.
- Pandas, for nice table representation and exporting data.

If you want to compile the cython extensions (optional) you also need:

- cython 0.20 or newer.
- a C compiler

For developing FRETBursts you should also install

- sphinx 1.3+ (we use napoleon extension) to build this documentation.
- pytest to execute the unit tests.

Note that, unless you know what you are doing, you should never install these dependencies manually. Use a scientific python distribution like Continuum Anaconda instead.

CHAPTER

TWO

# FRETBURSTS RELEASE NOTES

## 2.1 Version 0.7.1

- Require Python 3.6+. Python 2.7 is not supported anymore.
- Fix deprecation warning when plotting timetraces. Now matplotlib 3+ is required.
- Fix error loading Photon-HDF5 files with polarization data (issue)
- More fixes for PIE file with polarization, thanks to Christian Gebhardt for reporting the problem and suggesting solutions (issue)
- Passing list of strings to `loader.photon_hdf5()` loads each file into the same data object as an excitation spot.
- dplot function keyword argument `i=None` now plots concatenated data from all excitation spots. Does not apply to trace-based plots
- Fitter attributes relating to fit values now have parallel attributes ending in `_tot` which are for concatentated data across all spots.

## 2.2 Version 0.7 (Jul. 2018)

To update to the latest FRETBursts version type `conda install fretbursts -c conda-forge`. For more detailed instructions see *Getting Started*.

Exporting:

- Export photon burst data to pandas DataFrame (function bext.burst_photons)

Loading:

- Support for Photon-HDF5 0.5 and validation during loading
- Add function to load SM files acquired with 1-laser (96d39b)
- Support smFRET-1color measurements from "generic" Photon-HDF5 (ab87e8)
- Faster loading of nsALEX data when `ondisk=True` (a6b343)
- Add support for loading polarization and split data as "spectral" (a5b7d6, c73188)

Analysis:

- Background computation improvements: more robust, faster, better error messages (4fbf33, 7a3c17, 5a68d0,)

Other:

- New documentation theme (docs live at the same address fretbursts.readthedocs.io)

- A myriad of small improvements and bug and regression fixes (see git log for details)

## 2.3 Version 0.6.5 (Aug. 2017)

This is a minor release with an important bug fix for histograms plots and other tweaks mostly for PAX. New "short notebooks" for common tasks have also been added.

Bug fixes:

- Fix histograms offset by half bin when using matplotlib 2.x. (see commit d3102e).
- Fix `BurstsGap` giving an error when being sliced (see #62).

Other changes:

- Kinetics: better handling of time_zero in moving_window functions (see c25b68).
- Multispot: Add argument `skip_ch` to `Data.collapse` and to `dplot`.
- Plots: use `vmin=1` by default in `alex_jointplot` and `hexbin_alex`.
- PAX: rewrote burst size and correction factors to be more clear and general (see *`Data.burst_sizes_pax_ich`*)
- Plots: spread burst labels to reduce overlapping when plotting burst info with *`timetrace`*. See the new example notebook for timetrace plotting.
- **New notebooks:**
  - Example - Plotting timetraces with bursts
  - Example - Selecting FRET populations
  - Example - FRET histogram fitting

## 2.4 Version 0.6.4 (Jul. 2017)

This release adds support for *periodic acceptor excitation* (PAX) measurements. PAX is similar to s-ALEX, with the difference that only the A laser is alternated (see references [pax] and [48spot]). There are also a few minor bug fixes and better support for 48-spot data.

To update to the latest version type `conda install fretbursts -c conda-forge`. For installation instructions see *Getting Started*.

The list of changes include:

- Added PAX support
- Workaround for a numpy.histogram issue when input contains NaNs
- *`bext.burst_data()`*: bugfix, add tests and improve handling of multispot data
- Added `apionly` argument to `init_notebook()` for setting up the notebook plots without changing any plot style (see 958824).
- Support "empty" channels in multispot data.
- Improve plots for 48-spot data.
- **Refactoring of *`alex_jointplot`*.**

- Allow using custom `Data` fields for E and S in `alex_jointplot`.
- Remove rarely used arguments
- Set axis limits by default

- Added a new notebook showing how to customize *alex_jointplot* plots.
- Improved normalization of exponential curve representing the fitted background in *hist_bg* (see Issue 61). Many thanks to Danielis Rutkauskas for reporting the issue.
- Removed shortcut (underscore) syntax for single-spot. Code like `d.E_` needs to be changed to `d.E[0]`. This syntax was causing difficulties during developing new features for PAX. Please report if you would like for the syntax to be reintroduced.

## 2.5 Version 0.6.3 (Apr. 2017)

A few more small fixes in this release. If you have any installation issue please report it on github.

- Import `OpenFileDialog` when FRETBursts is imported (as in versions < 0.6.2)
- Fix loading SM files with numpy 1.12
- Use `phconvert` to decode SM files

## 2.6 Version 0.6.2 (Apr. 2017)

This is a technical release that removes the hard dependency on QT and solves some installation issues due to QT pinning on conda-forge.

## 2.7 Version 0.6.1 (Apr. 2017)

For this version of FRETBursts, conda packages are distributed for python 2.7, 3.5, 3.6 and numpy 1.11 and 1.12. FRETBursts still works with python 3.4 but conda packages are not provided anymore. Python 2.7 is now deprecated. Support for python 2.7 will be removed in a future version.

The current release includes the following changes:

- SangYoon Chung (@chungjjang80) found that the `L` argument in burst search was ignored and submitted a fix to the problem in PR #57. Tests were added to avoid future regressions.
- Fix access to the deprecated background attributes (introduced in 0.6). See b850a5.
- Add plot wrapper for 16-ch data.
- Improved example notebook showing how to export burst data. See Exporting Burst Data.
- Re-enable background rate caching. See PR #53.
- Support Path objects as filename in `loader.photon_hdf5()`. See 201b5c.
- Improve `Ph_sel` string representation, added factory method `Ph_sel.from_str` and added new tests. See 3dc5f0.

## 2.8 Version 0.6 (Jan. 2017)

- Improvements to the layout of 48-spot plots.
- Simplify background computation avoiding useless recomputations. This results in 3x speed increase in background computation for measurement loaded with `ondisk=True` and 30% speed increase when using `ondisk=False`. Now all background rates are stored in the dictionary *`Data.bg`*, while the mean background rate in the dictionary *`Data.bg_mean`*. The old attributes `Data.bg_*` and `Data.rate_*` have been deprecated and will be removed in a future release (see below).
- Fix loading files with `ondisk=True`. With this option timestamps are not kept in RAM but loaded spot-by-spot when needed. This option has no effect on single-spot measurements but will save RAM in multi-spot measurements.
- Add new plot functions hist_interphoton and hist_interphoton_single to plot the interphoton delay distribution. In previous versions the function `hist_bg` (and `hist_bg_single`) did the same plot but required the background to be fitted. `hist_interphoton*` do not require any prior background fit and also have a cleaner and improved API.
- Detect and handle smFRET files (no ALEX) with counts not only in D or A channels (f0e33d).
- Better error message when a burst filtering function fails (c7826d).

### 2.8.1 Backward-incompatible changes

#### Effect on burst search

Version 0.6 introduced a small change in how the auto-threshold for background estimation is computed. This results in slightly different background rates. As a consequence, burst searches setting a threshold as function of the background, will set a slightly different threshold and will find different number of bursts. The difference is not dramatic, but can result in slight numeric changes in estimated parameters.

#### Details of auto-threshold changes

The refactor included a change in how the background is computed when using `tail_min_us='auto'`. As before, with this setting, the background is estimated iteratively in two steps. A first raw estimation with a fixed threshold (250us), and second estimation with a threshold function of the rate computed in the first step. Before version 0.6, the first step estimated a single rate for the whole measurement. Now the first-step estimation is performed in each background period separately. As before, the second step computes the background separately in each background period. This change was motivated by the need to simplify the internal logic of background estimation, and to increase the computation efficiency and accuracy.

#### Background attributes

The background refactor resulted in an incompatible change in the *`Data.bg`* attribute. Users upgrading to version 0.6, may need to replace `Data.bg` with `Data.bg[Ph_sel('all')]` in their notebooks. Note that no official FRETBursts notebook was using `Data.bg`, so most users will not be affected.

#### Compatibility layer

All the old background-related attributes (bg_dd, bg_ad, bg_da, bg_aa, rate_dd, rate_ad, rate_da, rate_aa, rate_m) are still present but deprecated. The same data is now contained in the dictionaries *Data.bg* and *Data.bg_mean*. When using the deprecated attributes, a message will indicate the new syntax. If you see the deprecation warning, please update the notebook to avoid future errors.

#### Details of changed attributes

Before version 0.6, `Data.bg` contained background rates fitted for **all-photons** stream. `Data.bg` was a list of arrays: one array per spot, one array element per background period. In version 0.6+, `Data.bg` contains the background rates for **all** the fitted photon streams. `Data.bg` is now a dict using `Ph_sel` objects as keys. Each dict entry is a list of array, one array per spot and one array element per background period. For more details please refer to the following documentation *Data.bg* and *Data.bg_mean*.

## 2.9 Version 0.5.9 (Sep. 2016)

- Added support for pyqt and qt 5+.
- Fix burst selection with multispot data. See this commit.

There may still be some glitches when using the QT5 GUIs from the notebook, but installing (and importing) FRET-Bursts does not require QT4 anymore (QT5 is the current default in anaconda). Please report any issue.

## 2.10 Version 0.5.7 (Sep. 2016)

Refactoring and expansion of gamma and beta corrections. Briefly, in all the places where corrected burst sizes are being computed, we removed the `gamma1` argument and added a flag `donor_ref`. Additionally, the values `Data.S` are now beta corrected.

These changes affected several components as described below.

### 2.10.1 Data Class

- Methods `Data.burst_sizes_ich` and `Data.burst_sizes` now accept the arguments `gamma`, `beta` and `donor_ref`. The argument `gamma1` was removed. The two conventions of corrected burst sizes are chosen with the boolean flag `donor_ref`. See the burst_sizes_ich docs for details.
- New method `get_naa_corrected` returns the array of `naa` burst counts corrected with the passed `gamma` and `beta` values. Like for the burst size, the argument `donor_ref` selects the convention for the correction. See the get_naa_corrected docs for details.
- A new `Data` attribute `beta` (default: 1) stores a beta value that is used to compute the corrected S. This value is never implicitly used to compute corrected burst sizes or naa (for these a `beta` arguments needs to be passed explicitly).

### 2.10.2 Plot functions

Plot functions `hist_size` and `hist_brightness` accept the new arguments for corrected burst size (`gamma`, `beta` and `donor_ref`).

### 2.10.3 Burst selection

Burst selection by `size` and `naa` accept the new arguments for corrected burst size (`gamma`, `beta` and `donor_ref`).

### 2.10.4 Burst Weights

Functions that accept weights don't accept the gamma1 argument anymore, but they don't (yet) support the arguments `donor_ref` and `beta`. As a result, for the purpose of weighting, there is only one expression for corrected burst size (`na + gamma*nd`), with the option to add `naa` but without beta correction.

All these changes are covered by unit tests.

### 2.10.5 Installation via conda-forge

Since version 0.5.6 we started distributing conda packages for FRETBursts through the conda-forge channel (a community supported repository, as opposed to a private channel we were using before). To install or update FRETBursts you should now use:

```
conda install fretbursts -c conda-forge
```

Using the conda-forge channel simplifies our release process since their infrastructure automatically builds packages for multiple platforms and python versions. Please report any issues in installing or upgrading FRETBursts on the GitHub Issues page.

For more detailed installation instructions see the Getting Started documentation.

## 2.11 Version 0.5.6

For older release notes see GitHub Releases Page.

CHAPTER

THREE

# FRETBURSTS REFERENCE MANUAL

Contents:

## 3.1 Loader functions

While FRETBursts can load data files from different file formats, we advocate using Photon-HDF5, a file format specifically designed for freely-diffusing single-molecule spectroscopy data.

Photon-HDF5 files can be loaded with the function *photon_hdf5()*, regardless of the type of excitation or number of spots.

Single-spot s-ALEX measurement stored in SM files can be loaded via the function *usalex()* and single-spot ns-ALEX measurement stored in SPC files (Beckr & Hickl) can be loaded via the function *nsalex()*. To load data from arbitrary format see *Load data manually*.

Note that regardless of the format, for alternated excitation data, after loading the data you need to apply the alternation parameters using *alex_apply_period()*. After the parameters are applied you can proceed to background estimation and burst search.

**Contents**



### 3.1.1 List of loader functions

The `loader` module contains functions to load each supported data format. The loader functions load data from a specific format and return a new *fretbursts.burstlib.Data()* object containing the data.

This module contains the high-level function to load a data-file and to return a `Data()` object. The low-level functions that perform the binary loading and preprocessing can be found in the `dataload` folder.

fretbursts.loader.**alex_apply_period**(*d*, *delete_ph_t=True*)

Apply the ALEX period definition set in D_ON and A_ON attributes.

This function works both for us-ALEX and ns-ALEX data.

Note that you first need to load the data in a variable `d` and then set the alternation parameters using `d.add(D_ON=..., A_ON=...)`.

The typical pattern for loading ALEX data is the following:

```
d = loader.photon_hdf5(fname=fname)
d.add(D_ON=(2850, 580), A_ON=(900, 2580))
alex_plot_alternation(d)
```

If the plot looks good, apply the alternation with:

```
loader.alex_apply_period(d)
```

Now d is ready for further processing such as background estimation, burst search, etc…

fretbursts.loader.**nsalex**(*fname*)

Load nsALEX data from a SPC file and return a Data() object.

This function returns a Data() object to which you need to apply an alternation selection before performing further analysis (background estimation, burst search, etc.).

The pattern to load nsALEX data is the following:

```
d = loader.nsalex(fname=fname)
d.add(D_ON=(2850, 580), A_ON=(900, 2580))
alex_plot_alternation(d)
```

If the plot looks good apply the alternation with:

```
loader.alex_apply_period(d)
```

Now d is ready for further processing such as background estimation, burst search, etc…

fretbursts.loader.**nsalex_apply_period**(*d*, *delete_ph_t=True*)

Applies to the Data object d the alternation period previously set.

Note that you first need to load the data in a variable d and then set the alternation parameters using d.add(D_ON=..., A_ON=...).

The typical pattern for loading ALEX data is the following:

```
d = loader.photon_hdf5(fname=fname)
d.add(D_ON=(2850, 580), A_ON=(900, 2580))
alex_plot_alternation(d)
```

If the plot looks good, apply the alternation with:

```
loader.alex_apply_period(d)
```

Now d is ready for further processing such as background estimation, burst search, etc…

*See also:* `alex_apply_period()`.

fretbursts.loader.**photon_hdf5**(*filename*, *ondisk=False*, *require_setup=True*, *validate=False*, *fix_order=True*)

Load a data file saved in Photon-HDF5 format version 0.3 or higher.

Photon-HDF5 is a format for a wide range of timestamp-based single molecule data. For more info please see:

http://photon-hdf5.org/

**Parameters**

- **filename** (*str or pathlib.Path*) – path of the data file to be loaded.
- **ondisk** (*bool*) – if True, do not load the timestamps in memory using instead references to the HDF5 arrays. Default False.
- **require_setup** (*bool*) – if True (default) the input file need to have a setup group or won't be loaded. If False, accept files with missing setup group. Use False only for testing or DCR files.
- **validate** (*bool*) – if True validate the Photon-HDF5 file on loading. If False skip any validation.
- **fix_order** (*bool*) – if True fix the order of photons so all are monotonically increaseing in macrotime.

**Returns**
`fretbursts.burstlib.Data` object containing the data.

`fretbursts.loader.`**`sm_single_laser`**`(`*fname*`)`

Load SM files acquired using single-laser and 2 detectors.

`fretbursts.loader.`**`usalex`**`(`*fname*, *leakage=0*, *gamma=1.0*, *header=None*, *BT=None*`)`

Load usALEX data from a SM file and return a Data() object.

This function returns a Data() object to which you need to apply an alternation selection before performing further analysis (background estimation, burst search, etc.).

The pattern to load usALEX data is the following:

```
d = loader.usalex(fname=fname)
d.add(D_ON=(2850, 580), A_ON=(900, 2580), alex_period=4000)
plot_alternation_hist(d)
```

If the plot looks good, apply the alternation with:

```
loader.alex_apply_period(d)
```

Now `d` is ready for further processing such as background estimation, burst search, etc…

`fretbursts.loader.`**`usalex_apply_period`**`(`*d*, *delete_ph_t=True*, *remove_d_em_a_ex=False*`)`

Applies to the Data object `d` the alternation period previously set.

Note that you first need to load the data in a variable `d` and then set the alternation parameters using `d.add(D_ON=..., A_ON=...)`.

The typical pattern for loading ALEX data is the following:

```
d = loader.photon_hdf5(fname=fname)
d.add(D_ON=(2850, 580), A_ON=(900, 2580))
alex_plot_alternation(d)
```

If the plot looks good, apply the alternation with:

```
loader.alex_apply_period(d)
```

Now `d` is ready for further processing such as background estimation, burst search, etc…

*See also:* `alex_apply_period()`.

### 3.1.2 Load data manually

In case the data is available in a format not directly supported by FRETBursts it is possible to manually create a `fretbursts.burstslib.Data` object. For example, for non-ALEX smFRET data, two arrays of same length are needed: the timestamps and the acceptor-mask. The timestamps need to be an int64 numpy array containing the recorded photon timestamps in arbitrary units (usually dictated by the acquisition hardware clock period). The acceptor-mask needs to be a numpy boolean array that is `True` when the corresponding timestamps comes from the acceptor channel and `False` when it comes from the donor channel. Having these arrays a `Data` object can be manually created with:

```
d = Data(ph_times_m=[timestamps], A_em=[acceptor_mask],
         clk_p=10e-9, ALEX=False, nch=1, fname='file_name')
```

In the previous example, we set the timestamp unit (`clk_p`) to 10~ns and we specify that the data is not from an ALEX measurement. Creating `Data` objects for ALEX and ns-ALEX measurements follows the same lines.

## 3.2 The "Data()" class

The *Data* class is the main container for smFRET measurements. It contains timestamps, detectors and all the results of data processing such as background estimation, burst data, fitted FRET and so on.

The reference documentation of the class follows.

**Contents**



### 3.2.1 "Data()" class: description and attributes

A description of the *Data* class and its main attributes.

**class** `fretbursts.burstlib.`**`Data`**(*leakage=0.0*, *gamma=1.0*, *dir_ex=0.0*, ***kwargs*)

Container for all the information (timestamps, bursts) of a dataset.

Data() contains all the information of a dataset (name, timestamps, bursts, correction factors) and provides several methods to perform analysis (background estimation, burst search, FRET fitting, etc…).

When loading a measurement file a Data() object is created by one of the loader functions in `loaders.py`. Data() objects can be also created with *`Data.copy()`*, *`Data.fuse_bursts()`* or *`Data.select_bursts()`*.

To add or delete data-attributes use `.add()` or `.delete()` methods. All the standard data-attributes are listed below.

---

**Note:** Attributes of type "*list*" contain one element per channel. Each element, in turn, can be an array. For example `.ph_times_m[i]` is the array of timestamps for channel `i`; or `.nd[i]` is the array of donor counts in each burst for channel `i`.

---

**Measurement attributes**

**`fname`**

measurements file name

**Type**
string

**`nch`**

number of channels

**Type**
int

**`clk_p`**

clock period in seconds for timestamps in `ph_times_m`

**Type**
float

**`ph_times_m`**

list of timestamp arrays (int64). Each array contains all the timestamps (donor+acceptor) in one channel.

**Type**
list

**`A_em`**

list of boolean arrays marking acceptor timestamps. Each array is a boolean mask for the corresponding ph_times_m array.

**Type**
list

**`leakage`**

leakage (or bleed-through) fraction. May be scalar or same size as nch.

**Type**
float or array of floats

**`gamma`**

gamma factor. May be scalar or same size as nch.

**Type**
float or array of floats

**`D_em`**

**[ALEX-only]** boolean mask for `.ph_times_m[i]` for donor emission

**Type**
list of boolean arrays

`D_ex, A_ex`

**[ALEX-only]** boolean mask for `.ph_times_m[i]` during donor or acceptor excitation

**Type**
list of boolean arrays

`D_ON, A_ON`

**[ALEX-only]** start-end values for donor and acceptor excitation selection.

**Type**
2-element tuples of int

`alex_period`

**[ALEX-only]** duration of the alternation period in clock cycles.

**Type**
int

**Background Attributes**

The background is computed with *Data.calc_bg()* and is estimated in chunks of equal duration called *background periods*. Estimations are performed in each spot and photon stream. The following attributes contain the estimated background rate.

`bg`

background rates for the different photon streams, channels and background periods. Keys are `Ph_sel` objects and values are lists (one element per channel) of arrays (one element per background period) of background rates.

**Type**
dict

`bg_mean`

mean background rates across the entire measurement for the different photon streams and channels. Keys are `Ph_sel` objects and values are lists (one element per channel) of background rates.

**Type**
dict

`nperiods`

number of periods in which timestamps are split for background calculation, given per channel **NOTE: this is changed from previous versions, to support grouped experiments**

**Type**
array

`bg_fun`

function used to compute the background rates

**Type**
function

`Lim`

each element of this list is a list of index pairs for `.ph_times_m[i]` for **first** and **last** photon in each period.

**Type**
list

`Ph_p`

each element in this list is a list of timestamps pairs for **first** and **last** photon of each period.

**Type**
list

`bg_ph_sel`
photon selection used by Lim and Ph_p. See *fretbursts.ph_sel* for details.

**Type**
Ph_sel object

`Th_us`
thresholds in us used to select the tail of the interphoton delay distribution. Keys are `Ph_sel` objects and values are lists (one element per channel) of arrays (one element per background period).

**Type**
dict

Additionlly, there are a few deprecated attributes (`bg_dd`, `bg_ad`, `bg_da`, `bg_aa`, `rate_dd`, `rate_ad`, `rate_da`, `rate_aa` and `rate_m`) which will be removed in a future version. Please use *Data.bg* and *Data.bg_mean* instead.

**Burst search parameters (user input)**

These are the parameters used to perform the burst search (see *burst_search()*).

`ph_sel`
photon selection used for burst search. See *fretbursts.ph_sel* for details.

**Type**
Ph_sel object

`m`
number of consecutive timestamps used to compute the local rate during burst search

**Type**
int

`L`
min. number of photons for a burst to be identified and saved

**Type**
int

`P`
valid values [0..1]. Probability that a burst-start is due to a Poisson background. The employed Poisson rate is the one computed by `.calc_bg()`.

**Type**
float, probability

`F`
`(F * background_rate)` is the minimum rate for burst-start

**Type**
float

**Burst search data (available after burst search)**

When not specified, parameters marked as (list of arrays) contains arrays with one element per bursts. `mburst` arrays contain one "row" per burst. `TT` arrays contain one element per `period` (see above: background attributes).

`mburst`

list Bursts() one element per channel. See *fretbursts.phtools.burstsearch.Bursts*.

**Type**
list of Bursts objects

`TT`

list of arrays of $T$ values (in sec.). A $T$ value is the maximum delay between `m` photons to have a burst-start. Each channels has an array of $T$ values, one for each background "period" (see above).

**Type**
list of arrays

`T`

per-channel mean of `TT`

**Type**
array

`nd, na`

number of donor or acceptor photons during donor excitation in each burst

**Type**
list of arrays

`nt`

total number photons (nd+na+naa)

**Type**
list of arrays

`naa`

number of acceptor photons in each burst during acceptor excitation **[ALEX only]**

**Type**
list of arrays

`nar`

number of acceptor photons in each burst during donor excitation, not corrected for D-leakage and A-direct-excitation. **[PAX only]**

**Type**
list of arrays

`bp`

time period for each burst. Same shape as `nd`. This is needed to identify the background rate for each burst.

**Type**
list of arrays

`bg_bs`

background rates used for threshold computation in burst search (is a reference to `bg`, `bg_dd` or `bg_ad`).

**Type**
list

`fuse`

if not None, the burst separation in ms below which bursts have been fused (see `.fuse_bursts()`).

**Type**
None or float

**E**

FRET efficiency value for each burst: E = na/(na + gamma*nd).

**Type**
list

**S**

stoichiometry value for each burst: S = (gamma*nd + na) /(gamma*nd + na + naa)

**Type**
list

### 3.2.2 Summary information

List of `Data` attributes and methods providing summary information on the measurement:

**class** `fretbursts.burstlib.`**Data**

**time_max**

The last recorded time in seconds.

**time_min**

The first recorded time in seconds.

**ph_data_sizes**

Array of total number of photons (ph-data) for each channel.

**num_bursts**

Array of number of bursts in each channel.

**burst_sizes**(*gamma=1.0*, *add_naa=False*, *beta=1.0*, *donor_ref=True*)

Return gamma corrected burst sizes for all the channel.

Compute burst sizes by calling, for each channel, `burst_sizes_ich()`.

See `burst_sizes_ich()` for description of the arguments.

**Returns**
List of arrays of burst sizes, one array per channel.

**burst_sizes_pax_ich**(*ich=0*, *ph_sel=Ph_sel(Dex='DAem', Aex='DAem')*, *naa_aexonly=False*, *naa_comp=False*, *na_comp=False*, *gamma=1.0*, *beta=1.0*, *donor_ref=True*)

Return different definitions of PAX burst sizes for channel `ich`.

There are 4 basic "terms" corresponding to the 4 photon streams: `nd`, `na`, `nda`, `naa`. Which term is included is defined by the `ph_sel` argument (by default all are included). The other arguments specify the various corrections for each term.

**Parameters**

- **ich** (*int*) – the spot number, only relevant for multi-spot. In single-spot data there is only one channel (`ich=0`) so this argument may be omitted. Default 0.
- **gamma** (*float*) – coefficient for gamma correction of burst sizes. Default: 1. For more info see explanation above.
- **beta** (*float*) – beta correction factor used for the DAexAem term.
- **donor_ref** (*bool*) – True or False select different conventions for burst size correction. For details see `fretbursts.burstlib.Data.burst_sizes_ich()`.

- **ph_sel** (*Ph_sel object*) – defines which terms are included in the burst size.
- **na_comp** (*bool*) – If True, multiply the `na` term by `(1 + Wa/Wd)`, where Wa and Wd are the D and A alternation durations (typically Wa/Wd = 1).
- **naa_aexonly** (*bool*) – if True, the `naa` term is corrected to include only A emission due to A excitation. If False, the `naa` term includes all the counts in DAexAem. The `naa` term also depends on the `naa_comp` argument.
- **naa_comp** (*bool*) – If True, multiply the `naa` term by `(1 + Wa/Wd)`, where Wa and Wd are the D and A alternation durations (typically Wa/Wd = 1). The `naa` term also depends on the `naa_aexonly` argument.

**Returns**
Array of burst sizes for channel `ich`.

---

**Examples**

Burst sizes with all streams and no correction:

```
Data.burst_sizes_pax_ich(ph_sel=Ph_sel('all'))
```

$$F_{D_{ex}D_{em}} + F_{DA_{ex}D_{em}} + F_{FRET} + F_{DA_{ex}A_{em}}$$

Burst sizes with all streams and all corrections:

```
Data.burst_sizes_pax_ich(ph_sel=Ph_sel('all'), na_comp=True,
                         aa_aexonly=True, naa_comp=True)
```

$$\gamma(F_{D_{ex}D_{em}} + F_{DA_{ex}D_{em}}) + \left(1 + \frac{W_A}{W_D}\right)(F_{FRET} + (F_{DA_{ex}A_{em}} - F_{D_{ex}A_{em}})\,\beta^{-1})$$

---

**See also:**

*Data.burst_sizes_ich()*

**burst_sizes_ich**(*ich=0*, *gamma=1.0*, *add_naa=False*, *beta=1.0*, *donor_ref=True*)

Return gamma corrected burst sizes for channel `ich`.

If `donor_ref == True` (default) the gamma corrected burst size is computed according to:

```
1)    nd + na / gamma
```

Otherwise, if `donor_ref == False`, the gamma corrected burst size is:

```
2)    nd * gamma  + na
```

With the definition (1) the corrected burst size is equal to the raw burst size for zero-FRET or D-only bursts (that's why is `donor_ref`). With the definition (2) the corrected burst size is equal to the raw burst size for 100%-FRET bursts.

In an ALEX measurement, use `add_naa = True` to add counts from AexAem stream to the returned burst size. The argument `gamma` and `beta` are used to correctly scale `naa` so that it become commensurate with the Dex corrected burst size. In particular, when using definition (1) (i.e. `donor_ref = True`), the total burst size is:

```
(nd + na/gamma) + naa / (beta * gamma)
```

Conversely, when using definition (2) (`donor_ref = False`), the total burst size is:

```
(nd * gamma + na) + naa / beta
```

**Parameters**

- **ich** (*int*) – the spot number, only relevant for multi-spot. In single-spot data there is only one channel (`ich=0`) so this argument may be omitted. Default 0.
- **add_naa** (*boolean*) – when True, add a term for AexAem photons when computing burst size. Default False.
- **gamma** (*float*) – coefficient for gamma correction of burst sizes. Default: 1. For more info see explanation above.
- **beta** (*float*) – beta correction factor used for the AexAem term of the burst size. Default 1. If `add_naa = False` or measurement is not ALEX this argument is ignored. For more info see explanation above.
- **donor_ref** (*bool*) – select the convention for burst size correction. See details above in the function description.

**Returns**
Array of burst sizes for channel `ich`.

`burst_widths`

List of arrays of burst duration in seconds. One array per channel.

`ph_in_bursts_ich`(*ich=0*, *ph_sel=Ph_sel(Dex='DAem', Aex='DAem')*)

Return timestamps of photons inside bursts for channel `ich`.

**Returns**
Array of photon timestamps in channel `ich` and photon selection `ph_sel` that are inside any burst.

`ph_in_bursts_mask_ich`(*ich=0*, *ph_sel=Ph_sel(Dex='DAem', Aex='DAem')*)

Return mask of all photons inside bursts for channel `ich`.

**Returns**
Boolean array for photons in channel `ich` and photon selection `ph_sel` that are inside any burst.

`status`(*add=''*, *noname=False*)

Return a string with burst search, corrections and selection info.

`name`

last subfolder + file name with no extension.

**Type**
Measurement name

`Name`(*add=''*)

Return short filename + status information.

### 3.2.3 Analysis methods

The following methods perform background estimation, burst search and burst-data calculations:

- *Data.calc_bg()*
- *Data.burst_search()*
- *Data.calc_fret()*
- *Data.calc_ph_num()*
- *Data.fuse_bursts()*
- *Data.calc_sbr()*
- *Data.calc_max_rate()*

The methods documentation follows:

`class fretbursts.burstlib.Data`

**calc_bg**(*fun*, *time_s=60*, *tail_min_us=500*, *F_bg=2*, *error_metrics=None*, *fit_allph=True*)

Compute time-dependent background rates for all the channels.

Compute background rates for donor, acceptor and both detectors. The rates are computed every `time_s` seconds, allowing to track possible variations during the measurement.

**Parameters**

- **fun** (*function*) – function for background estimation (example `bg.exp_fit`)
- **time_s** (*float, seconds*) – compute background each time_s seconds
- **tail_min_us** (*float, tuple or string*) – min threshold in us for photon waiting times to use in background estimation. If float is the same threshold for 'all', DD, AD and AA photons and for all the channels. If a 3 or 4 element tuple, each value is used for 'all', DD, AD or AA photons, same value for all the channels. If 'auto', the threshold is computed for each stream ('all', DD, DA, AA) and for each channel as `bg_F * rate_ml0`. `rate_ml0` is an initial estimation of the rate performed using `bg.exp_fit()` and a fixed threshold (default 250us).
- **F_bg** (*float*) – when `tail_min_us` is 'auto', is the factor by which the initial background estimation if multiplied to compute the threshold.
- **error_metrics** (*string*) – Specifies the error metric to use. See *fretbursts.background.exp_fit()* for more details.
- **fit_allph** (*bool*) – if True (default) the background for the all-photon is fitted. If False it is computed as the sum of backgrounds in all the other streams.

The background estimation functions are defined in the module `background` (conventionally imported as `bg`).

---

**Example**

Compute background with `bg.exp_fit` (inter-photon delays MLE tail fitting), every 30s, with automatic tail-threshold:

```
d.calc_bg(bg.exp_fit, time_s=20, tail_min_us='auto')
```

---

**Returns**
None, all the results are saved in the object itself.

`burst_search`(*L=None*, *m=10*, *F=6.0*, *P=None*, *min_rate_cps=None*, *ph_sel=Ph_sel(Dex='DAem', Aex='DAem')*, *compact=False*, *index_allph=True*, *c=-1*, *computefret=True*, *max_rate=False*, *dither=False*, *pure_python=False*, *verbose=False*, *mute=False*, *pax=False*)

Performs a burst search with specified parameters.

This method performs a sliding-window burst search without binning the timestamps. The burst starts when the rate of `m` photons is above a minimum rate, and stops when the rate falls below the threshold. The result of the burst search is stored in the `mburst` attribute (a list of Bursts objects, one per channel) containing start/stop times and indexes. By default, after burst search, this method computes donor and acceptor counts, it applies burst corrections (background, leakage, etc…) and computes E (and S in case of ALEX). You can skip these steps by passing `computefret=False`.

The minimum rate can be explicitly specified with the `min_rate_cps` argument, or computed as a function of the background rate with the `F` argument.

**Parameters**

- **m** (*int*) – number of consecutive photons used to compute the photon rate. Typical values 5-20. Default 10.
- **L** (*int or None*) – minimum number of photons in burst. If None (default) L = m is used.
- **F** (*float*) – defines how many times higher than the background rate is the minimum rate used for burst search (`min rate = F * bg. rate`), assuming that `P = None` (default). Typical values are 3-9. Default 6.
- **P** (*float*) – threshold for burst detection expressed as a probability that a detected bursts is not due to a Poisson background. If not None, `P` overrides `F`. Note that the background process is experimentally super-Poisson so this probability is not physically very meaningful. Using this argument is discouraged.
- **min_rate_cps** (*float or list/array*) – minimum rate in cps for burst start. If not None, it has the precedence over `P` and `F`. If non-scalar, contains one rate per each multispot channel. Typical values range from 20e3 to 100e3.
- **ph_sel** (*Ph_sel object*) – defines the "photon selection" (or stream) to be used for burst search. Default: all photons. See `fretbursts.ph_sel` for details.
- **compact** (*bool*) – if True, a photon selection of only one excitation period is required and the timestamps are "compacted" by removing the "gaps" between each excitation period.
- **index_allph** (*bool*) – if True (default), the indexes of burst start and stop (`istart`, `istop`) are relative to the full timestamp array. If False, the indexes are relative to timestamps selected by the `ph_sel` argument.
- **c** (*float*) – correction factor used in the rate vs time-lags relation. `c` affects the computation of the burst-search parameter `T`. When `F` is not None, `T = (m - 1 - c) / (F * bg_rate)`. When using `min_rate_cps`, `T = (m - 1 - c) / min_rate_cps`.
- **computefret** (*bool*) – if True (default) compute donor and acceptor counts, apply corrections (background, leakage, direct excitation) and compute E (and S). If False, skip all these steps and stop just after the initial burst search.
- **max_rate** (*bool*) – if True compute the max photon rate inside each burst using the same `m` used for burst search. If False (default) skip this step.
- **dither** (*bool*) – if True applies dithering corrections to burst counts. Default False. See `Data.dither()`.

- **pure_python** (*bool*) – if True, uses the pure python functions even when optimized Cython functions are available.
- **pax** (*bool*) – this has effect only if measurement is PAX. In this case, when True computes E using a PAX-enhanced formula: `(2 na) / (2 na + nd + nda)`. Otherwise use the usual usALEX formula: `na / na + nd`. Quantities `nd`/`na` are D/A burst counts during D excitation period, while `nda` is D emission during A excitation period.

**Note:** when using `P` or `F` the background rates are needed, so `.calc_bg()` must be called before the burst search.

**Example**

d.burst_search(m=10, F=6)

**Returns**
None, all the results are saved in the `Data` object.

`calc_fret`(*count_ph=False*, *corrections=True*, *dither=False*, *mute=False*, *pure_python=False*, *pax=False*)

Compute FRET (and stoichiometry if ALEX) for each burst.

This is an high-level functions that can be run after burst search. By default, it will count Donor and Acceptor photons, perform corrections (background, leakage), and compute gamma-corrected FRET efficiencies (and stoichiometry if ALEX).

**Parameters**

- **count_ph** (*bool*) – if True (default), calls `calc_ph_num()` to counts Donor and Acceptor photons in each bursts
- **corrections** (*bool*) – if True (default), applies background and bleed-through correction to burst data
- **dither** (*bool*) – whether to apply dithering to burst size. Default False.
- **mute** (*bool*) – whether to mute all the printed output. Default False.
- **pure_python** (*bool*) – if True, uses the pure python functions even when the optimized Cython functions are available.
- **pax** (*bool*) – this has effect only if measurement is PAX. In this case, when True computes E using a PAX-enhanced formula: `(2 na) / (2 na + nd + nda)`. Otherwise use the usual usALEX formula: `na / na + nd`. Quantities `nd`/`na` are D/A burst counts during D excitation period, while `nda` is D emission during A excitation period.

**Returns**
None, all the results are saved in the object.

`calc_ph_num`(*alex_all=False*, *pure_python=False*)

Computes number of D, A (and AA) photons in each burst.

**Parameters**

- **alex_all** (*bool*) – if True and self.ALEX is True, computes also the donor channel photons during acceptor excitation (`nda`)
- **pure_python** (*bool*) – if True, uses the pure python functions even when the optimized Cython functions are available.

**Returns**
Saves nd, na, nt (and eventually naa, nda) in self. Returns None.

**fuse_bursts**(*ms=0*, *process=True*, *mute=False*)

Return a new *Data* object with nearby bursts fused together.

**Parameters**

- **ms** (*float*) – fuse all burst separated by less than ms millisecs. If < 0 no burst is fused. Note that with ms = 0, overlapping bursts are fused.
- **process** (*bool*) – if True (default), reprocess the burst data in the new object applying corrections and computing FRET.
- **mute** (*bool*) – if True suppress any printed output.

**calc_sbr**(*ph_sel=Ph_sel(Dex='DAem', Aex='DAem')*, *gamma=1.0*)

Return Signal-to-Background Ratio (SBR) for each burst.

**Parameters**

- **ph_sel** (*Ph_sel object*) – object defining the photon selection for which to compute the sbr. Changes the photons used for burst size and the corresponding background rate. Valid values here are Ph_sel('all'), Ph_sel(Dex='Dem'), Ph_sel(Dex='Aem'). See *fretbursts.ph_sel* for details.
- **gamma** (*float*) – gamma value used to compute corrected burst size in the case ph_sel is Ph_sel('all'). Ignored otherwise.

**Returns**
A list of arrays (one per channel) with one value per burst. The list is also saved in sbr attribute.

**calc_max_rate**(*m*, *ph_sel=Ph_sel(Dex='DAem', Aex='DAem')*, *compact=False*, *c=1*)

Compute the max m-photon rate reached in each burst.

**Parameters**

- **m** (*int*) – number of timestamps to use to compute the rate. As for burst search, typical values are 5-20.
- **ph_sel** (*Ph_sel object*) – object defining the photon selection. See *fretbursts.ph_sel* for details.
- **c** (*float*) – this parameter is used in the definition of the rate estimator which is `(m - 1 - c) / t[last] - t[first]`. For more details see *phtools.phrates.mtuple_rates()*.

### 3.2.4 Burst corrections

#### Correction factors

The following are the various burst correction factors. They are Data properties, so setting their value automatically updates all the burst quantities (including E and S).

**class** fretbursts.burstlib.**Data**

**gamma**

Gamma correction factor (compensates DexDem and DexAem unbalance).

**leakage**
Spectral leakage (bleed-through) of D emission in the A channel.

**dir_ex**
Direct excitation correction factor.

**chi_ch**
Per-channel relative gamma factor.

### Correction methods

List of `Data` methods used to apply burst corrections.

`class fretbursts.burstlib.Data`

**background_correction**(*relax_nt=False*, *mute=False*)
Apply background correction to burst sizes (nd, na,...)

**leakage_correction**(*mute=False*)
Apply leakage correction to burst sizes (nd, na,...)

**dither**(*lsb=2*, *mute=False*)
Add dithering (uniform random noise) to burst counts (nd, na,...).

The dithering amplitude is the range -0.5*lsb .. 0.5*lsb.

## 3.2.5 Burst selection methods

`Data` methods that allow to filter bursts according to different rules. See also *Burst selection*.

`class fretbursts.burstlib.Data`

**select_bursts**(*filter_fun*, *negate=False*, *computefret=True*, *args=None*, ***kwargs*)
Return an object with bursts filtered according to `filter_fun`.

This is the main method to select bursts according to different criteria. The selection rule is defined by the selection function `filter_fun`. FRETBursts provides a several predefined selection functions see *Burst selection*. New selection functions can be defined and passed to this method to implement arbitrary selection rules.

**Parameters**

- **filter_fun** (*function*) – function used for burst selection
- **negate** (*boolean*) – If True, negates (i.e. take the complementary) of the selection returned by `filter_fun`. Default `False`.
- **computefret** (*boolean*) – If True (default) recompute donor and acceptor counts, corrections and FRET quantities (i.e. E, S) in the new returned object.
- **args** (*tuple or None*) – positional arguments for `filter_fun()`

**kwargs:**
Additional keyword arguments passed to `filter_fun()`.

**Returns**
A new `Data` object containing only the selected bursts.

**Note:** In order to save RAM, the timestamp arrays (`ph_times_m`) of the new Data() points to the same arrays of the original Data(). Conversely, all the bursts data (`mburst`, `nd`, `na`, etc...) are new distinct objects.

**select_bursts_mask**(*filter_fun*, *negate=False*, *return_str=False*, *args=None*, ***kwargs*)

Returns mask arrays to select bursts according to `filter_fun`.

The function `filter_fun` is called to compute the mask arrays for each channel.

This method is useful when you want to apply a selection from one object to a second object. Otherwise use `Data.select_bursts()`.

**Parameters**

- **filter_fun** (*function*) – function used for burst selection
- **negate** (*boolean*) – If True, negates (i.e. take the complementary) of the selection returned by `filter_fun`. Default `False`.
- **return_str** – if True return, for each channel, a tuple with a bool array and a string that can be added to the measurement name to indicate the selection. If False returns only the bool array. Default False.
- **args** (*tuple or None*) – positional arguments for `filter_fun()`

**kwargs:**

Additional keyword arguments passed to `filter_fun()`.

**Returns**

A list of boolean arrays (one per channel) that define the burst selection. If `return_str` is True returns a list of tuples, where each tuple is a bool array and a string.

**See also:**

`Data.select_bursts()`, `Data.select_bursts_mask_apply()`

**select_bursts_mask_apply**(*masks*, *computefret=True*, *str_sel=''*)

Returns a new Data object with bursts selected according to `masks`.

This method select bursts using a list of boolean arrays as input. Since the user needs to create the boolean arrays first, this method is useful when experimenting with new selection criteria that don't have a dedicated selection function. Usually, however, it is easier to select bursts through `Data.select_bursts()` (using a selection function).

**Parameters**

- **masks** (*list of arrays*) – each element in this list is a boolean array that selects bursts in a channel.
- **computefret** (*boolean*) – If True (default) recompute donor and acceptor counts, corrections and FRET quantities (i.e. E, S) in the new returned object.

**Returns**

A new `Data` object containing only the selected bursts.

**Note:** In order to save RAM, the timestamp arrays (`ph_times_m`) of the new Data() points to the same arrays of the original Data(). Conversely, all the bursts data (`mburst`, `nd`, `na`, etc...) are new distinct objects.

**See also:**

*Data.select_bursts()*, Data.select_mask()

### 3.2.6 Fitting methods

Some fitting methods for burst data. Note that E and S histogram fitting with generic models is now handled with the new *fitting framework*.

**class** fretbursts.burstlib.**Data**

**fit_E_generic**(*E1=-1*, *E2=2*, *fit_fun=<function two_gaussian_fit_hist>*, *weights=None*, *gamma=1.0*, ***fit_kwargs*)

Fit E in each channel with `fit_fun` using burst in [E1,E2] range. All the fitting functions are defined in *fretbursts.fit.gaussian_fitting*.

**Parameters**

- **weights** (*string or None*) – specifies the type of weights If not None `weights` will be passed to `fret_fit.get_weights()`. `weights` can be not-None only when using fit functions that accept weights (the ones ending in `_hist` or `_EM`)
- **gamma** (*float*) – passed to `fret_fit.get_weights()` to compute weights

All the additional arguments are passed to `fit_fun`. For example `p0` or `mu_fix` can be passed (see `fit.gaussian_fitting` for details).

**Note:** Use this method for CDF/PDF or hist fitting. For EM fitting use *fit_E_two_gauss_EM()*.

**fit_E_m**(*E1=-1*, *E2=2*, *weights='size'*, *gamma=1.0*)

Fit E in each channel with the mean using bursts in [E1,E2] range.

**Note:** This two fitting are equivalent (but the first is much faster):

```
fit_E_m(weights='size')
fit_E_minimize(kind='E_size', weights='sqrt')
```

However `fit_E_minimize()` does not provide a model curve.

**fit_E_ML_poiss**(*E1=-1*, *E2=2*, *method=1*, ***kwargs*)

ML fit for E modeling size ~ Poisson, using bursts in [E1,E2] range.

**fit_E_minimize**(*kind='slope'*, *E1=-1*, *E2=2*, ***kwargs*)

Fit E using method `kind` ('slope' or 'E_size') and bursts in [E1,E2] If `kind` is 'slope' the fit function is fret_fit.fit_E_slope() If `kind` is 'E_size' the fit function is fret_fit.fit_E_E_size() Additional arguments in `kwargs` are passed to the fit function.

**fit_E_two_gauss_EM**(*fit_func=<function two_gaussian_fit_EM>*, *weights='size'*, *gamma=1.0*, ***kwargs*)

Fit the E population to a Gaussian mixture model using EM method. Additional arguments in `kwargs` are passed to the fit_func().

### 3.2.7 Timestamp access methods

The following methods are used to access (or iterate over) the arrays of timestamps (for different photon streams), timestamps masks and burst data.

- *Data.get_ph_times()*
- *Data.ph_in_bursts_ich()*
- *Data.ph_in_bursts_mask_ich()*
- *Data.iter_ph_times()*
- *Data.get_ph_mask()*
- *Data.iter_ph_masks()*
- *Data.iter_bursts_ph()*
- *Data.expand()*
- *Data.copy()*
- *Data.slice_ph()*

The methods documentation follows:

class fretbursts.burstlib.**Data**

**get_ph_times**(*ich=0*, *ph_sel=Ph_sel(Dex='DAem', Aex='DAem')*, *compact=False*)

Returns the timestamps array for channel `ich`.

This method always returns in-memory arrays, even when ph_times_m is a disk-backed list of arrays.

**Parameters**

- **ph_sel** (*Ph_sel object*) – object defining the photon selection. See *fretbursts.ph_sel* for details.
- **compact** (*bool*) – if True, a photon selection of only one excitation period is required and the timestamps are "compacted" by removing the "gaps" between each excitation period.

**iter_ph_times**(*ph_sel=Ph_sel(Dex='DAem', Aex='DAem')*, *compact=False*)

Iterator that returns the arrays of timestamps in `.ph_times_m`.

**Parameters**

**Same arguments as :meth:`get_ph_mask` except for `ich`.**

**get_ph_mask**(*ich=0*, *ph_sel=Ph_sel(Dex='DAem', Aex='DAem')*)

Returns a mask for `ph_sel` photons in channel `ich`.

The masks are either boolean arrays or slices (full or empty). In both cases they can be used to index the timestamps of the corresponding channel.

**Parameters**

**ph_sel** (*Ph_sel object*) – object defining the photon selection. See *fretbursts.ph_sel* for details.

**iter_ph_masks**(*ph_sel=Ph_sel(Dex='DAem', Aex='DAem')*)

Iterator returning masks for `ph_sel` photons.

**Parameters**

**ph_sel** (*Ph_sel object*) – object defining the photon selection. See *fretbursts.ph_sel* for details.

`iter_bursts_ph`(*ich=0*)

Iterate over (start, stop) indexes to slice photons for each burst.

`ph_in_bursts_ich`(*ich=0*, *ph_sel=Ph_sel(Dex='DAem', Aex='DAem')*)

Return timestamps of photons inside bursts for channel `ich`.

**Returns**
Array of photon timestamps in channel `ich` and photon selection `ph_sel` that are inside any burst.

`ph_in_bursts_mask_ich`(*ich=0*, *ph_sel=Ph_sel(Dex='DAem', Aex='DAem')*)

Return mask of all photons inside bursts for channel `ich`.

**Returns**
Boolean array for photons in channel `ich` and photon selection `ph_sel` that are inside any burst.

`expand`(*ich=0*, *alex_naa=False*, *width=False*)

Return per-burst D and A sizes (nd, na) and their background counts.

This method returns for each bursts the corrected signal counts and background counts in donor and acceptor channels. Optionally, the burst width is also returned.

**Parameters**

- **ich** (*int*) – channel for the bursts (can be not 0 only in multi-spot)
- **alex_naa** (*bool*) – if True and self.ALEX, returns burst sizes and background also for acceptor photons during accept. excitation
- **width** (*bool*) – whether return the burst duration (in seconds).

**Returns**
*List of arrays* – nd, na, donor bg, acceptor bg. If `alex_naa` is True returns: nd, na, naa, bg_d, bg_a, bg_aa. If `width` is True returns the bursts duration (in sec.) as last element.

`copy`(*mute=False*)

Copy data in a new object. All arrays copied except for ph_times_m

`slice_ph`(*time_s1=0*, *time_s2=None*, *s='slice'*)

Return a new Data object with ph in [`time_s1`,`time_s2`] (seconds)

If ALEX, this method must be called right after `fretbursts.loader.alex_apply_periods()` (with `delete_ph_t=True`) and before any background estimation or burst search.

## 3.3 Photon selections

In this module we define the class *Ph_sel* used to specify a "selection" of a sub-set of photons/timestamps (i.e. all-photons, Donor-excitation-period photons, etc…).

A photon selection is one of the base *photon streams* or a combination of them. Base *photon streams* are photon from the donor (or acceptor) emission channel detected during the donor (or acceptor) excitation period. For non-ALEX data there is only the donor excitation period.

The following table shows base *photon streams* for smFRET data (non-ALEX):

| Photon selection | Syntax |
|---|---|
| D-emission | `Ph_sel(Dex='Dem')` |
| A-emission | `Ph_sel(Dex='Aem')` |

and for ALEX data:

| Photon selection | Syntax |
|---|---|
| D-emission during D-excitation | `Ph_sel(Dex='Dem')` |
| A-emission during D-excitation | `Ph_sel(Dex='Aem')` |
| D-emission during A-excitation | `Ph_sel(Aex='Dem')` |
| A-emission during A-excitation | `Ph_sel(Aex='Aem')` |

Additionally, all the photons can be selected with `Ph_sel('all')` (that is a shortcut for `Ph_sel(Dex='DAem', Aex='DAem')`).

---

**Examples**

- `Ph_sel(Dex='DAem', Aex='DAem')` or `Ph_sel('all')` select all photons.
- `Ph_sel(Dex='DAem')` selects only donor and acceptor photons emitted during donor excitation. These are all the photons for non-ALEX data.
- `Ph_sel(Dex='Aem', Aex='Aem')` selects all the photons detected from the acceptor-emission channel.

---

The documentation for the *Ph_sel* class follows.

**class** `fretbursts.ph_sel.`**`Ph_sel`**(*Dex=None*, *Aex=None*)

Class that describes a selection of photons.

This class takes two arguments `Dex` and `Aex`. Valid values for the arguments are the strings 'DAem', 'Dem', 'Aem' or `None`. These values select, respectively, donor+acceptor, donor-only, acceptor-only or no photons during an excitation period (`Dex` or `Aex`).

The class must be called with at least one keyword argument or using the string 'all' as the only argument. Calling `Ph_sel('all')` is equivalent to `Ph_sel(Dex='DAem', Aex='DAem')`. Not specifying a keyword argument is equivalent to setting it to None.

## 3.4 Background estimation

### 3.4.1 background.py

Routines to compute the background from an array of timestamps. This module is normally imported as `bg` when fretbursts is imported.

The important functions are *exp_fit()* and *exp_cdf_fit()* that provide two (fast) algorithms to estimate the background without binning. These functions are not usually called directly but passed to `Data.calc_bg()` to compute the background of a measurement.

See also *exp_hist_fit()* for background estimation using an histogram fit.

`fretbursts.background.`**`exp_fit`**(*ph*, *tail_min_us=None*, *clk_p=1.25e-08*, *error_metrics=None*)

Return a background rate using the MLE of mean waiting-times.

Compute the background rate, selecting waiting-times (delays) larger than a minimum threshold.

This function performs a Maximum Likelihood (ML) fit. For exponentially-distributed waiting-times this is the empirical mean.

**Parameters**

- **ph** (*array*) – timestamps array from which to extract the background
- **tail_min_us** (*float*) – minimum waiting-time in micro-secs
- **clk_p** (*float*) – clock period for timestamps in `ph`
- **error_metrics** (*string or None*) – Valid values are 'KS' or 'CM'. 'KS' (Kolmogorov-Smirnov statistics) computes the error as the max of deviation of the empirical CDF from the fitted CDF. 'CM' (Crames-von Mises) uses the L^2 distance. If None, no error metric is computed (returns None).

**Returns**
*2-Tuple* – Estimated background rate in cps, and a "quality of fit" index (the lower the better) according to the chosen metric. If error_metrics==None, the returned "quality of fit" is None.

**See also:**

*exp_cdf_fit()*, *exp_hist_fit()*

`fretbursts.background.exp_cdf_fit`(*ph*, *tail_min_us=None*, *clk_p=1.25e-08*, *error_metrics=None*)

Return a background rate fitting the empirical CDF of waiting-times.

Compute the background rate, selecting waiting-times (delays) larger than a minimum threshold.

This function performs a least square fit of an exponential Cumulative Distribution Function (CDF) to the empirical CDF of waiting-times.

**Parameters**

- **ph** (*array*) – timestamps array from which to extract the background
- **tail_min_us** (*float*) – minimum waiting-time in micro-secs
- **clk_p** (*float*) – clock period for timestamps in `ph`
- **error_metrics** (*string or None*) – Valid values are 'KS' or 'CM'. 'KS' (Kolmogorov-Smirnov statistics) computes the error as the max of deviation of the empirical CDF from the fitted CDF. 'CM' (Crames-von Mises) uses the L^2 distance. If None, no error metric is computed (returns None).

**Returns**
*2-Tuple* – Estimated background rate in cps, and a "quality of fit" index (the lower the better) according to the chosen metric. If error_metrics==None, the returned "quality of fit" is None.

**See also:**

*exp_fit()*, *exp_hist_fit()*

`fretbursts.background.exp_hist_fit`(*ph*, *tail_min_us*, *binw=5e-05*, *clk_p=1.25e-08*, *weights='hist_counts'*, *error_metrics=None*)

Compute background rate with WLS histogram fit of waiting-times.

Compute the background rate, selecting waiting-times (delays) larger than a minimum threshold.

This function performs a Weighed Least Squares (WLS) fit of the histogram of waiting times to an exponential decay.

**Parameters**

- **ph** (*array*) – timestamps array from which to extract the background
- **tail_min_us** (*float*) – minimum waiting-time in micro-secs
- **binw** (*float*) – bin width for waiting times, in seconds.
- **clk_p** (*float*) – clock period for timestamps in `ph`

- **weights** (*None or string*) – if None no weights is applied. if is 'hist_counts', each bin has a weight equal to its counts if is 'inv_hist_counts', the weight is the inverse of the counts.
- **error_metrics** (*string or None*) – Valid values are 'KS' or 'CM'. 'KS' (Kolmogorov-Smirnov statistics) computes the error as the max of deviation of the empirical CDF from the fitted CDF. 'CM' (Crames-von Mises) uses the L^2 distance. If None, no error metric is computed (returns None).

**Returns**
*2-Tuple* – Estimated background rate in cps, and a "quality of fit" index (the lower the better) according to the chosen metric. If error_metrics==None, the returned "quality of fit" is None.

**See also:**

`exp_fit()`, `exp_cdf_fit()`

### 3.4.2 Low-level background fit functions

Generic functions to fit exponential populations.

These functions can be used directly, or, in a typical FRETBursts workflow they are passed to higher level methods.

*See also:*

- *Background estimation*

`fretbursts.fit.exp_fitting.`**`expon_fit`**`(`*s*, *s_min=0*, *offset=0.5*, *calc_residuals=True*`)`

Fit sample `s` to an exponential distribution using the ML estimator.

This function computes the rate (Lambda) using the maximum likelihood (ML) estimator of the mean waiting-time (Tau), that for an exponentially distributed sample is the sample-mean.

**Parameters**

- **s** (*array*) – array of exponetially-distributed samples
- **s_min** (*float*) – all samples < `s_min` are discarded (`s_min` must be >= 0).
- **offset** (*float*) – offset for computing the CDF. See `get_ecdf()`.
- **calc_residuals** (*bool*) – if True compute the residuals of the fitted exponential versus the empirical CDF.

**Returns**
A 4-tuple of the fitted rate (1/life-time), residuals array, residuals x-axis array, sample size after threshold.

`fretbursts.fit.exp_fitting.`**`expon_fit_cdf`**`(`*s*, *s_min=0*, *offset=0.5*, *calc_residuals=True*`)`

Fit of an exponential model to the empirical CDF of `s`.

This function computes the rate (Lambda) fitting a line (linear regression) to the log of the empirical CDF.

**Parameters**

- **s** (*array*) – array of exponetially-distributed samples
- **s_min** (*float*) – all samples < `s_min` are discarded (`s_min` must be >= 0).
- **offset** (*float*) – offset for computing the CDF. See `get_ecdf()`.
- **calc_residuals** (*bool*) – if True compute the residuals of the fitted exponential versus the empirical CDF.

**Returns**
A 4-tuple of the fitted rate (1/life-time), residuals array, residuals x-axis array, sample size after threshold.

`fretbursts.fit.exp_fitting.`**`expon_fit_hist`**(*s*, *bins*, *s_min=0*, *weights=None*, *offset=0.5*, *calc_residuals=True*)

Fit of an exponential model to the histogram of `s` using least squares.

**Parameters**

- **s** (*array*) – array of exponetially-distributed samples
- **bins** (*float or array*) – if float is the bin width, otherwise is the array of bin edges (passed to `numpy.histogram`)
- **s_min** (*float*) – all samples < `s_min` are discarded (`s_min` must be >= 0).
- **weights** (*None or string*) – if None no weights is applied. if is 'hist_counts', each bin has a weight equal to its counts if is 'inv_hist_counts', the weight is the inverse of the counts.
- **offset** (*float*) – offset for computing the CDF. See `get_ecdf()`.
- **calc_residuals** (*bool*) – if True compute the residuals of the fitted exponential versus the empirical CDF.

**Returns**
A 4-tuple of the fitted rate (1/life-time), residuals array, residuals x-axis array, sample size after threshold.

`fretbursts.fit.exp_fitting.`**`get_ecdf`**(*s*, *offset=0.5*)

Return arrays (x, y) for the empirical CDF curve of sample `s`.

See the code for more info (is a one-liner!).

**Parameters**

- **s** (*array of floats*) – sample
- **offset** (*float, default 0.5*) – Offset to add to the y values of the CDF

**Returns**
*(x, y) (tuple of arrays)* – the x and y values of the empirical CDF

`fretbursts.fit.exp_fitting.`**`get_residuals`**(*s*, *tau_fit*, *offset=0.5*)

Returns residuals of sample `s` CDF vs an exponential CDF.

**Parameters**

- **s** (*array of floats*) – sample
- **tau_fit** (*float*) – mean waiting-time of the exponential distribution to use as reference
- **offset** (*float*) – Default 0.5. Offset to add to the empirical CDF. See `get_ecdf()` for details.

**Returns**
*residuals (array)* – residuals of empirical CDF compared with analytical CDF with time constant `tau_fit`.

## 3.5 Burst selection

After performing a burst search is common to select bursts according to different criteria (burst size, FRET efficiency, etc…).

In FRETBursts this can be easily accomplished using the method *Data.select_bursts()*. This method takes a *selection function* as parameters. *Data.select_bursts()* returns a new *Data* object containing only the new sub-set of bursts. A new selection can be applied to this new object as well. In this way, different selection criteria can be freely combined in order to obtain a burst population satisfying arbitrary constrains.

FRETBursts provides a large number of *selection functions*. Moreover, creating a new selection function is extremely simple, requiring (usually) 2-3 lines of code. You can take the functions in `select_bursts.py` as examples to create your own selection rule.

In the next section we list all the selection functions. You may also want to check the *Data* methods that deal with burst selection:

- *Data.select_bursts()*
- *Data.select_bursts_mask()*
- *Data.select_bursts_mask_apply()*

### 3.5.1 Selection functions

The module `select_bursts` defines functions to select bursts according to different criteria.

These functions are usually passed to *Data.select_bursts()*. For example:

```
ds = d.select_bursts(select_bursts.E, th1=0.2, th2=0.6)
```

returns a new object `ds` containing only the bursts of `d` that pass the specified selection criterium (`E` between 0.2 and 0.6 in this case).

`fretbursts.select_bursts.`**`E`**(*d*, *ich=0*, *E1=-inf*, *E2=inf*)

Select bursts with E between E1 and E2.

`fretbursts.select_bursts.`**`ES`**(*d*, *ich=0*, *E1=-inf*, *E2=inf*, *S1=-inf*, *S2=inf*, *rect=True*)

Select bursts with E between `E1` and `E2` and S between `S1` and `S2`.

When `rect` is True the selection is rectangular otherwise is elliptical.

**See also:**

For plotting the ES region selected by (`E1`, `E2`, `S1`, `S2`, `rect`):

- *fretbursts.burst_plot.plot_ES_selection()*

`fretbursts.select_bursts.`**`ES_ellips`**(*d*, *ich=0*, *E1=-1000.0*, *E2=1000.0*, *S1=-1000.0*, *S2=1000.0*)

Select bursts with E-S inside an ellipsis inscribed in E1, E2, S1, S2.

`fretbursts.select_bursts.`**`ES_rect`**(*d*, *ich=0*, *E1=-inf*, *E2=inf*, *S1=-inf*, *S2=inf*)

Select bursts inside the rectangle defined by E1, E2, S1, S2.

`fretbursts.select_bursts.`**`brightness`**(*d*, *ich=0*, *th1=0*, *th2=inf*, *add_naa=False*, *gamma=1*, *beta=1*, *donor_ref=True*)

Select bursts with size/width between th1 and th2 (cps).

`fretbursts.select_bursts.consecutive`(*d*, *ich=0*, *th1=0*, *th2=inf*, *kind='both'*)

Select consecutive bursts with th1 <= separation <= th2 (in sec.).

**Parameters**

**kind** (*string*) – valid values are 'first' to select the first burst of each pair, 'second' to select the second burst of each pair and 'both' to select both bursts in each pair.

`fretbursts.select_bursts.na`(*d*, *ich=0*, *th1=20*, *th2=inf*)

Select bursts with (na >= th1) and (na <= th2).

`fretbursts.select_bursts.na_bg`(*d*, *ich=0*, *F=5*)

Select bursts with (na >= bg_ad*F).

`fretbursts.select_bursts.na_bg_p`(*d*, *ich=0*, *P=0.05*, *F=1.0*)

Select bursts w/ AD signal using P{F*BG>=na} < P.

`fretbursts.select_bursts.naa`(*d*, *ich=0*, *th1=20*, *th2=inf*, *gamma=1.0*, *beta=1.0*, *donor_ref=True*, *naa_comp=False*, *naa_aexonly=True*)

Select bursts with (naa >= th1) and (naa <= th2).

The `naa` quantity can be optionally corrected using gamma and beta factors.

**Parameters**

- **th1, th2** (*floats*) – lower (`th1`) and upper (`th2`) bounds for selecting `naa`. By default `th2 = inf` (i.e. no upper limit).
- **gamma, beta** (*floats*) – arguments used to compute gamma- and beta-corrected burst sizes. See *fretbursts.burstlib.Data.burst_sizes_ich()* for details.
- **donor_ref** (*bool*) – Select the convention for `naa` correction. If True (default), uses `naa / (beta * gamma)`. Otherwise, uses `naa / beta`. It is suggested to use the same `donor_ref` convention when combining `Dex size` and `naa` burst selections so that the thresholds values of the two selections will be commensurable.
- **na_comp** (*bool*) – **[PAX-only]** If True, multiply the `na` term by `(1 + Wa/Wd)`, where Wa and Wd are the D and A alternation durations (typically Wa/Wd = 1).
- **naa_aexonly** (*bool*) – **[PAX-only]** if True, the `naa` term is corrected to include only A emission due to A excitation. If False, the `naa` term includes all the counts in DAexAem. The `naa` term also depends on the `naa_comp` argument.
- **naa_comp** (*bool*) – **[PAX-only]** If True, multiplies the `naa` term by `(1 + Wa/Wd)` where Wa and Wd are the D and A alternation durations (typically Wa/Wd = 1). The `naa` term also depends on the `naa_aexonly` argument.

**See also:**

- *fretbursts.burstlib.Data.burst_sizes_pax_ich()*.

`fretbursts.select_bursts.naa_bg`(*d*, *ich=0*, *F=5*)

Select bursts with (naa >= bg_aa*F).

`fretbursts.select_bursts.naa_bg_p`(*d*, *ich=0*, *P=0.05*, *F=1.0*)

Select bursts w/ AA signal using P{F*BG>=naa} < P.

`fretbursts.select_bursts.nd`(*d*, *ich=0*, *th1=20*, *th2=inf*)

Select bursts with (nd >= th1) and (nd <= th2).

`fretbursts.select_bursts.`**`nd_bg`**(*d*, *ich=0*, *F=5*)

Select bursts with (nd >= bg_dd*F).

`fretbursts.select_bursts.`**`nd_bg_p`**(*d*, *ich=0*, *P=0.05*, *F=1.0*)

Select bursts w/ DD signal using P{F*BG>=nd} < P.

`fretbursts.select_bursts.`**`nda_percentile`**(*d*, *ich=0*, *q=50*, *low=False*, *gamma=1.0*, *add_naa=False*)

Select bursts with SIZE >= q-percentile (or <= if `low` is True)

`gamma` and `add_naa` are passed to *`fretbursts.burstlib.Data.burst_sizes_ich()`* to compute the burst size.

`fretbursts.select_bursts.`**`nt_bg`**(*d*, *ich=0*, *F=5*)

Select bursts with (nt >= bg*F).

`fretbursts.select_bursts.`**`nt_bg_p`**(*d*, *ich=0*, *P=0.05*, *F=1.0*)

Select bursts w/ signal using P{F*BG>=nt} < P.

`fretbursts.select_bursts.`**`peak_phrate`**(*d*, *ich=0*, *th1=0*, *th2=inf*)

Select bursts with peak phtotons rate between th1 and th2 (cps).

Note that this function requires to compute the peak photon rate first using *`fretbursts.burstlib.Data.calc_max_rate()`*.

`fretbursts.select_bursts.`**`period`**(*d*, *ich=0*, *bp1=0*, *bp2=None*)

Select bursts from period bp1 to period bp2 (included).

`fretbursts.select_bursts.`**`sbr`**(*d*, *ich=0*, *th1=0*, *th2=inf*)

Select bursts with SBR between `th1` and `th2`.

`fretbursts.select_bursts.`**`single`**(*d*, *ich=0*, *th=1*)

Select bursts that are at least th millisec apart from the others.

`fretbursts.select_bursts.`**`size`**(*d*, *ich=0*, *th1=20*, *th2=inf*, *add_naa=False*, *gamma=1.0*, *beta=1.0*, *donor_ref=True*, *ph_sel=None*, *naa_aexonly=False*, *naa_comp=False*, *na_comp=False*)

Select bursts with burst sizes (i.e. counts) between `th1` and `th2`.

The burst size is the number of photon in a burst. By default it includes all photons during donor excitation (`Dex`). To add *AexAem* photons to the burst size use `add_naa=True`. If `ph_sel` is specified use a PAX-specific definition of size as defined in *`fretbursts.burstlib.Data.burst_sizes_pax_ich()`*.

**Parameters**

- **d** (*Data object*) – the object containing the measurement.
- **ich** (*int*) – the spot number, only relevant for multi-spot. In single-spot data there is only CH-0 so this argument may be omitted. Default 0.
- **th1, th2** (*floats*) – select bursts with `th1 <= size <= th2`. Default `th2 = inf` (i.e. no upper limit).
- **add_naa** (*boolean*) – when True, add AexAem photons when computing burst burst size. Default False.
- **gamma, beta** (*floats*) – arguments used to compute gamma- and beta-corrected burst sizes. See *`fretbursts.burstlib.Data.burst_sizes_ich()`* for details.
- **donor_ref** (*bool*) – Select the convention for `naa` correction. See *`fretbursts.burstlib.Data.burst_sizes_ich()`* for details.

- **ph_sel** (*Ph_sel object or None*) – if not None, use PAX-specific burst size definition. ph_sel defines which terms are included in the burst size.
- **na_comp** (*bool*) – **[PAX-only]** If True, multiply the `na` term by `(1 + Wa/Wd)`, where Wa and Wd are the D and A alternation durations (typically Wa/Wd = 1).
- **naa_aexonly** (*bool*) – **[PAX-only]** if True, the `naa` term is corrected to include only A emission due to A excitation. If False, the `naa` term includes all the counts in DAexAem. The `naa` term also depends on the `naa_comp` argument.
- **naa_comp** (*bool*) – **[PAX-only]** If True, multiply the `naa` term by `(1 + Wa/Wd)` where Wa and Wd are the D and A alternation durations (typically Wa/Wd = 1). The `naa` term also depends on the `naa_aexonly` argument.

**Returns**
A tuple containing an array (the burst mask) and a string which briefly describes the selection.

**See also:**

- *`fretbursts.burstlib.Data.burst_sizes_ich()`*.
- *`fretbursts.burstlib.Data.burst_sizes_pax_ich()`*.

`fretbursts.select_bursts.`**`str_G`**`(`*gamma*, *donor_ref*`)`

A string indicating gamma value and convention for burst size correction.

`fretbursts.select_bursts.`**`time`**`(`*d*, *ich=0*, *time_s1=0*, *time_s2=None*`)`

Select the burst starting from time_s1 to time_s2 (in seconds).

`fretbursts.select_bursts.`**`topN_max_rate`**`(`*d*, *ich=0*, *N=500*`)`

Select `N` bursts with the highest max burst rate.

`fretbursts.select_bursts.`**`topN_nda`**`(`*d*, *ich=0*, *N=500*, *gamma=1.0*, *add_naa=False*`)`

Select the N biggest bursts in the channel.

`gamma` and `add_naa` are passed to *`fretbursts.burstlib.Data.burst_sizes_ich()`* to compute the burst size.

`fretbursts.select_bursts.`**`topN_sbr`**`(`*d*, *ich=0*, *N=200*`)`

Select the top `N` bursts with highest SBR.

`fretbursts.select_bursts.`**`width`**`(`*d*, *ich=0*, *th1=0.5*, *th2=inf*`)`

Select bursts with (width >= th1) and (width <= th2), in ms.

## 3.6 Fit framework

This page contains only a general description of FRETBursts fitting functionalities. The content of this page is:

**Contents**

For the reference documentation for fitting multi-channel histograms see:

## 3.6.1 MultiFitter reference documentation

This model provides a class for fitting multi-channel data (`MultiFitter`) and a series of predefined functions for common models used to fit E or S histograms.

**Contents**



### The MultiFitter class

**class** `fretbursts.mfit.`**`MultiFitter`**(*data_list*, *skip_ch=None*)

A class handling a list of 1-D datasets for histogramming, KDE, fitting.

This class takes a list of 1-D arrays of samples (such as E values per burst). The list contains one 1-D array for each channel in a multispot experiment. In single-spot experiments the list contains only one array of samples. For each dataset in the list, this class compute histograms, KDEs and fits (both histogram fit and KDE maximum). The list of datasets is stored in the attribute `data_list`. The histograms can be fitted with an arbitrary model (lmfit.Model). From KDEs the peak position in a range can be estimated.

Optionally weights can be assigned to each element in a dataset. To assign weights a user can assign the `.weights` attribute with a list of arrays; corresponding arrays in `.weights` and `.data_list` must have the same size.

Alternatively a function returning the weights can be used. In this case, the method `.set_weights_func` allows to set the function to be called to generate weights.

**`calc_kde`**(*bandwidth=0.03*, *calc_tot=True*)

Compute the list of kde functions and save it in `.kde`.

**`find_kde_max`**(*x_kde*, *xmin=None*, *xmax=None*, *calc_tot=True*)

Finds the peak position of kde functions between `xmin` and `xmax`.

Results are saved in the list `.kde_max_pos`.

**`fit_histogram`**(*model=None*, *pdf=True*, *fit_tot=True*, ***fit_kwargs*)

Fit the histogram of each channel using the same lmfit model.

A list of `lmfit.Minimizer` is stored in `.fit_res`. The fitted values for all the parameters and all the channels are save in a Pandas DataFrame `.params`.

**Parameters**

- **model** (*lmfit.Model object*) – lmfit Model with all the parameters already initialized used for fitting.

- **pdf** (*bool*) – if True fit the normalized histogram (.hist_pdf) otherwise fit the raw counts (.hist_counts).
- **fit_kwargs** (*dict*) – keyword arguments passed to `model().fit`.
- **fit_tot** (*bool*) – if True then fit the sum of the data as well

**histogram**(*binwidth=0.03*, *bins=None*, *verbose=False*, ***kwargs*)

Compute the histogram of the data for each channel.

If `bins` is None, `binwidth` determines the bins array (saved in `self.hist_bins`). If `bins` is not None, `binwidth` is ignored and `self.hist_binwidth` is computed from `self.hist_bins`.

The kwargs and `bins` are passed to `numpy.histogram`.

**set_weights_func**(*weight_func*, *weight_kwargs=None*)

Setup of the function returning the weights for each data-set.

To compute the weights for each dataset the `weight_func` is called multiple times. Keys in `weight_kwargs` are arguments of `weight_func`. Values in `weight_kwargs` are either scalars, in which case they are passed to `weight_func`, or lists. When an argument is a list, only one element of the list is passed each time.

**Parameters**

- **weight_func** (*function*) – function that returns the weights
- **weight_kwargs** (*dict*) – keyword arguments to be passed to `weight_func`.

### Model factory functions

In this section you find the documentation for the factory-functions that return pre-initialized models for fitting E and S data.

fretbursts.mfit.**factory_gaussian**(*center=0.1*, *sigma=0.1*, *amplitude=1*)

Return an lmfit Gaussian model that can be used to fit data.

Arguments are initial values for the model parameters.

**Returns**
An `lmfit.Model` object with all the parameters already initialized.

fretbursts.mfit.**factory_asym_gaussian**(*center=0.1*, *sigma1=0.1*, *sigma2=0.1*, *amplitude=1*)

Return a lmfit Asymmetric Gaussian model that can be used to fit data.

For the definition of asymmetric Gaussian see `asym_gaussian()`. Arguments are initial values for the model parameters.

**Returns**
An `lmfit.Model` object with all the parameters already initialized.

fretbursts.mfit.**factory_two_gaussians**(*add_bridge=False*, *p1_center=0.1*, *p2_center=0.9*, *p1_sigma=0.03*, *p2_sigma=0.03*)

Return a 2-Gaussian + (optional) bridge model that can fit data.

The optional "bridge" component (i.e. a plateau between the two peaks) is a function that is non-zero only between `p1_center` and `p2_center` and is defined as:

```
br_amplitude * (1 - g(x, p1_center, p1_sigma) - g(x, p1_center, p2_sigma))
```

where `g` is a gaussian function with amplitude = 1 and `br_amplitude` is the height of the plateau and the only additional parameter introduced by the bridge. Note that both centers and sigmas parameters in the bridge are the same ones of the adjacent Gaussian peaks. Therefore a 2-Gaussian + bridge model has 7 free parameters: 3 for each Gaussian and an additional one for the bridge. The bridge function is implemented in `bridge_function()`.

**Parameters**

- **p1_center, p2_center** (*float*) – initial values for the centers of the two Gaussian components.
- **p1_sigma, p2_sigma** (*float*) – initial values for the sigmas of the two Gaussian components.
- **add_bridge** (*bool*) – if True adds a bridge function between the two gaussian peaks. If False the model has only two Gaussians.

**Returns**
An `lmfit.Model` object with all the parameters already initialized.

`fretbursts.mfit.`**`factory_two_asym_gaussians`**(*add_bridge=False*, *p1_center=0.1*, *p2_center=0.9*, *p1_sigma=0.03*, *p2_sigma=0.03*)

Return a 2-Asym-Gaussians + (optional) bridge model that can fit data.

The Asym-Gaussian function is `asym_gaussian()`.

**Parameters**
**add_bridge** (*bool*) – if True adds a bridge function between the two gaussian peaks. If False the model has only two Asym-Gaussians.

The other arguments are initial values for the model parameters.

**Returns**
An `lmfit.Model` object with all the parameters already initialized.

`fretbursts.mfit.`**`factory_three_gaussians`**(*p1_center=0.0*, *p2_center=0.5*, *p3_center=1*, *sigma=0.05*)

Return a 3-Gaussian model that can fit data.

The other arguments are initial values for the `center` for each Gaussian component plus an single `sigma` argument that is used as initial sigma for all the Gaussians. Note that during the fitting the sigma of each Gaussian is varied independently.

**Returns**
An `lmfit.Model` object with all the parameters already initialized.

### Utility functions

The following functions are utility functions used to build the the model functions (i.e. the "factory functions") for the fitting.

`fretbursts.mfit.`**`bridge_function`**(*x*, *center1*, *center2*, *sigma1*, *sigma2*, *amplitude*)

A "bridge" function, complementary of two gaussian peaks.

Let `g` be a Gaussian function (with amplitude = 1), the bridge function is defined as:

```
amplitude * (1 - g(x, center1, sigma1) - g(x, center2, sigma2))
```

for `center1 < x < center2`. The function is 0 otherwise.

**Parameters**

- **x** (*array*) – 1-D array for the independent variable
- **center1** (*float*) – center of the first gaussian (left side)

- **center2** (*float*) – center of the second gaussian (right side)
- **sigma1** (*float*) – sigma of the left-side gaussian
- **sigma2** (*float*) – sigma of the right-side gaussian
- **amplitude** (*float*) – maximum (asymptotic) value of the bridge (plateau)

**Returns**
An array (same shape as `x`) with the function values.

`fretbursts.mfit.`**`asym_gaussian`**(*x*, *center*, *sigma1*, *sigma2*, *amplitude*)

A asymmetric gaussian function composed by two gaussian halves.

This function is composed from two gaussians joined at their peak, so that the left and right side decay with different sigmas.

**Parameters**

- **x** (*array*) – 1-D array for the independent variable
- **center** (*float*) – function peak position
- **sigma1** (*float*) – sigma of the left-side gaussian (for x < center)
- **sigma2** (*float*) – sigma of the right-side gaussian (for x > center)
- **amplitude** (*float*) – maximum value reach for x = center.

**Returns**
An array (same shape as `x`) with the function values.

### 3.6.2 Overview

FRETBursts uses of the powerful lmfit library for most fittings (like E or S histogram fitting). Lmfit should be automatically installed when installing FRETBursts, but in any case it is easily installable via `pip install lmfit`. For more installation info see *FRETBursts Installation*.

FRETBursts requires `lmfit` version 0.8 or higher.

### 3.6.3 Fitting E or S histograms

The module *`fretbursts.mfit`* provides a class *`fretbursts.mfit.MultiFitter`* that allow to build histograms and KDE on a multi-channel sample population (typically E or S values for each burst). The MultiFitter class can find the max peak position of a KDE or fit the histogram with an arbitrary model. A set of predefined models is provided to handle common cases. Sensible defaults are applied but the user can control every detail of the fit by setting initial values, parameter bounds (min, max), algebraic constrains and so on. New models can be created by composing simpler models (by using + operator). See the lmfit documentation for more info on how to define models and composite models.

A convenience function `fretbursts.burstlib_ext.burst_fitter()` can be used to create a `MultiFitter` object to fit either E or S. As an example let suppose having a measurement loaded in the variable `d`. To create a fitter object and compute the FRET histogram we execute:

```
bext.burst_fitter(d) # Creates d.E_fitter
d.E_fitter.histogram() # Compute the histogram for all the channels
```

Now we fit the E histogram with a 2-Gaussians model:

```
d.E_fitter.fit_histogram(mfit.factory_two_gaussians)
```

And plot the histogram and the fitted model:

```
dplot(d, hist_fret, show_model=True)
```

More detailed example can be found in the tutorials in notebooks on us-ALEX analysis.

### 3.6.4 Lmfit introduction

Lmfit provides a simple and flexible interface for non-linear least squares and other minimization methods. All the model parameters can be fixed/varied, have bounds (min, max) or constrained to an algebraic expression.

Moreover lmfit provides a Model class and a set of built-in models that allows to express curve-fitting problems in an compact and expressive form. Basic models (such as a Gaussian peak) and be composed allowing an easy definitions of a variety of models (2 or 3 Gaussians).

For more information refer to the official lmfit documentation.

### 3.6.5 Legacy Fit functions

A set of legacy functions used in versions of FRETBursts < 0.4 are defined in `fretbursts/fit`. This function are retained for backward compatibility but should not be used in new analysis.

These are low-level (i.e. generic) fit functions to fit gaussian or exponential models.

#### Gaussian fitting

This module provides functions to fit gaussian distributions and gaussian distribution mixtures (2 components). These functions can be used directly, or more often, in a typical FRETBursts workflow they are passed to higher level methods like `fretbursts.burstlib.Data.fit_E_generic()`.

Single Gaussian distribution fit:

- `gaussian_fit_hist()`
- `gaussian_fit_cdf()`
- `gaussian_fit_pdf()`

For 2-Gaussians fit we have the following models:

- `two_gauss_mix_pdf()`: *PDF of 2-components Gaussians mixture*
- `two_gauss_mix_ab()`: *linear combination of 2 Gaussians*

Main functions for mixture of 2 Gaussian distribution fit:

- `two_gaussian_fit_hist()` *histogram fit using `leastsq`*
- `two_gaussian_fit_hist_min()` *histogram fit using `minimize`*
- `two_gaussian_fit_hist_min_ab()` *the same but using _ab model*
- `two_gaussian_fit_cdf()` *curve fit of the CDF*
- `two_gaussian_fit_EM()` *Expectation-Maximization fit*
- `two_gaussian_fit_EM_b()` *the same with boundaries*

Also, some functions to fit 2-D gaussian distributions and mixtures are implemented but not thoroughly tested.

The reference documentation for **all** the functions follows.

`fretbursts.fit.gaussian_fitting.`**`bound_check`**(*val*, *bounds*)

Returns `val` clipped inside the interval `bounds`.

`fretbursts.fit.gaussian_fitting.`**`gaussian2d_fit`**(*sx*, *sy*, *guess=[0.5, 1]*)

2D-Gaussian fit of samples S using a fit to the empirical CDF.

`fretbursts.fit.gaussian_fitting.`**`gaussian_fit_cdf`**(*s*, *mu0=0*, *sigma0=1*, *return_all=False*, ***leastsq_kwargs*)

Gaussian fit of samples s fitting the empirical CDF. Additional kwargs are passed to the leastsq() function. If return_all=False then return only the fitted (mu,sigma) values If return_all=True (or full_output=True is passed to leastsq) then the full output of leastsq and the histogram is returned.

`fretbursts.fit.gaussian_fitting.`**`gaussian_fit_curve`**(*x*, *y*, *mu0=0*, *sigma0=1*, *a0=None*, *return_all=False*, ***kwargs*)

Gaussian fit of curve (x,y). If a0 is None then only (mu,sigma) are fitted (to a gaussian density). `kwargs` are passed to the leastsq() function.

If return_all=False then return only the fitted (mu,sigma) values If return_all=True (or full_output=True is passed to leastsq) then the full output of leastsq is returned.

`fretbursts.fit.gaussian_fitting.`**`gaussian_fit_hist`**(*s*, *mu0=0*, *sigma0=1*, *a0=None*, *bins=array([-0.5, -0.499, -0.498, ..., 1.497, 1.498, 1.499])*, *return_all=False*, *leastsq_kwargs={}*, *weights=None*, ***kwargs*)

Gaussian fit of samples s fitting the hist to a Gaussian function. If a0 is None then only (mu,sigma) are fitted (to a gaussian density). kwargs are passed to the histogram function. If return_all=False then return only the fitted (mu,sigma) values If return_all=True (or full_output=True is passed to leastsq) then the full output of leastsq and the histogram is returned. `weights` optional weights for the histogram.

`fretbursts.fit.gaussian_fitting.`**`gaussian_fit_ml`**(*s*, *mu_sigma_guess=[0.5, 1]*)

Gaussian fit of samples s using the Maximum Likelihood (ML method). Didactical, since scipy.stats.norm.fit() implements the same method.

`fretbursts.fit.gaussian_fitting.`**`gaussian_fit_pdf`**(*s*, *mu0=0*, *sigma0=1*, *a0=1*, *return_all=False*, *leastsq_kwargs={}*, ***kwargs*)

Gaussian fit of samples s using a fit to the empirical PDF. If a0 is None then only (mu,sigma) are fitted (to a gaussian density). `kwargs` are passed to get_epdf(). If return_all=False then return only the fitted (mu,sigma) values If return_all=True (or full_output=True is passed to leastsq) then the full output of leastsq and the PDF curve is returned.

`fretbursts.fit.gaussian_fitting.`**`get_epdf`**(*s*, *smooth=0*, *N=1000*, *smooth_pdf=False*, *smooth_cdf=True*)

Compute the empirical PDF of the sample `s`.

If smooth > 0 then apply a gaussian filter with sigma=smooth. N is the number of points for interpolation of the CDF on a uniform range.

`fretbursts.fit.gaussian_fitting.`**`normpdf`**(*x*, *mu=0*, *sigma=1.0*)

Return the normal pdf evaluated at `x`.

`fretbursts.fit.gaussian_fitting.`**`reorder_parameters`**(*p*)

Reorder 2-gauss mix params to have the 1st component with smaller mean.

`fretbursts.fit.gaussian_fitting.`**`reorder_parameters_ab`**(*p*)

Reorder 2-gauss mix params to have the 1st component with smaller mean.

`fretbursts.fit.gaussian_fitting.`**`two_gauss_mix_ab`**(*x*, *p*)

Mixture of two Gaussians with no area constrain.

`fretbursts.fit.gaussian_fitting.`**`two_gauss_mix_pdf`**(*x*, *p*)

PDF for the distribution of a mixture of two Gaussians.

`fretbursts.fit.gaussian_fitting.`**`two_gaussian2d_fit`**(*sx*, *sy*, *guess=[0.5, 1]*)

2D-Gaussian fit of samples S using a fit to the empirical CDF.

`fretbursts.fit.gaussian_fitting.`**`two_gaussian_fit_EM`**(*s*, *p0=[0, 0.1, 0.6, 0.1, 0.5]*, *max_iter=300*, *ptol=0.0001*, *fix_mu=[0, 0]*, *fix_sig=[0, 0]*, *debug=False*, *weights=None*)

Fit the sample s with two gaussians using Expectation Maximization.

This vesion allows to optionally fix mean or std. dev. of each component.

**Parameters**

- **s** (*array*) – population of samples to be fitted
- **p0** (*sequence-like*) – initial parameters [mu0, sig0, mu1, sig1, a]
- **bound** (*tuple of pairs*) – sequence of (min, max) values that constrain the parameters. If min or max are None, no boundary is set.
- **ptol** (*float*) – convergence condition. Relative max variation of any parameter.
- **max_iter** (*int*) – max number of iteration in case of non convergence.
- **weights** (*array*) – optional weigths, same size as `s` (for ex. 1/sigma^2 ~ nt).

**Returns**

Array of parameters for the 2-gaussians (5 elements)

`fretbursts.fit.gaussian_fitting.`**`two_gaussian_fit_EM_b`**(*s*, *p0=[0, 0.1, 0.6, 0.1, 0.5]*, *weights=None*, *bounds=[(None, None), (None, None), (None, None), (None, None), (None, None)]*, *max_iter=300*, *ptol=0.0001*, *debug=False*)

Fit the sample s with two gaussians using Expectation Maximization.

This version allows setting boundaries for each parameter.

**Parameters**

- **s** (*array*) – population of samples to be fitted
- **p0** (*sequence-like*) – initial parameters [mu0, sig0, mu1, sig1, a]
- **bound** (*tuple of pairs*) – sequence of (min, max) values that constrain the parameters. If min or max are None, no boundary is set.
- **ptol** (*float*) – convergence condition. Relative max variation of any parameter.
- **max_iter** (*int*) – max number of iteration in case of non convergence.
- **weights** (*array*) – optional weigths, same size as `s` (for ex. 1/sigma^2 ~ nt).

**Returns**

Array of parameters for the 2-gaussians (5 elements)

`fretbursts.fit.gaussian_fitting.`**`two_gaussian_fit_KDE_curve`**(*s*, *p0=[0, 0.1, 0.6, 0.1, 0.5]*, *weights=None*, *bandwidth=0.05*, *x_pdf=None*, *debug=False*, *method='SLSQP'*, *bounds=None*, *verbose=False*, ***kde_kwargs*)

Fit sample `s` with two gaussians using a KDE pdf approximation.

The 2-gaussian pdf is then curve-fitted to the KDE pdf.

**Parameters**

- **s** (*array*) – population of samples to be fitted
- **p0** (*sequence-like*) – initial parameters [mu0, sig0, mu1, sig1, a]
- **bandwidth** (*float*) – bandwidth for the KDE algorithm
- **method** (*string*) – fit method, can be 'leastsq' or one of the methods accepted by scipy `minimize()`
- **bounds** (*None or 5-element list*) – if not None, each element is a (min,max) pair of bounds for the corresponding parameter. This argument can be used only with L-BFGS-B, TNC or SLSQP methods. If bounds are used, parameters cannot be fixed
- **x_pdf** (*array*) – array on which the KDE PDF is evaluated and curve-fitted
- **weights** (*array*) – optional weigths, same size as `s` (for ex. 1/sigma^2 ~ nt).
- **debug** (*bool*) – if True performs more tests and print more info.

Additional kwargs are passed to scipy.stats.gaussian_kde().

**Returns**
Array of parameters for the 2-gaussians (5 elements)

`fretbursts.fit.gaussian_fitting.`**`two_gaussian_fit_cdf`**(*s*, *p0=[0.0, 0.05, 0.6, 0.1, 0.5]*, *fix_mu=[0, 0]*, *fix_sig=[0, 0]*)

Fit the sample s with two gaussians using a CDF fit.

Curve fit 2-gauss mixture Cumulative Distribution Function (CDF) to the empirical CDF for sample `s`.

Note that with a CDF fit no weighting is possible.

**Parameters**

- **s** (*array*) – population of samples to be fitted
- **p0** (*5-element list or array*) – initial guess or parameters
- **fix_mu** (*tuple of bools*) – Whether to fix the mean of the gaussians
- **fix_sig** (*tuple of bools*) – Whether to fix the sigma of the gaussians

**Returns**
Array of parameters for the 2-gaussians (5 elements)

`fretbursts.fit.gaussian_fitting.`**`two_gaussian_fit_curve`**(*x*, *y*, *p0*, *return_all=False*, *verbose=False*, ***kwargs*)

Fit a 2-gaussian mixture to the (x,y) curve. `kwargs` are passed to the leastsq() function.

If return_all=False then return only the fitted parameters If return_all=True then the full output of leastsq is returned.

`fretbursts.fit.gaussian_fitting.`**`two_gaussian_fit_hist`**(*s*, *bins=array([-0.5, -0.499, -0.498, ..., 1.497, 1.498, 1.499])*, *weights=None*, *p0=[0.2, 1, 0.8, 1, 0.3]*, *fix_mu=[0, 0]*, *fix_sig=[0, 0]*, *fix_a=False*)

Fit the sample s with 2-gaussian mixture (histogram fit).

Uses scipy.optimize.leastsq function. Parameters can be fixed but cannot be constrained in an interval.

**Parameters**

- **s** (*array*) – population of samples to be fitted
- **p0** (*5-element list or array*) – initial guess or parameters
- **bins** (*int or array*) – bins passed to `np.histogram()`
- **weights** (*array*) – optional weights passed to `np.histogram()`
- **fix_a** (*tuple of bools*) – Whether to fix the amplitude of the gaussians
- **fix_mu** (*tuple of bools*) – Whether to fix the mean of the gaussians
- **fix_sig** (*tuple of bools*) – Whether to fix the sigma of the gaussians

**Returns**
Array of parameters for the 2-gaussians (5 elements)

`fretbursts.fit.gaussian_fitting.`**`two_gaussian_fit_hist_min`**(*s*, *bounds=None*, *method='L-BFGS-B'*, *bins=array([-0.5, -0.499, -0.498, ..., 1.497, 1.498, 1.499])*, *weights=None*, *p0=[0.2, 1, 0.8, 1, 0.3]*, *fix_mu=[0, 0]*, *fix_sig=[0, 0]*, *fix_a=False*, *verbose=False*)

Fit the sample `s` with 2-gaussian mixture (histogram fit). [Bounded]

Uses scipy.optimize.minimize allowing constrained minimization.

**Parameters**

- **s** (*array*) – population of samples to be fitted
- **method** (*string*) – one of the methods accepted by scipy `minimize()`
- **bounds** (*None or 5-element list*) – if not None, each element is a (min,max) pair of bounds for the corresponding parameter. This argument can be used only with L-BFGS-B, TNC or SLSQP methods. If bounds are used, parameters cannot be fixed
- **p0** (*5-element list or array*) – initial guess or parameters
- **bins** (*int or array*) – bins passed to `np.histogram()`
- **weights** (*array*) – optional weights passed to `np.histogram()`
- **fix_a** (*tuple of bools*) – Whether to fix the amplitude of the gaussians
- **fix_mu** (*tuple of bools*) – Whether to fix the mean of the gaussians
- **fix_sig** (*tuple of bools*) – Whether to fix the sigma of the gaussians
- **verbose** (*boolean*) – allows printing fit information

**Returns**
Array of parameters for the 2-gaussians (5 elements)

`fretbursts.fit.gaussian_fitting.`**`two_gaussian_fit_hist_min_ab`**(*s*, *bounds=None*, *method='L-BFGS-B'*, *bins=array([-0.5, -0.499, -0.498, ..., 1.497, 1.498, 1.499])*, *weights=None*, *p0=[0.2, 1, 0.8, 1, 0.3]*, *fix_mu=[0, 0]*, *fix_sig=[0, 0]*, *fix_a=[0, 0]*, *verbose=False*)

Histogram fit of sample `s` with 2-gaussian functions.

Uses scipy.optimize.minimize allowing constrained minimization. Also each parameter can be fixed.

The order of the parameters is: mu1, sigma1, a1, mu2, sigma2, a2.

**Parameters**

- **s** (*array*) – population of samples to be fitted
- **method** (*string*) – one of the methods accepted by scipy `minimize()`
- **bounds** (*None or 6-element list*) – if not None, each element is a (min,max) pair of bounds for the corresponding parameter. This argument can be used only with L-BFGS-B, TNC or SLSQP methods. If bounds are used, parameters cannot be fixed
- **p0** (*6-element list or array*) – initial guess or parameters
- **bins** (*int or array*) – bins passed to `np.histogram()`
- **weights** (*array*) – optional weights passed to `np.histogram()`
- **fix_a** (*tuple of bools*) – Whether to fix the amplitude of the gaussians
- **fix_mu** (*tuple of bools*) – Whether to fix the mean of the gaussians
- **fix_sig** (*tuple of bools*) – Whether to fix the sigma of the gaussians
- **verbose** (*boolean*) – allows printing fit information

**Returns**

Array of parameters for the 2-gaussians (6 elements)

### Exponential fitting

Generic functions to fit exponential populations.

These functions can be used directly, or, in a typical FRETBursts workflow they are passed to higher level methods.

*See also:*

- *Background estimation*

`fretbursts.fit.exp_fitting.`**`expon_fit`**`(`*s*, *s_min=0*, *offset=0.5*, *calc_residuals=True*`)`

Fit sample `s` to an exponential distribution using the ML estimator.

This function computes the rate (Lambda) using the maximum likelihood (ML) estimator of the mean waiting-time (Tau), that for an exponentially distributed sample is the sample-mean.

**Parameters**

- **s** (*array*) – array of exponetially-distributed samples
- **s_min** (*float*) – all samples < `s_min` are discarded (`s_min` must be >= 0).
- **offset** (*float*) – offset for computing the CDF. See `get_ecdf()`.
- **calc_residuals** (*bool*) – if True compute the residuals of the fitted exponential versus the empirical CDF.

**Returns**

A 4-tuple of the fitted rate (1/life-time), residuals array, residuals x-axis array, sample size after threshold.

`fretbursts.fit.exp_fitting.`**`expon_fit_cdf`**(*s*, *s_min=0*, *offset=0.5*, *calc_residuals=True*)

Fit of an exponential model to the empirical CDF of `s`.

This function computes the rate (Lambda) fitting a line (linear regression) to the log of the empirical CDF.

**Parameters**

- **s** (*array*) – array of exponetially-distributed samples
- **s_min** (*float*) – all samples < `s_min` are discarded (`s_min` must be >= 0).
- **offset** (*float*) – offset for computing the CDF. See `get_ecdf()`.
- **calc_residuals** (*bool*) – if True compute the residuals of the fitted exponential versus the empirical CDF.

**Returns**

A 4-tuple of the fitted rate (1/life-time), residuals array, residuals x-axis array, sample size after threshold.

`fretbursts.fit.exp_fitting.`**`expon_fit_hist`**(*s*, *bins*, *s_min=0*, *weights=None*, *offset=0.5*, *calc_residuals=True*)

Fit of an exponential model to the histogram of `s` using least squares.

**Parameters**

- **s** (*array*) – array of exponetially-distributed samples
- **bins** (*float or array*) – if float is the bin width, otherwise is the array of bin edges (passed to `numpy.histogram`)
- **s_min** (*float*) – all samples < `s_min` are discarded (`s_min` must be >= 0).
- **weights** (*None or string*) – if None no weights is applied. if is 'hist_counts', each bin has a weight equal to its counts if is 'inv_hist_counts', the weight is the inverse of the counts.
- **offset** (*float*) – offset for computing the CDF. See `get_ecdf()`.
- **calc_residuals** (*bool*) – if True compute the residuals of the fitted exponential versus the empirical CDF.

**Returns**

A 4-tuple of the fitted rate (1/life-time), residuals array, residuals x-axis array, sample size after threshold.

`fretbursts.fit.exp_fitting.`**`get_ecdf`**(*s*, *offset=0.5*)

Return arrays (x, y) for the empirical CDF curve of sample `s`.

See the code for more info (is a one-liner!).

**Parameters**

- **s** (*array of floats*) – sample
- **offset** (*float, default 0.5*) – Offset to add to the y values of the CDF

**Returns**

*(x, y) (tuple of arrays)* – the x and y values of the empirical CDF

`fretbursts.fit.exp_fitting.`**`get_residuals`**(*s*, *tau_fit*, *offset=0.5*)

Returns residuals of sample `s` CDF vs an exponential CDF.

**Parameters**

- **s** (*array of floats*) – sample

- **tau_fit** (*float*) – mean waiting-time of the exponential distribution to use as reference
- **offset** (*float*) – Default 0.5. Offset to add to the empirical CDF. See `get_ecdf()` for details.

**Returns**
*residuals (array)* – residuals of empirical CDF compared with analytical CDF with time constant `tau_fit`.

## 3.7 Direct FRET fitting

*See also Fit framework*

This module contains functions for direct fitting of burst populations (FRET peaks) without passing through a FRET histogram.

This module provides a standard interface for different fitting algorithms.

`fretbursts.fret_fit.`**`fit_E_E_size`**(*nd*, *na*, *weights=None*, *gamma=1.0*, *gamma_correct=False*)

Fit the E with least-square minimization of errors on burst E values.

`fretbursts.fret_fit.`**`fit_E_binom`**(*nd*, *na*, *noprint=False*, *method='c'*, ***kwargs*)

Fit the E with MLE using binomial distribution. method ('a','b', or 'c') choose how to handle negative (nd,na) values.

`fretbursts.fret_fit.`**`fit_E_cdf`**(*nd*, *na*, *gamma=1.0*, ***kwargs*)

Fit E using the CDF curve-fit (see gaussian_fit_cdf). No weights are possible with this method.

`fretbursts.fret_fit.`**`fit_E_hist`**(*nd*, *na*, *gamma=1.0*, ***kwargs*)

Fit E using the histogram curve-fit (see gaussian_fit_hist). You can specify `weights` that will be passed to the `histogram` function.

`fretbursts.fret_fit.`**`fit_E_m`**(*nd*, *na*, *weights=None*, *gamma=1.0*, *gamma_correct=False*)

Fit the E with a weighted mean of burst E values.

`fretbursts.fret_fit.`**`fit_E_poisson_na`**(*nd*, *na*, *bg_a*, ***kwargs*)

Fit the E using MLE with na extracted from a Poisson.

`fretbursts.fret_fit.`**`fit_E_poisson_nd`**(*nd*, *na*, *bg_d*, ***kwargs*)

Fit the E using MLE with nd extracted from a Poisson.

`fretbursts.fret_fit.`**`fit_E_poisson_nt`**(*nd*, *na*, *bg_a*, ***kwargs*)

Fit the E using MLE with na extracted from a Poisson.

`fretbursts.fret_fit.`**`fit_E_slope`**(*nd*, *na*, *weights=None*, *gamma=1.0*)

Fit E with a least-squares fitting of slope on (nd,na) plane.

`fretbursts.fret_fit.`**`get_dist_euclid`**(*nd*, *na*, *E_fit=None*, *slope=None*)

Returns the euclidean distance of (nd,na) from a fit line. The fit line is specified by `slope` or by `E_fit`. Intercept is always 0.

`fretbursts.fret_fit.`**`get_weights`**(*nd*, *na*, *weights*, *naa=0*, *gamma=1.0*, *widths=None*)

Return burst weights computed according to different criteria.

The burst size is computed as `nd*gamma + na + naa`.

**Parameters**

- **nd, na, naa** (*1D arrays*) – photon counts in each burst.

- **gamma** (*float*) – gamma factor used for corrected burst size.
- **width** (*None array*) – array of burst durations used when weights='brightness'
- **weights** (*string or None*) – type of weights, possible weights are: 'size' burst size, 'size_min' burst size - min(burst size), 'size2' (burst size)^2, 'sqrt' sqrt(burst size), 'inv_size' 1/(burst size), 'inv_sqrt' 1/sqrt(burst size), 'cum_size' CDF_of_burst_sizes(burst size), 'cum_size2' CDF_of_burst_sizes(burst size)^2, 'brightness' the burst size divided by the burst width. If None returns uniform weights.
- **widths** (*1D array*) – bursts duration in seconds, needed only when weights = 'brightness'.

**Returns**
1D array of weights, one element per burst.

`fretbursts.fret_fit.`**`log_likelihood_binom`**`(`*E*, *nd*, *na*`)`

Likelihood function for (nd,na) to be from a binom with p=E (no BG).

`fretbursts.fret_fit.`**`log_likelihood_poisson_na`**`(`*E*, *nd*, *na*, *bg_a*`)`

Likelihood function for na extracted from Poisson. nd, na BG corrected.

`fretbursts.fret_fit.`**`log_likelihood_poisson_nd`**`(`*E*, *nd*, *na*, *bg_d*`)`

Likelihood function for nd extracted from Poisson. nd, na BG corrected.

`fretbursts.fret_fit.`**`log_likelihood_poisson_nt`**`(`*E*, *nd*, *na*, *bg_a*`)`

Likelihood function for na extracted from Poisson. nd, na BG corrected.

`fretbursts.fret_fit.`**`sim_nd_na`**`(`*E*, *N=1000*, *size_mean=100*`)`

Simulate an exponential-size burst distribution with binomial (nd,na)

## 3.8 Plotting Data

**Contents**



This module defines all the plotting functions for the `fretbursts.burstlib.Data` object.

The main plot function is `dplot()` that takes, as parameters, a `Data()` object and a 1-ch-plot-function and creates a subplot for each channel.

The 1-ch plot functions are usually called through `dplot` but can also be called directly to make a single channel plot.

The 1-ch plot functions names all start with the plot type (`timetrace`, `ratetrace`, `hist` or `scatter`).

**Example 1** - Plot the timetrace for all ch:

```
dplot(d, timetrace, scroll=True)
```

**Example 2** - Plot a FRET histogramm for each ch with a fit overlay:

```
dplot(d, hist_fret, show_model=True)
```

For more examples refer to FRETBurst notebooks.

### 3.8.1 Timetrace and ratetrace plots

`fretbursts.burst_plot.timetrace`(*d*, *i=0*, *binwidth=0.001*, *bins=None*, *tmin=0*, *tmax=200*, *bursts=False*, *burst_picker=True*, *scroll=False*, *show_rate_th=True*, *F=None*, *rate_th_style={'label': None}*, *show_aa=True*, *legend=False*, *set_ax_limits=True*, *burst_color='#BBBBBB'*, *plot_style=None*)

Plot the timetraces (histogram) of photon timestamps.

**Parameters**

- **d** (*Data object*) – the measurement's data to plot.
- **i** (*int*) – the channel to plot. Default 0.
- **binwidth** (*float*) – the bin width (seconds) of the timetrace histogram.
- **bins** (*array or None*) – If not None, defines the bin edges for the timetrace (overriding `binwidth`). If None, `binwidth` is use to generate uniform bins.
- **tmin, tmax** (*float*) – min and max time (seconds) to include in the timetrace. Note that a long time range and a small `binwidth` can require a significant amount of memory.
- **bursts** (*bool*) – if True, plot the burst start-stop times.
- **burst_picker** (*bool*) – if True, enable the ability to click on bursts to obtain burst info. This function requires the matplotlib's QT backend.
- **scroll** (*bool*) – if True, activate a scrolling bar to quickly scroll through the timetrace. This function requires the matplotlib's QT backend.
- **show_rate_th** (*bool*) – if True, plot the burst search threshold rate.
- **F** (*bool*) – if `show_rate` is True, show a rate `F` times larger than the background rate.
- **rate_th_style** (*dict*) – matplotlib style for the rate line.
- **show_aa** (*bool*) – if True, plot a timetrace for the AexAem photons. If False (default), plot timetraces only for DexDem and DexAem streams.
- **legend** (*bool*) – whether to show the legend or not.
- **set_ax_limits** (*bool*) – if True, set the xlim to zoom on a small portion of timetrace. If False, do not set the xlim, display the full timetrace.
- **burst_color** (*string*) – string containing the the HEX RGB color to use to highlight the burst regions.
- **plot_style** (*dict*) – matplotlib's style for the timetrace lines.

`fretbursts.burst_plot.timetrace_single`(*d*, *i=0*, *binwidth=0.001*, *bins=None*, *tmin=0*, *tmax=200*, *ph_sel=Ph_sel(Dex='DAem', Aex='DAem')*, *invert=False*, *bursts=False*, *burst_picker=True*, *scroll=False*, *cache_bins=True*, *plot_style=None*, *show_rate_th=True*, *F=None*, *rate_th_style={}*, *set_ax_limits=True*, *burst_color='#BBBBBB'*)

Plot the timetrace (histogram) of timestamps for a photon selection.

See `timetrace()` to plot multiple photon selections (i.e. Donor and Acceptor photons) in one step.

`fretbursts.burst_plot.`**`ratetrace`**(*d*, *i=0*, *m=None*, *max_num_ph=1000000.0*, *tmin=0*, *tmax=200*, *bursts=False*, *burst_picker=True*, *scroll=False*, *show_rate_th=True*, *F=None*, *rate_th_style={'label': None}*, *show_aa=True*, *legend=False*, *set_ax_limits=True*, *burst_color='#BBBBBB'*)

Plot the rate timetraces of photon timestamps.

**Parameters**

- **d** (*Data object*) – the measurement's data to plot.
- **i** (*int*) – the channel to plot. Default 0.
- **max_num_ph** (*int*) – Clip the rate timetrace after the max number of photons `max_num_ph` is reached.
- **tmin, tmax** (*float*) – min and max time (seconds) to include in the timetrace. Note that a long time range and a small `binwidth` can require a significant amount of memory.
- **bursts** (*bool*) – if True, plot the burst start-stop times.
- **burst_picker** (*bool*) – if True, enable the ability to click on bursts to obtain burst info. This function requires the matplotlib's QT backend.
- **scroll** (*bool*) – if True, activate a scrolling bar to quickly scroll through the timetrace. This function requires the matplotlib's QT backend.
- **show_rate_th** (*bool*) – if True, plot the burst search threshold rate.
- **F** (*bool*) – if `show_rate` is True, show a rate `F` times larger than the background rate.
- **rate_th_style** (*dict*) – matplotlib style for the rate line.
- **show_aa** (*bool*) – if True, plot a timetrace for the AexAem photons. If False (default), plot timetraces only for DexDem and DexAem streams.
- **legend** (*bool*) – whether to show the legend or not.
- **set_ax_limits** (*bool*) – if True, set the xlim to zoom on a small portion of timetrace. If False, do not set the xlim, display the full timetrace.
- **burst_color** (*string*) – string containing the the HEX RGB color to use to highlight the burst regions.

`fretbursts.burst_plot.`**`ratetrace_single`**(*d*, *i=0*, *m=None*, *max_num_ph=1000000.0*, *tmin=0*, *tmax=200*, *ph_sel=Ph_sel(Dex='DAem', Aex='DAem')*, *invert=False*, *bursts=False*, *burst_picker=True*, *scroll=False*, *plot_style={}*, *show_rate_th=True*, *F=None*, *rate_th_style={}*, *set_ax_limits=True*, *burst_color='#BBBBBB'*)

Plot the ratetrace of timestamps for a photon selection.

See `ratetrace()` to plot multiple photon selections (i.e. Donor and Acceptor photons) in one step.

`fretbursts.burst_plot.`**`timetrace_bg`**(*d*, *i=0*, *nolegend=False*, *ncol=2*, *plot_style={}*, *show_da=False*)

Timetrace of background rates.

`fretbursts.burst_plot.`**`timetrace_b_rate`**(*d*, *i=0*)

Timetrace of bursts-per-second in each period.

### 3.8.2 1D Histograms

#### Bursts: ratiometric quantities

`fretbursts.burst_plot.hist_fret`(*d*, *i=0*, *ax=None*, *binwidth=0.03*, *bins=None*, *pdf=True*, *hist_style='bar'*, *weights=None*, *gamma=1.0*, *add_naa=False*, *show_fit_stats=False*, *show_fit_value=False*, *fit_from='kde'*, *show_kde=False*, *bandwidth=0.03*, *show_kde_peak=False*, *show_model=False*, *show_model_peaks=True*, *hist_bar_style=None*, *hist_plot_style=None*, *model_plot_style=None*, *kde_plot_style=None*, *verbose=False*)

Plot FRET histogram and KDE.

The most used argument is `binwidth` that sets the histogram bin width.

For detailed documentation see `hist_burst_data()`.

`fretbursts.burst_plot.hist_S`(*d*, *i=0*, *ax=None*, *binwidth=0.03*, *bins=None*, *pdf=True*, *hist_style='bar'*, *weights=None*, *gamma=1.0*, *add_naa=False*, *show_fit_stats=False*, *show_fit_value=False*, *fit_from='kde'*, *show_kde=False*, *bandwidth=0.03*, *show_kde_peak=False*, *show_model=False*, *show_model_peaks=True*, *hist_bar_style=None*, *hist_plot_style=None*, *model_plot_style=None*, *kde_plot_style=None*, *verbose=False*)

Plot S histogram and KDE.

The most used argument is `binwidth` that sets the histogram bin width.

For detailed documentation see `hist_burst_data()`.

`fretbursts.burst_plot.hist_burst_data`(*d*, *i=0*, *data_name='E'*, *ax=None*, *binwidth=0.03*, *bins=None*, *vertical=False*, *pdf=False*, *hist_style='bar'*, *weights=None*, *gamma=1.0*, *add_naa=False*, *show_fit_stats=False*, *show_fit_value=False*, *fit_from='kde'*, *show_kde=False*, *bandwidth=0.03*, *show_kde_peak=False*, *show_model=False*, *show_model_peaks=True*, *hist_bar_style=None*, *hist_plot_style=None*, *model_plot_style=None*, *kde_plot_style=None*, *verbose=False*)

Plot burst_data (i.e. E, S, etc…) histogram and KDE.

This a generic function to plot histograms for any burst data. In particular this function is called by `hist_fret()` and `hist_S()` to make E and S histograms respectively.

Histograms and KDE can be plotted on any `Data` variable after burst search. To show a model, a model must be fitted first by calling `d.E_fitter.fit_histogram()`. To show the KDE peaks position, they must be computed first with `d.E_fitter.find_kde_max()`.

The arguments are shown below grouped in logical sections.

**Generic arguments**

**Parameters**

- **data_name** (*string*) – name of the burst data (i.e. 'E' or 'S')
- **ax** (*None or matplotlib axis*) – optional axis instance to plot in.
- **vertical** (*bool*) – if True the x axis is oriented vertically.
- **verbose** (*bool*) – if False, suppress any printed output.

**Histogram arguments**: control the histogram appearance

**Parameters**

- **hist_style** (*string*) – if 'bar' use a classical bar histogram, otherwise do a normal line plot of bin counts vs bin centers
- **bins** (*None or array*) – if None the bins are computed according to `binwidth`. If not None contains the arrays of bin edges and overrides `binwidth`.
- **binwidth** (*float*) – bin width for the histogram.
- **pdf** (*bool*) – if True, normalize the histogram to obtain a PDF.
- **hist_bar_style** (*dict*) – style dict for the histogram when `hist_style == 'bar'`.
- **hist_plot_style** (*dict*) – style dict for the histogram when `hist_style != 'bar'`.

**Model arguments**: control the model plot

**Parameters**

- **show_model** (*bool*) – if True shows the model fitted to the histogram
- **model** (*lmfit.Model object or None*) – lmfit Model used for histogram fitting. If None the histogram is not fitted.
- **show_model_peaks** (*bool*) – if True marks the position of model peaks
- **model_plot_style** (*dict*) – style dict for the model plot

**KDE arguments**: control the KDE plot

**Parameters**

- **show_kde** (*bool*) – if True shows the KDE curve
- **show_kde_peak** (*bool*) – if True marks the position of the KDE peak
- **bandwidth** (*float or None*) – bandwidth used to compute the KDE If None the KDE is not computed.
- **kde_plot_style** (*dict*) – style dict for the KDE curve

**Weights arguments** (weights are used to weight bursts according to their size, affecting histograms and KDEs).

**Parameters**

- **weights** (*string or None*) – kind of burst-size weights. See *`fretbursts.fret_fit.get_weights()`*.
- **gamma** (*float*) – gamma factor passed to `get_weights()`.
- **add_naa** (*bool*) – if True adds `naa` to the burst size.

**Fit text arguments**: control how to print annotation with fit information.

**Parameters**

- **fit_from** (*string*) – determines how to obtain the fit value. If 'kde' the fit value is the KDE peak. Otherwise it must be the name of a model parameter that will be used as fit value.
- **show_fit_value** (*bool*) – if True annotate the plot with fit value.
- **show_fit_stats** (*bool*) – if True annotate the figure with mean fit value and max deviation across the channels (for multi-spot).

### Bursts: tail distributions

`fretbursts.burst_plot.hist_size`(*d*, *i=0*, *which='all'*, *bins=(0, 600, 4)*, *pdf=False*, *weights=None*, *yscale='log'*, *gamma=1*, *beta=1*, *donor_ref=True*, *add_naa=False*, *ph_sel=None*, *naa_aexonly=False*, *naa_comp=False*, *na_comp=False*, *vline=None*, *label_prefix=None*, *legend=True*, *color=None*, *plot_style=None*)

Plot histogram of "burst sizes", according to different definitions.

**Parameters**

- **d** (*Data*) – Data object
- **i** (*int*) – channel index
- **bins** (*array or None*) – array of bin edges. If len(bins) == 3 then is interpreted as (start, stop, step) values.
- **which** (*string*) – what photons to include in "size". Valid values are 'all', 'nd', 'na', 'naa'. When 'all', sizes are computed with `d.burst_sizes()` (by default nd + na); 'nd', 'na', 'naa' get counts from `d.nd`, `d.na`, `d.naa` (respectively Dex-Dem, Dex-Aem, Aex-Aem).
- **gamma, beta** (*floats*) – factors used to compute the corrected burst size. Ignored when `which` != 'all'. See `fretbursts.burstlib.Data.burst_sizes_ich()`.
- **add_naa** (*bool*) – if True, include `naa` to the total burst size.
- **donor_ref** (*bool*) – convention used for corrected burst size computation. See `fretbursts.burstlib.Data.burst_sizes_ich()` for details.
- **na_comp** (*bool*) – **[PAX-only]** If True, multiply the `na` term by `(1 + Wa/Wd)`, where Wa and Wd are the D and A alternation durations (typically Wa/Wd = 1).
- **naa_aexonly** (*bool*) – **[PAX-only]** if True, the `naa` term is corrected to include only A emission due to A excitation. If False, the `naa` term includes all the counts in DAexAem. The `naa` term also depends on the `naa_comp` argument.
- **naa_comp** (*bool*) – **[PAX-only]** If True, multiply the `naa` term by `(1 + Wa/Wd)` where Wa and Wd are the D and A alternation durations (typically Wa/Wd = 1). The `naa` term also depends on the `naa_aexonly` argument.
- **label_prefix** (*string or None*) – a custom prefix for the legend label.
- **color** (*string or tuple or None*) – matplotlib color used for the plot.
- **pdf** (*bool*) – if True, normalize the histogram to obtain a PDF.
- **yscale** (*string*) – 'log' or 'linear', sets the plot y scale.
- **legend** (*bool*) – if True add legend to plot
- **plot_style** (*dict*) – dict of matplotlib line style passed to `plot`.
- **vline** (*float*) – If not None, plot vertical line at the specified x position.

**See also:**

- `fretbursts.burstlib.Data.burst_sizes_ich()`.
- `fretbursts.burstlib.Data.burst_sizes_pax_ich()`.

fretbursts.burst_plot.**hist_size_all**(*d*, *i=0*, ***kwargs*)

Plot burst sizes for all the combinations of photons.

Calls `hist_size()` multiple times with different `which` parameters.

fretbursts.burst_plot.**hist_width**(*d*, *i=0*, *bins=(0, 10, 0.025)*, *pdf=True*, *weights=None*, *yscale='log'*, *color=None*, *plot_style=None*, *vline=None*)

Plot histogram of burst durations.

**Parameters**

- **d** (*Data*) – Data object
- **i** (*int*) – channel index
- **bins** (*array or None*) – array of bin edges. If len(bins) == 3 then is interpreted as (start, stop, step) values.
- **pdf** (*bool*) – if True, normalize the histogram to obtain a PDF.
- **color** (*string or tuple or None*) – matplotlib color used for the plot.
- **yscale** (*string*) – 'log' or 'linear', sets the plot y scale.
- **plot_style** (*dict*) – dict of matplotlib line style passed to `plot`.
- **vline** (*float*) – If not None, plot vertical line at the specified x position.

fretbursts.burst_plot.**hist_brightness**(*d*, *i=0*, *bins=(0, 60, 1)*, *pdf=True*, *weights=None*, *yscale='log'*, *gamma=1*, *add_naa=False*, *ph_sel=Ph_sel(Dex='DAem', Aex='DAem')*, *beta=1.0*, *donor_ref=True*, *naa_aexonly=False*, *naa_comp=False*, *na_comp=False*, *label_prefix=None*, *color=None*, *plot_style=None*, *vline=None*)

Plot histogram of burst brightness, i.e. burst size / duration.

**Parameters**

- **d** (*Data*) – Data object
- **i** (*int*) – channel index
- **bins** (*array or None*) – array of bin edges. If len(bins) == 3 then is interpreted as (start, stop, step) values.
- **gamma, beta** (*floats*) – factors used to compute the corrected burst size. See `fretbursts.burstlib.Data.burst_sizes_ich()`.
- **add_naa** (*bool*) – if True, include `naa` to the total burst size.
- **donor_ref** (*bool*) – convention used for corrected burst size computation. See `fretbursts.burstlib.Data.burst_sizes_ich()` for details.
- **na_comp** (*bool*) – **[PAX-only]** If True, multiply the `na` term by `(1 + Wa/Wd)`, where Wa and Wd are the D and A alternation durations (typically Wa/Wd = 1).
- **naa_aexonly** (*bool*) – **[PAX-only]** if True, the `naa` term is corrected to include only A emission due to A excitation. If False, the `naa` term includes all the counts in DAexAem. The `naa` term also depends on the `naa_comp` argument.
- **naa_comp** (*bool*) – **[PAX-only]** If True, multiply the `naa` term by `(1 + Wa/Wd)` where Wa and Wd are the D and A alternation durations (typically Wa/Wd = 1). The `naa` term also depends on the `naa_aexonly` argument.
- **label_prefix** (*string or None*) – a custom prefix for the legend label.

- **color** (*string or tuple or None*) – matplotlib color used for the plot.
- **pdf** (*bool*) – if True, normalize the histogram to obtain a PDF.
- **yscale** (*string*) – 'log' or 'linear', sets the plot y scale.
- **plot_style** (*dict*) – dict of matplotlib line style passed to `plot`.
- **vline** (*float*) – If not None, plot vertical line at the specified x position.

`fretbursts.burst_plot.`**`hist_sbr`**(*d*, *i=0*, *bins=(0, 30, 1)*, *pdf=True*, *weights=None*, *color=None*, *plot_style=None*)

Histogram of per-burst Signal-to-Background Ratio (SBR).

`fretbursts.burst_plot.`**`hist_burst_phrate`**(*d*, *i=0*, *bins=(0, 1000, 20)*, *pdf=True*, *weights=None*, *color=None*, *plot_style=None*, *vline=None*)

Histogram of max photon rate in each burst.

### Others

`fretbursts.burst_plot.`**`hist_interphoton_single`**(*d*, *i=0*, *binwidth=0.0001*, *tmax=None*, *bins=None*, *ph_sel=Ph_sel(Dex='DAem', Aex='DAem')*, *period=None*, *yscale='log'*, *xscale='linear'*, *xunit='ms'*, *plot_style=None*)

Plot histogram of interphoton delays for a single photon streams.

**Parameters**

- **d** (*Data object*) – the input data.
- **i** (*int*) – the channel for which the plot must be done. Default is 0. For single-spot data the only valid value is 0.
- **binwidth** (*float*) – histogram bin width in seconds.
- **tmax** (*float or None*) – max timestamp delay in the histogram (seconds). If None (default), uses the the max timestamp delay in the stream. If not None, the plotted histogram may be further trimmed to the smallest delay with counts > 0 if this delay happens to be smaller than `tmax`.
- **bins** (*array or None*) – specifies the bin edged (in seconds). When `bins` is not None then the arguments `binwidth` and `tmax` are ignored. When `bins` is None, the bin edges are computed from the `binwidth` and `tmax` arguments.
- **ph_sel** (*Ph_sel object*) – photon stream for which plotting the histogram
- **period** (*int*) – the background period to use for plotting the histogram. The background period is a time-slice of the measurement from which timestamps are taken. If `period` is None (default) the time-windows is the full measurement.
- **yscale** (*string*) – scale for the y-axis. Valid values include 'log' and 'linear'. Default 'log'.
- **xscale** (*string*) – scale for the x-axis. Valid values include 'log' and 'linear'. Default 'linear'.
- **xunit** (*string*) – unit used for the x-axis. Valid values are 's', 'ms', 'us', 'ns'. Default 'ms'.
- **plot_style** (*dict*) – keyword arguments to be passed to matplotlib's `plot` function. Used to customize the plot style.

`fretbursts.burst_plot.hist_interphoton`(*d*, *i=0*, *binwidth=0.0001*, *tmax=None*, *bins=None*, *period=None*, *yscale='log'*, *xscale='linear'*, *xunit='ms'*, *plot_style=None*, *show_da=False*, *legend=True*)

Plot histogram of photon interval for different photon streams.

**Parameters**

- **d** (*Data object*) – the input data.
- **i** (*int*) – the channel for which the plot must be done. Default is 0. For single-spot data the only valid value is 0.
- **binwidth** (*float*) – histogram bin width in seconds.
- **tmax** (*float or None*) – max timestamp delay in the histogram (seconds). If None (default), uses the the max timestamp delay in the stream. If not None, the plotted histogram may be further trimmed to the smallest delay with counts > 0 if this delay happens to be smaller than `tmax`.
- **bins** (*array or None*) – specifies the bin edged (in seconds). When `bins` is not None then the arguments `binwidth` and `tmax` are ignored. When `bins` is None, the bin edges are computed from the `binwidth` and `tmax` arguments.
- **period** (*int*) – the background period to use for plotting the histogram. The background period is a time-slice of the measurement from which timestamps are taken. If `period` is None (default) the time-windows is the full measurement.
- **yscale** (*string*) – scale for the y-axis. Valid values include 'log' and 'linear'. Default 'log'.
- **xscale** (*string*) – scale for the x-axis. Valid values include 'log' and 'linear'. Default 'linear'.
- **xunit** (*string*) – unit used for the x-axis. Valid values are 's', 'ms', 'us', 'ns'. Default 'ms'.
- **plot_style** (*dict*) – keyword arguments to be passed to matplotlib's `plot` function. Used to customize the plot style.
- **show_da** (*bool*) – If False (default) do not plot the AexDem photon stream. Ignored when the measurement is not ALEX.
- **legend** (*bool*) – If True (default) plot a legend.

`fretbursts.burst_plot.hist_bg_single`(*d*, *i=0*, *binwidth=0.0001*, *tmax=0.01*, *bins=None*, *ph_sel=Ph_sel(Dex='DAem', Aex='DAem')*, *period=0*, *yscale='log'*, *xscale='linear'*, *xunit='ms'*, *plot_style=None*, *show_fit=True*, *fit_style=None*, *manual_rate=None*)

Plot histogram of photon interval for a single photon streams.

Optionally plots the fitted background as an exponential curve. Most arguments are described in `hist_interphoton_single()`. In the following we document only the additional arguments.

**Parameters**

- **show_fit** (*bool*) – If True shows the fitted background rate as an exponential distribution.
- **manual_rate** (*float or None*) – When not None use this value as background rate (ignoring the value saved in Data).
- **fit_style** (*dict*) – arguments passed to matplotlib's `plot` for for plotting the exponential curve.

For a description of all the other arguments see `hist_interphoton_single()`.

`fretbursts.burst_plot.hist_bg`(*d*, *i=0*, *binwidth=0.0001*, *tmax=0.01*, *bins=None*, *period=0*, *yscale='log'*, *xscale='linear'*, *xunit='ms'*, *plot_style=None*, *show_da=False*, *legend=True*, *show_fit=True*, *fit_style=None*)

Plot histogram of photon interval for different photon streams.

Optionally plots the fitted background. Most arguments are described in `hist_interphoton()`. In the following we document only the additional arguments.

**Parameters**

- **show_fit** (*bool*) – If True shows the fitted background rate as an exponential distribution.
- **fit_style** (*dict*) – arguments passed to matplotlib's `plot` for for plotting the exponential curve.

For a description of all the other arguments see `hist_interphoton()`.

`fretbursts.burst_plot.`**`hist_burst_delays`**(*d*, *i=0*, *bins=(0, 10, 0.2)*, *pdf=False*, *weights=None*, *color=None*, *plot_style=None*)

Histogram of waiting times between bursts.

`fretbursts.burst_plot.`**`hist_asymmetry`**(*d*, *i=0*, *bin_max=2*, *binwidth=0.1*, *stat_func=<function median>*)

### 3.8.3 ALEX plots

`fretbursts.burst_plot.`**`alex_jointplot`**(*d*, *i=0*, *gridsize=50*, *cmap='Spectral_r'*, *kind='hex'*, *vmax_fret=True*, *vmin=1*, *vmax=None*, *joint_kws=None*, *marginal_kws=None*, *marginal_color=10*, *rightside_text=False*, *E_name='E'*, *S_name='S'*)

Plot an ALEX join plot: an E-S 2D histograms with marginal E and S.

This function plots a jointplot: an inner 2D E-S distribution plot and the marginal distributions for E and S separately. By default, the inner plot is an hexbin plot, i.e. the bin shape is hexagonal. Hexagonal bins reduce artifacts due to discretization. The marginal plots are histograms with a KDE overlay.

**Parameters**

- **d** (*Data object*) – the variable containing the bursts to plot
- **i** (*int*) – the channel number. Default 0.
- **gridsize** (*int*) – the grid size for the 2D histogram (hexbin)
- **C** (*1D array or None*) – array of weights, it must have size equal to the number of bursts in channel `i` (d.num_bursts[i]). Passed to matplotlib hexbin().
- **cmap** (*string*) – name of the colormap for the 2D histogram. In addition to matplotlib colormaps, FRETburst defines these custom colormaps: 'alex_light', 'alex_dark' and 'alex_lv'. Default 'alex_light'.
- **kind** (*string*) – kind of plot for the 2-D distribution. Valid values: 'hex' for hexbin plots, 'kde' for kernel density estimation, 'scatter' for scatter plot.
- **vmax_fret** (*bool*) – if True, the colormap max value is equal to the max bin counts in the FRET region (S < 0.8). If False the colormap max is equal to the max bin counts.
- **vmin** (*int*) – min value in the histogram mapped by the colormap. Default 0, the colormap lowest color represents bins with 0 counts.
- **vmax** (*int or None*) – max value in the histogram mapped by the colormap. When None, vmax is computed automatically from the data and dependes on the argument `vmax_fret`. Default `None`.

- **joint_kws** (*dict*) – keyword arguments passed to the function which plots the inner 2-D distribution (i.e matplotlib scatter or hexbin or seaborn kdeplot). and hence to matplolib hexbin to customize the plot style.
- **marginal_kws** (*dict*) – keyword arguments passed to the function `hist_burst_data()` used to plot the maginal distributions.
- **marginal_color** (*int or color*) – color to be used for the marginal histograms. It can be an integer or any color accepted by matplotlib. If integer, it represents a color in the colormap `cmap` from 0 (lowest cmap color) to 99 (highest cmap color).
- **rightside_text** (*bool*) – when True, print the measurement name on the right side of the figure. When False (default) no additional text is printed.
- **E_name, S_name** (*string*) – name of the `Data` attribute to be used for E and S. The default is 'E' and 'S' respectively. These arguments are used when adding your own cutom E or S attributes to Data using `Data.add`. In this case, you can specify the name of these custom attributes so that they can be plotted as an E-S histogram.

**Returns**
A `matplotlib.figure.Figure` object that can be used for tweaking the plot.

`fretbursts.burst_plot.`**`hist2d_alex`**(*d*, *i=0*, *vmin=2*, *vmax=0*, *binwidth=0.05*, *S_max_norm=0.8*, *interp='bicubic'*, *cmap='hot'*, *under_color='white'*, *over_color='white'*, *scatter=True*, *scatter_ms=3*, *scatter_color='orange'*, *scatter_alpha=0.2*, *gui_sel=False*, *cbar_ax=None*, *grid_color='#D0D0D0'*)

Plot 2-D E-S ALEX histogram with a scatterplot overlay.

`fretbursts.burst_plot.`**`hexbin_alex`**(*d*, *i=0*, *vmin=1*, *vmax=None*, *gridsize=80*, *cmap='Spectral_r'*, *E_name='E'*, *S_name='S'*, ***hexbin_kwargs*)

Plot an hexbin 2D histogram for E-S.

`fretbursts.burst_plot.`**`plot_ES_selection`**(*ax*, *E1*, *E2*, *S1*, *S2*, *rect=True*, ***kwargs*)

Plot an overlay ROI on top of an E-S plot (i.e. ALEX histogram).

This function plots a rectangle and inscribed ellipsis with x-axis limits (E1, E2) and y-axis limits (S1, S2).

Note that, a dict with keys (E1, E2, S1, S2, rect) can be also passed to `fretbursts.select_bursts.ES()` to apply a selection.

**Parameters**

- **ax** (*matplotlib axis*) – the axis where the rectangle is plotted. Typically you pass the axis of a previous E-S scatter plot or histogram.
- **E1, E2, S1, S2** (*floats*) – limits for E and S (X and Y axis respectively) used to plot the rectangle.
- **rect** (*bool*) – if True, the rectangle is highlighted and the ellipsis is grey. The color are swapped otherwise.
- ****kwargs** – other keywords passed to both matplotlib's `Rectangle` and `Ellipse`.

**See also:**

For selecting bursts according to (`E1`, `E2`, `S1`, `S2`, `rect`) see:

- `fretbursts.select_bursts.ES()`

`fretbursts.burst_plot.`**`plot_alternation_hist`**(*d*, *bins=None*, *ax=None*, ***kwargs*)

Plot the ALEX alternation histogram for the variable `d`.

This function works both for us-ALEX and ns-ALEX data.

This function must be called on ALEX data **before** calling *fretbursts.loader.alex_apply_period()*.

`fretbursts.burst_plot.`**`plot_alternation_hist_nsalex`**(*d*, *bins=None*, *ax=None*, *ich=0*, *hist_style={}*, *span_style={}*)

Plot the ns-ALEX alternation histogram for the variable `d`.

This function must be called on ns-ALEX data **before** calling *fretbursts.loader.alex_apply_period()*.

### 3.8.4 Scatter plots

`fretbursts.burst_plot.`**`scatter_width_size`**(*d*, *i=0*)

Scatterplot of burst width versus size.

`fretbursts.burst_plot.`**`scatter_da`**(*d*, *i=0*, *alpha=0.3*)

Scatterplot of donor vs acceptor photons (nd, vs na) in each burst.

`fretbursts.burst_plot.`**`scatter_rate_da`**(*d*, *i=0*)

Scatter of nd rate vs na rate (rates for each burst).

`fretbursts.burst_plot.`**`scatter_fret_size`**(*d*, *i=0*, *which='all'*, *gamma=1*, *add_naa=False*, *plot_style=None*)

Scatterplot of FRET efficiency versus burst size.

`fretbursts.burst_plot.`**`scatter_fret_nd_na`**(*d*, *i=0*, *gamma=1.0*, ***kwargs*)

Scatterplot of FRET versus gamma-corrected burst size.

`fretbursts.burst_plot.`**`scatter_fret_width`**(*d*, *i=0*)

Scatterplot of FRET versus burst width.

`fretbursts.burst_plot.`**`scatter_naa_nt`**(*d*, *i=0*, *alpha=0.5*)

Scatterplot of nt versus naa.

`fretbursts.burst_plot.`**`scatter_alex`**(*d*, *i=0*, ***kwargs*)

Scatterplot of E vs S. Keyword arguments passed to `plot`.

## 3.9 Burst Search in FRETBursts

This section describes details and conventions used to implement burst search in FRETBursts. For a more general explanation of burst search concepts see (Ingargiola PLOS ONE 2016). For usage examples see the s-ALEX notebook. An analysis of implementation performances of the *low-level burst search* can be found in this blog post: Optimizing burst search in python.

### 3.9.1 Defining the rate estimator

Before describing FRETBursts implementation let me introduce an expression for computing rates of random events that will be used later on. A general expression, used by FRETBursts (since version 0.5.6), for estimating the rate using $m$ consecutive timestamps is:

$$\hat{\lambda} = \frac{m - 1 - c}{t_{i+m-1} - t_i} \tag{3.1}$$

where $c$ is a parameter that can be passed to all FRETBursts functions that deal with photon rates. Note that $m$ is the number of photons and $m-1$ is the number of inter-photon delays. For example, using $c = 1$, yields an unbiased estimator of the rate for events generated by a stationary Poisson process. See this notebook for a discussion of the different estimator properties as a function of $c$. In practice, the choice of $c$ is just a convention and it is provided for flexibility and to match results of other software that may use a different definition.

In FRETBursts version 0.5.5 or earlier, there is no `c` parameter and the rate is always computed as $\hat{\lambda} = m/(t_{i+m-1} - t_i)$ (equivalent to $c = -1$).

### 3.9.2 Conventions in burst search

Burst search is mainly performed calling the method `Data.burst_search()`. The AND-gate burst search function (`fretbursts.burstlib_ext.burst_search_and()`) calls `Data.burst_search()` under the hood, so all the considerations below are also valid for the AND-gate version.

With `Data.burst_search()`, you can perform burst search by setting a "rate threshold" *F* times larger than the background rate (argument `F`), or you can just set a single fixed rate for the full measurement (argument `min_rate_cps`). In both cases the real burst search is performed by the low-level function `phtools.burstsearch.bsearch_py()`, which takes as input parameters *m* and *T*. This function finds bursts when a group of *m* consecutive photons lies within a time window *T*. You can find an analysis of the algorithm implementation and performance considerations in this blog post.

When using the `F` argument, FRETBursts will choose the appropriate `T` for each background period in order to obtain a "rate threshold" *F times larger than background rate*. In this case, FRETBursts uses the following expression to compute *T* (derived from (3.1)):

$$T(t) = \frac{m - 1 - c}{F \cdot \hat{\lambda}_{bg}(t)}$$

where $\hat{\lambda}_{bg}(t)$ is the estimated background rate as a function of time ($t$).

Conversely, when directly fixing a rate with the argument `min_rate_cps` ($\lambda_{th}$), FRETBursts computes $T$ using the expression:

$$T = \frac{m - 1 - c}{\lambda_{th}}$$

The parameter $c$ can be specified when performing burst search. When not specified, the default value of $c = -1$ is used. This choice preserves backward compatibility with results obtained with FRETBursts 0.5.5 or earlier.

### 3.9.3 The Core Algorithm

The different types of burst search described in the previous sections are implemented calling the same low-level burst search function which implements the core "sliding window" algorithm. Here we explain in details this core algorithm.

The low-level burst search takes as an input the array of (monotonically increasing) photon timestamps, as well as two other arguments *m* (the number of timestamps) and *T* (the time window duration). Starting from the the first element of the array, we consider all the m-tuple of timestamps [0..m-1], [1..m], etc.

**Point 1.** For each m-tuple if the timestamps are contained in a time window smaller or equal to $T$ we mark a burst start at the position of the first timestamp in the current m-tuple. Otherwise we take the next m-tuple and repeat the check.

Once a burst starts, we keep "sliding" the m-tuple one timestamp a time. If the current m-tuple is still contained in a window of duration $T$ the burst continues. When the current m-tuple is contained in a window larger than $T$ the burst ends. When this happens, the last timestamp in a burst is the (m-1)-th timestamp of current m-tuple (i.e. the last timestamps of the **previous** m-tuple which was still contained in a window $T$). After the burst ends, we continue as in point 1 checking the next m-tuple. This is shifted by only one timestamp (i.e. there is no jump when the burst ends).

At this point, it can happen that the current m-tuple is contained in $T$ and a new burst starts right away. In this situation the new bursts will have m-2 timestamps overlapping with the previous one.

At the end of the timestamp array, if a burst is currently started we end it by marking the last timestamp as burst stop. The set of bursts obtained in this way has the minimum-rate property, i.e. all the m-tuple of consecutive timestamps in any burst are guaranteed to be contained in a windows $T$ or smaller. Conversely, a few bursts will overlap and thus share some timestamps. If the user wants to avoid overlapping bursts a burst fusion steps must be applied as described in next section. Note, however, that after fusing overlapping bursts at least one m-tuple inside each fused burst will not have the minimum-rate property, i.e. the m-tuple is contained in a window larger than $T$.

The previous function is implemented in `phtools.burstsearch.bsearch_py()` (pure python version) and in `phtools.burstsearch_c.bsearch_c()` (optimized cython version). Several tests make sure that the two functions return numerically identical results. An analysis of performance of of different implementations can be found in this blog post: Optimizing burst search in python.

### 3.9.4 Burst Fusion

Burst fusion is an operation which fuses consecutive bursts if the start of the second bursts minus the end of the first burst (called burst separation) is <= of a fusion time $t_f$. When bursts are overlapping (see previous section) the burst separation is negative. Therefore, to avoid overlapping bursts, we need to apply fusion with separation of 0. Note that with this condition, if a bursts ends on a timestamp which is the start of the next burst (i.e. 1 overlapping photon) the two bursts will be fused. Conversely if one burst ends and the next burst starts one photon later (0 overlapping photons) the two bursts will be kept separated. In the latter case, there will be no timestamp between the end of the previous burst and the start of the next one.

To perform burst fusion use the method `Data.fuse_bursts()`.

### 3.9.5 Low-level burst search functions

The module `phtools.burstsearch` provides the low-level (or core) burst search and photon counting functions. This module also provides `Bursts`, a container for a set of bursts. `Bursts` provides attributes for the main burst quatitites (`istart`, `istop`, `start`, `stop`, `counts`, `width`, etc…). It implements the iterator interface (iterate burst by burst). Moreover `Bursts` can be indexed (`[]`, i.e. `getitem` interface) supporting the same indexing as a numpy 1-D array.

The burst search functions return a 2-D array (burst array) of shape Nx4, where N is the number of bursts. This array can used to build a `Bursts` object using:

```
Bursts(bursts_array)
```

As an example, let assume having a burst array `bursts`. To take a slice of only the first 10 bursts you can do:

```
bursts10 = bursts[:10]   # new Bursts object with the first 10 bursts
```

To obtain the burst start of all the bursts:

```
bursts.start
```

To obtain the burst counts (number of photons) for the 10-th to 20-th burst:

```
bursts[10:20].counts
```

For efficiency, when iterating over `Bursts` the returned burst is a named tuple `Burst`, which implements the same attributes as `Bursts` (istart, istop, start, stop, counts and width). This results in faster iteration and attribute access than using `Bursts` objects with only one burst.

Three methods allow to transform Bursts to refer to a new timestamps array:

- `Bursts.recompute_times()`

- *Bursts.recompute_index_expand()*
- *Bursts.recompute_index_reduce()*

Finally, in order to support fusion of consecutive bursts, we provide the class *BurstsGap* (and single-burst version *BurstGap*) which add the attributes `gap` and `gap_counts` that contains the duration and the number of photons in gaps inside a burst. The attribute `width` is the total burst duration minus `gap`, while `counts` is the total number of photons minus photons falling inside gaps (gaps are open intervals, do not include edges).

`class fretbursts.phtools.burstsearch.`**`Burst`**`(`*istart*, *istop*, *start*, *stop*`)`

Container for a single burst.

**`property counts`**

Number of photons in the burst.

**`property ph_rate`**

Photon rate in the burst (total photon counts/duration).

**`property width`**

Burst duration in timestamps unit.

`class fretbursts.phtools.burstsearch.`**`BurstGap`**`(`*istart*, *istop*, *start*, *stop*, *gap*, *gap_counts*`)`

**`property counts`**

Number of photons in the burst, minus `gap_counts`.

**`static from_burst`**`(`*burst*`)`

Build a `BurstGap` from a *Burst* object.

**`property width`**

Burst duration in timestamps unit, minus `gap` time.

`class fretbursts.phtools.burstsearch.`**`Bursts`**`(`*burstarray*`)`

A container for burst data.

This class provides a container for burst data. It has a set of attributes (`start`, `stop`, `istart`, `istop`, `counts`, `width`, `ph_rate`, `separation`) that can be accessed to obtain burst data. Only a few fundamental attributes are stored, the others are comuputed on-fly using python properties.

Other attributes are `dataframe` (a `pandas.DataFrame` with the complete burst data), `num_bursts` (the number of bursts).

`Bursts` objects can be built from a list of single *Burst* objects by using the method *Bursts.from_list()*, or from 2D arrays containing bursts data (one row per burst; columns: istart, istop, start, stop) such as the ones returned by burst search functions (e.g. *bsearch_py()*).

`Bursts` objects are iterable, yielding one burst a time (*Burst* objects). `Bursts` can be compared for equality (with ==) and copied (*Bursts.copy()*).

Additionally basic methods for burst manipulation are provided:

- `recompute_times` recompute start and stop times using the current start and stop index and a new timestamps array passed as argument.
- `recompute_index_*` recompute start and stop indexes to refer to an expanded or reduced timestamp selection.

Other methods are:

- `and_gate` computing burst intersection with a second set of bursts. Used to implement the dual-channel burst search (DCBS).

Methods that may be implemented in the future:

- `or_gate`: computing union with a second set of bursts.
- `fuse_bursts`: fuse nearby bursts.

**and_gate**(*bursts2*)

From 2 burst arrays return bursts defined as intersection (AND rule).

The two input burst-arrays come from 2 different burst searches. Returns new bursts representing the overlapping bursts in the 2 inputs with start and stop defined as intersection (or AND) operator.

Both input and output are Bursts objects.

**Parameters**
**bursts_a** (*Bursts object*) – second set of bursts to be intersected with bursts in self. The number of bursts in self and `bursts_a` can be different.

**Returns**
Bursts object containing intersections (AND) of overlapping bursts.

**copy**()

Return a new copy of current `Bursts` object.

**property counts**

Number of photons in each burst.

**property dataframe**

A `pandas.DataFrame` containing burst data, one row per burst.

**classmethod empty**(*num_bursts=0*)

Return an empty `Bursts()` object.

**classmethod from_list**(*bursts_list*)

Build a new `Bursts()` object from a list of *`Burst`*.

**property istart**

Index of 1st ph in each burst

**property istop**

Index of last ph in each burst

**join**(*bursts*, *sort=False*)

Join the current `Bursts` object with another one. Returns a copy.

**classmethod merge**(*list_of_bursts*, *sort=False*)

Merge `Bursts` in `list_of_bursts`, returning a new `Bursts` object.

**property num_bursts**

Number of bursts.

**property ph_rate**

Photon rate in burst (tot size/duration)

**recompute_index_expand**(*mask*, *out=None*)

Recompute istart and istop from selection `mask` to full timestamps.

This method returns a new Bursts object with recomputed istart and istop. Old istart, istop are assumed to be index of a reduced array `timestamps[mask]`. New istart, istop are computed to be index of a "full" timestamps array of size `mask.size`.

This is useful when performing burst search on a timestamps selection and we want to transform the burst data to use the index of the "full" timestamps array.

**Parameters**

- **mask** (*bool array*) – boolean mask defining the timestamps selection on which the old istart and istop were computed.
- **out** (*None or Bursts*) – if None (default), do computations on a copy of the current object. Otherwise, modify the `Bursts` object passed (can be used for in-place operations).

**Returns**
`Bursts` object with recomputed istart/istop.

`recompute_index_reduce`(*times_reduced*, *out=None*)

Recompute istart and istop on reduced timestamps `times_reduced`.

This method returns a new Bursts object with same start and stop times and recomputed istart and istop. Old istart, istop are assumed to be index of a "full" timestamps array of size `mask.size`. New istart, istop are computed to be index of the reduced timestamps array `timestamps_reduced`.

Note: it is required that all the start and stop times are also contained in the reduced timestamps selection.

This method is the inverse of *recompute_index_expand()*.

**Parameters**

- **times_reduced** (*array*) – array of selected timestamps used to compute the new istart and istop. This array needs to be a sub-set of the original timestamps array.
- **out** (*None or Bursts*) – if None (default), do computations on a copy of the current object. Otherwise, modify the `Bursts` object passed (can be used for in-place operations).

**Returns**
`Bursts` object with recomputed istart/istop times.

`recompute_times`(*times*, *out=None*)

Recomputes start, stop times using timestamps from a new array.

This method computes burst start, stop using the index of timestamps from the current object and timestamps from the passed array `times`.

This is useful, for example, when burst search is computed on a "compacted" timestamps array (i.e. removing the gaps outside the alternation period in usALEX experiments), and afterwards the "real" start and stop times needs to be recomputed.

**Parameters**

- **times** (*array*) – array of photon timestamps
- **out** (*None or Bursts*) – if None (default), do computations on a copy of the current object. Otherwise, modify the `Bursts` object passed (can be used for in-place operations).

**Returns**
`Bursts` object with recomputed start/stop times.

`property separation`

Separation between nearby bursts

`property size`

Number of bursts. Used for compatibility with ndarray.size. Use `Bursts.num_bursts` preferentially.

**property start**

Time of 1st ph in each burst

**property stop**

Time of last ph in each burst

**property width**

Burst duration in timestamps units.

**class** `fretbursts.phtools.burstsearch.`**`BurstsGap`**(*burstarray*)

A container for bursts with optional gaps.

This class extend Bursts adding the attributes/properties `gap` (a duration) and `gap_counts` (counts in gap) that allow accounting for gaps inside bursts.

**property counts**

Number of photons in each burst, minus the `gap_counts`.

**property dataframe**

A `pandas.DataFrame` containing burst data, one row per burst.

**classmethod from_list**(*bursts_list*)

Build a new `BurstsGap()` from a list of *BurstGap*.

**property gap**

Time gap inside a burst

**property gap_counts**

Number of photons falling inside gaps of each burst.

**property width**

Burst duration in timestamps units, minus the `gap` time.

`fretbursts.phtools.burstsearch.`**`bsearch_py`**(*times*, *L*, *m*, *T*, *slice_=None*, *label='Burst search'*, *verbose=True*)

Sliding window burst search. Pure python implementation.

Finds bursts in the array `time` (int64). A burst starts when the photon rate is above a minimum threshold, and ends when the rate falls below the same threshold. The rate-threshold is defined by the ratio `m/T` (`m` photons in a time interval `T`). A burst is discarded if it has less than `L` photons.

**Parameters**

- **times** (*array, int64*) – array of timestamps on which to perform the search
- **L** (*int*) – minimum number of photons in a bursts. Bursts with size (or counts) < L are discarded.
- **m** (*int*) – number of consecutive photons used to compute the rate.
- **T** (*float*) – max time separation of `m` photons to be inside a burst
- **slice_** (*tuple*) – 2-element tuple used to slice times
- **label** (*string*) – a label printed when the function is called
- **verbose** (*bool*) – if False, the function does not print anything.

**Returns**

Array of burst data Nx4, type int64. Column order is: istart, istop, start, stop.

`fretbursts.phtools.burstsearch.`**`count_ph_in_bursts`**`(`*bursts*, *mask*`)`

Counts number of photons in each burst counting only photons in `mask`.

This function takes a *`Bursts`* object and a boolean mask (photon selection) and computes the number of photons selected by the mask. It is used, for example, to count donor and acceptor photons in each burst.

For a multi-channel version see *`mch_count_ph_in_bursts_py()`*.

**Parameters**

- **bursts** (*Bursts object*) – the bursts used as input
- **mask** (*1D boolean array*) – the photon mask. The boolean mask must be of the same size of the timestamp array used for burst search.

**Returns**

A 1D array containing the number of photons in each burst counting only photons in the selection mask.

`fretbursts.phtools.burstsearch.`**`mch_count_ph_in_bursts_py`**`(`*Mburst*, *Mask*`)`

Counts number of photons in each burst counting only photons in `Mask`.

This multi-channel function takes a list of a *`Bursts`* objects and photon masks and computes the number of photons selected by the mask in each channel.

It is used, for example, to count donor and acceptor photons in each burst.

For a single-channel version see `count_ph_in_bursts_py()`.

**Parameters**

- **Mburst** (*list Bursts objects*) – a list of bursts collections, one per ch.
- **Mask** (*list of 1D boolean arrays*) – a list of photon masks (one per ch), For each channel, the boolean mask must be of the same size of the timestamp array used for burst search.

**Returns**

A list of 1D array, each containing the number of photons in each burst counting only photons in the selection mask.

## 3.10 Photon rates functions

This module provides functions to compute photon rates from timestamps arrays. Different methods to compute rates are implemented:

1. Consecutive set of *m* timestamps ("sliding m-tuple")
2. KDE-based methods with Gaussian or Laplace distribution or rectangular kernels.

**Note:** When using of "sliding m-tuple" method (1), rates can be only computed for each consecutive set of *m* timestamps. The time-axis can be computed from the mean timestamp in each m-tuple.

When using the KDE method, rates can be computed at any time point. Practically, the time points at which rates are computed are timestamps (in a photon stream). In other words, we don't normally use a uniformly sampled time axis but we use a timestamps array as time axis for the rate.

Note that computing rates with a fixed sliding time window and sampling the function by centering the window on each timestamp is equivalent to a KDE-based rate computation using a rectangular kernel.

fretbursts.phtools.phrates.**kde_gaussian**(*timestamps*, *tau*, *time_axis=None*)

Computes Gaussian KDE for `timestamps` evaluated at `time_axis`.

Computes KDE rates of `timestamps` using a Gaussian kernel:

```
kernel = exp( -(t - t0)^2 / (2 * tau^2)) )
```

The rate is computed for each time point in `time_axis`. When `time_axis` is None, then `timestamps` is used as time axis.

**Parameters**

- **timestamps** (*array*) – arrays of photon timestamps
- **tau** (*float*) – sigma of the Gaussian kernel
- **time_axis** (*array or None*) – array of time points where the rate is computed. If None, uses `timestamps` as time axis.

**Returns**

*rates (array)* – non-normalized rates (just the sum of the Gaussian kernels). To obtain rates in Hz divide the array by `2.5*tau`.

fretbursts.phtools.phrates.**kde_laplace**(*timestamps*, *tau*, *time_axis=None*)

Computes exponential KDE for `timestamps` evaluated at `time_axis`.

Computes KDE rates of `timestamps` using a laplace distribution kernel (i.e. symmetric-exponential):

```
kernel = exp( -|t - t0| / tau)
```

The rate is computed for each time point in `time_axis`. When `time_axis` is None, then `timestamps` is used as time axis.

**Parameters**

- **timestamps** (*array*) – arrays of photon timestamps
- **tau** (*float*) – time constant of the exponential kernel
- **time_axis** (*array or None*) – array of time points where the rate is computed. If None, uses `timestamps` as time axis.

**Returns**

*rates (array)* – non-normalized rates (just the sum of the exponential kernels). To obtain rates in Hz divide the array by `2*tau` (or other conventional x*tau duration).

fretbursts.phtools.phrates.**kde_rect**(*timestamps*, *tau*, *time_axis=None*)

Computes KDE with rect kernel for `timestamps` evaluated at `time_axis`.

Computes KDE rates of `timestamps` using a rectangular kernel which is 1 in the range `[-tau/2, tau/2]` and 0 otherwise.

The rate is computed for each time point in `time_axis`. When `time_axis` is None, then `timestamps` is used as time axis.

**Parameters**

- **timestamps** (*array*) – arrays of photon timestamps
- **tau** (*float*) – duration of the rectangular kernel
- **time_axis** (*array or None*) – array of time points where the rate is computed. If None, uses `timestamps` as time axis.

**Returns**
*rates (array)* – non-normalized rates (just the sum of the rectangular kernels). To obtain rates in Hz divide the array by `tau`.

`fretbursts.phtools.phrates.`**`mtuple_delays`**(*ph*, *m*)

Compute array of m-photons delays of size ph.size - m + 1.

The m-photons delay is defined as the difference between the last and first timestamp in each set of *m* consecutive timestamps. The m-photons delay expression is:

```
t[i + m - 1] - t[i]
```

for each *i* in `[0 .. ph.size - m]`.

**Parameters**

- **ph** (*array*) – photon timestamps array
- **m** (*int*) – number of timestamps to use

**Returns**
Array of m-photons delays, with size equal to `ph.size - m + 1`.

`fretbursts.phtools.phrates.`**`mtuple_delays_min`**(*ph*, *m*)

Compute the min m-photons delay in `ph`.

`fretbursts.phtools.phrates.`**`mtuple_rates`**(*ph*, *m*, *c=1*)

Compute the instantaneous rates for timestamps in `ph` using `m` photons.

Compute the rates for all the consecutive sets of *m* photons. Noting that the number of inter-photon delays is *n = m - 1*, the rate is computed with the expression:

```
(n - c) / (t[last] - t[first])
```

where "last" and "first" refer to the last and first timestamp in each group of *m* consecutive timestamps.

By changing `c` we obtain estimators with different properties. When c=1 (default), the result is the unbiased estimator of the rate. When c=1/3 we obtain the estimator whose median is equal to the the rate. Empirically, the minimal RMS error is committed with c=2. All the previous considerations are valid under the assumption that we are estimating the rate of events generated by a stationary Poisson process.

**Parameters**

- **ph** (*array*) – photon timestamps array
- **m** (*int*) – number of timestamps to use for computing the rate
- **c** (*float*) – correction factor for the rate estimation.

**Returns**
Array of rates, with size equal to `ph.size - m + 1`.

`fretbursts.phtools.phrates.`**`mtuple_rates_max`**(*ph*, *m*, *c=1*)

Compute max m-photon rate in `ph`.

`fretbursts.phtools.phrates.`**`mtuple_rates_t`**(*ph*, *m*)

Compute mean time for each rate computed by `mtuple_rates`.

## 3.11 FRETBursts plugins

The module `burtlib_ext.py` (by default imported as `bext`) contains extensions to `burstslib.py`. It can be though as a simple plugin system for FRETBursts.

Functions here defined operate on *fretbursts.burstlib.Data()* objects extending the functionality beyond the core functions and methods defined in `burstlib`. This modularization allows to implement new functions without overloading the *fretbursts.burstlib.Data* with an high number of non-core methods.

The type of functions here implemented are quite diverse. A short summary follows.

- *burst_search_and_gate()* performs the AND-gate burst search taking intersection of the bursts detected in two photons streams.
- *burst_data()* returns a pandas DataFrame with burst data (one burst per row). Burst data includes sizes, duration, E, S, etc. . . .
- *burst_photons()* returns a pandas DataFrame with photon data such as timestamps or nanotimes inside bursts (one photon per row).
- *bursts_fitter()* and *fit_bursts_kde_peak()* help to build and fit histograms and KDEs for E or S.
- *calc_mdelays_hist()* computes the histogram of the m-delays distribution of photon intervals.
- *moving_window_chunks()*: slices the measurement using a moving-window (along the time axis). Used to follow or detect kinetics.
- *join_data()* joins different measurements to create a single "virtual" measurement from a series of measurements.

Finally a few functions deal with burst timestamps:

- *ph_burst_stats()* compute any statistics (for example mean or median) on the timestamps of each burst.
- *asymmetry()* returns a burst "asymmetry index" based on the difference between Donor and Acceptor timestamps.

`fretbursts.burstlib_ext.`**`asymmetry`**(*dx*, *ich=0*, *func=<function mean>*, *dropnan=True*)

Compute an asymmetry index for each burst in channel `ich`.

It computes each burst the difference func({t_D}) - func({t_A}) where `func` is a function (default `mean`) that computes some statistics on the timestamp and {t_D} and {t_A} are the sets of D or A timestamps in a bursts (during D excitation).

**Parameters**

- **d** (*Data*) – Data() object
- **ich** (*int*) – channel index
- **func** (*function*) – the function to be used to extract D and A photon statistics in each bursts.

**Returns**

An arrays of photon timestamps (one array per burst).

`fretbursts.burstlib_ext.`**`burst_data`**(*dx*, *include_bg=False*, *include_ph_index=False*, *skip_ch=None*)

Return a table (`pd.DataFrame`) of burst data (one row per burst).

Columns include:

- *E* and *S*: FRET and stoichiometry for each burst.
- *nd*, *na*, *naa*, *nda*: burst counts in DexDem, DexAem, AexAem and AexDem photon streams.
- *t_start*, *t_stop*: time (in seconds) of first and last photon inside the burst

- *width_ms*: burst duration in milliseconds
- *size_raw*: the total uncorrected burst counts in the photon stream used for burst search

Optional columns include:

- *i_start*, *i_stop*: index of burst start and stop relative to the original timestamps array (requires `include_ph_index=True`)
- *bg_dd*, *bg_ad*, *bg_aa*, *bg_da*: background contribution in the DexDem, DexAem, AexAem, AexDem photon streams (requires `include_bg=True`).

If the peak photon-counts in each bursts has been computed (see `fretbursts.burstlib.Data.calc_max_rate()`), it will be included as a column called *max_rate*.

**Parameters**

- **include_bg** (*bool*) – if True includes additional columns for burst background (see above). Default False.
- **include_ph_index** (*bool*) – if True includes additional two columns for index of first and last timestamp in each burst. Default False.
- **skip_ch** (*list or None*) – List of channels to skip if measurement is multispot.

**Returns**

A pandas's DataFrame containing burst data (one row per burst).

`fretbursts.burstlib_ext.`**`burst_data_period_mean`**(*dx*, *burst_data*)

Compute mean `burst_data` in each period.

**Parameters**

- **dx** (*Data object*) – contains the burst data to process
- **burst_data** (*list of arrays*) – one array per channel, each array has one element of "burst data" per burst.

**Returns**

2D of arrays with shape (nch, nperiods).

---

**Example**

burst_period_mean(dx, dx.nt)

---

`fretbursts.burstlib_ext.`**`burst_photons`**(*dx*, *skip_ch=None*)

Return a `pandas.DataFrame` of photon-data for bursts in `dx`.

The returned DataFrame has one row per "photon". Columns include:

- *timestamp*: the timestamp of each photon
- *nantotime*: the TCSPC nanotime of each photon (if available)
- *stream*: a categorical column indicating the stream of each photon.
- *spot*: (multispot only) the spot number for each photon

The returned DataFrame has a hierarchical index made of two integers: (burst_id, photon_id). `burst_id` identifies the burst while `photon_id` identifies each photon in a burst. `burst_id` is the same number used as index in the `DataFrame` returned by `burst_data()`. `photon_id` always starts at 0 for the first photon in each burst.

**Parameters**

- **dx** (*Data*) – the Data object containing the measurement
- **skip_ch** (*list or None*) – List of channels to skip if measurement is multispot. Default None

**Returns**
A pandas's DataFrame containing the photon data for the bursts in `dx`. The DataFrame has one row per photon.

`fretbursts.burstlib_ext.`**`burst_search_and_gate`**(*dx*, *F=6*, *m=10*, *min_rate_cps=None*, *c=-1*, *ph_sel1=Ph_sel(Dex='DAem', Aex=None)*, *ph_sel2=Ph_sel(Dex=None, Aex='Aem')*, *compact=False*, *mute=False*)

Return a Data object containing bursts obtained by and-gate burst-search.

The and-gate burst search is a composition of 2 burst searches performed on different photon selections. The bursts in the and-gate burst search are the overlapping bursts in the 2 initial burst searches, and their duration is the intersection of the two overlapping bursts.

By default the 2 photon selections are D+A photons during D excitation (`Ph_sel(Dex='DAem')`) and A photons during A excitation (`Ph_sel(Aex='Aex')`).

**Parameters**

- **dx** (*Data object*) – contains the data on which to perform the burst search. Background estimation must be performed before the search.
- **F** (*float or tuple*) – burst search parameter F. If it is a 2-element tuple, specifies F separately for `ph_sel1` and `ph_sel2`.
- **m** (*int or tuple*) – Burst search parameter m. If it is a 2-element tuple, specifies m separately for `ph_sel1` and `ph_sel2`.
- **min_rate_cps** (*float or tuple*) – min. rate in cps for burst detection. If not None, min_rate_cps overrides any value passed in `F`. If a 2-element tuple specifies min_rate_cps separately for `ph_sel1` and `ph_sel2`. In multispot data, it can also be an array (or a 2-tuple or arrays) with size equal to the number of spots.
- **c** (*float*) – parameter used set the definition of the rate estimatator. See `c` parameter in *`burstlib.Data.burst_search()`* for details.
- **ph_sel1** (*Ph_sel object*) – photon selections used for bursts search 1.
- **ph_sel2** (*Ph_sel object*) – photon selections used for bursts search 2.
- **mute** (*bool*) – if True nothing is printed. Default: False.

**Returns**
A new `Data` object containing bursts from the and-gate search.

See also *`fretbursts.burstlib.Data.burst_search()`*.

`fretbursts.burstlib_ext.`**`bursts_fitter`**(*dx*, *burst_data='E'*, *save_fitter=True*, *weights=None*, *gamma=1*, *add_naa=False*, *skip_ch=None*, *binwidth=None*, *bandwidth=None*, *model=None*, *verbose=False*, *fit_tot=True*)

Create a mfit.MultiFitter object (for E or S) add it to `dx`.

A MultiFitter object allows to fit multi-channel data with the same model.

**Parameters**

- **dx** (*Data*) – `Data` object containing the FRET data
- **save_fitter** (*bool*) – if True save the `MultiFitter` object in the `dx` object with name: burst_data + '_fitter'.

- **burst_data** (*string*) – name of burst-data attribute (i.e 'E' or 'S').
- **weights** (*string or None*) – kind of burst-size weights. See `fretbursts.fret_fit.get_weights()`.
- **gamma** (*float*) – gamma factor passed to `get_weights()`.
- **add_naa** (*bool*) – if True adds `naa` to the burst size.
- **binwidth** (*float or None*) – bin width used to compute the histogram. If None the histogram is not computed.
- **bandwidth** (*float or None*) – bandwidth used to compute the KDE If None the KDE is not computed.
- **model** (*lmfit.Model object or None*) – lmfit Model used for histogram fitting. If None the histogram is not fitted.
- **verbose** (*bool*) – if False avoids printing any output.
- **fit_tot** (*bool*) – whether to perform histogram fitting on combination of all arrays

**Returns**
The `mfit.MultiFitter` object with the specified burst-size weights.

`fretbursts.burstlib_ext.`**`calc_bg_brute`**(*dx*, *min_ph_delay_list=None*, *return_all=False*, *error_metrics='KS'*)

Compute background for all the ch, ph_sel and periods.

This function performs a brute-force search of the min ph delay threshold. The best threshold is the one the minimizes the error function. The best background fit is the rate fitted using the best threshold.

**Parameters**

- **min_ph_delay_list** (*sequence*) – sequence of values used for the brute-force search. Background and error will be computed for each value in `min_ph_delay_list`.
- **return_all** (*bool*) – if True return all the fitted backgrounds and error functions. Default False.
- **error_metrics** (*string*) – Specifies the error metric to use. See `fretbursts.background.exp_fit()` for more details.

**Returns**
Two arrays with best threshold (us) and best background. If `return_all = True` also returns the dictionaries containing all the fitted backgrounds and errors.

`fretbursts.burstlib_ext.`**`calc_bg_brute_cache`**(*dx*, *min_ph_delay_list=None*, *return_all=False*, *error_metrics='KS'*, *force_recompute=False*)

Compute background for all the ch, ph_sel and periods caching results.

This function performs a brute-force search of the min ph delay threshold. The best threshold is the one the minimizes the error function. The best background fit is the rate fitted using the best threshold.

Results are cached to disk and loaded transparently when needed. The cache file is an HDF5 file named `dx.fname[:-5] + '_BKG.hdf5'`.

**Parameters**

- **min_ph_delay_list** (*sequence*) – sequence of values used for the brute-force search. Background and error will be computed for each value in `min_ph_delay_list`.
- **return_all** (*bool*) – if True return all the fitted backgrounds and error functions. Default False.

- **error_metrics** (*string*) – Specifies the error metric to use. See `fretbursts.background.exp_fit()` for more details.
- **force_recompute** (*bool*) – if True, recompute results even if a cache is found.

**Returns**
Two arrays with best threshold (us) and best background. If `return_all = True` also returns the dictionaries containing all the fitted backgrounds and errors.

`fretbursts.burstlib_ext.`**`calc_mdelays_hist`**(*d*, *ich=0*, *m=10*, *period=(0, -1)*, *bins_s=(0, 10, 0.02)*, *ph_sel=Ph_sel(Dex='DAem', Aex='DAem')*, *bursts=False*, *bg_fit=True*, *bg_F=0.8*)

Compute histogram of m-photons delays (or waiting times).

**Parameters**

- **dx** (*Data object*) – contains the burst data to process.
- **ich** (*int*) – the channel number. Default 0.
- **m** (*int*) – number of photons used to compute each delay.
- **period** (*int or 2-element tuple*) – index of the period to use. If tuple, the period range between period[0] and period[1] (included) is used.
- **bins_s** (*3-element tuple*) – start, stop and step for the bins
- **ph_sel** (*Ph_sel object*) – photon selection to use.

**Returns**

*Tuple of values* –

- bin_x (array): array of bins centers
- histograms_y (array): arrays of histograms, contains 1 or 2 histograms (when `bursts` is False or True)
- bg_dist (random distribution): erlang distribution with same rate as background (kcps)
- a, rate_kcps (floats, optional): amplitude and rate for an Erlang distribution fitted to the histogram for bin_x > bg_mean*bg_F. Returned only if `bg_fit` is True.

`fretbursts.burstlib_ext.`**`calc_mean_lifetime`**(*dx*, *t1=0*, *t2=inf*, *ph_sel=Ph_sel(Dex='DAem', Aex='DAem')*)

Compute the mean lifetime in each burst.

**Parameters**

- **t1, t2** (*floats*) – min and max value (in TCSPC bin units) for the nanotime to be included in the mean
- **ph_sel** (*Ph_sel object*) – object defining the photon selection. See `fretbursts.ph_sel` for details.

**Returns**
List of arrays of per-burst mean lifetime. One array per channel.

`fretbursts.burstlib_ext.`**`fit_bursts_kde_peak`**(*dx*, *burst_data='E'*, *bandwidth=0.03*, *weights=None*, *gamma=1*, *add_naa=False*, *x_range=(-0.1, 1.1)*, *x_ax=None*, *save_fitter=True*)

Fit burst data (typ. E or S) by finding the KDE max on all the channels.

**Parameters**

- **dx** (*Data*) – `Data` object containing the FRET data
- **burst_data** (*string*) – name of burst-data attribute (i.e 'E' or 'S').
- **bandwidth** (*float*) – bandwidth for the Kernel Density Estimation
- **weights** (*string or None*) – kind of burst-size weights. See `fretbursts.fret_fit.get_weights()`.
- **gamma** (*float*) – gamma factor passed to `get_weights()`.
- **add_naa** (*bool*) – if True adds `naa` to the burst size.
- **save_fitter** (*bool*) – if True save the `MultiFitter` object in the `dx` object with name: burst_data + '_fitter'.
- **x_range** (*tuple of floats*) – min-max range where to search for the peak. Used to select a single peak in a multi-peaks distribution.
- **x_ax** (*array or None*) – x-axis used to evaluate the Kernel Density

**Returns**
An array of max peak positions (one per ch). If the number of channels is 1 returns a scalar.

`fretbursts.burstlib_ext.`**`get_burst_photons`**(*d*, *ich=0*, *ph_sel=Ph_sel(Dex='DAem', Aex='DAem')*)

Return a list of arrays of photon timestamps in each burst.

Deprecated since version 0.6.5: Use `burst_photons()` instead.

**Parameters**

- **d** (*Data*) – Data() object
- **ich** (*int*) – channel index
- **ph_sel** (*Ph_sel*) – photon selection. It allows to select timestamps from a specific photon selection. Example ph_sel=Ph_sel(Dex='Dem'). See `fretbursts.ph_sel` for details.

**Returns**
A list of arrays of photon timestamps (one array per burst).

`fretbursts.burstlib_ext.`**`group_data`**(*d_list*)

Group a list of data objects into single data object as though each data object was a different set of set of spots, so the returned data object appears as a multi-spot experiment. Usefull for joining technical repeats, especially so that background calculation etc can be calculated equally on all members. This serves as an alternative to `join_data()` but has some key differences: 1. Data objects can be at any state of analysis 2. Photon data is maintained, so reassesment of background, burst search etc. all remain possible 3. Each data object is treated as a separate spot, so burst arrays are not concatenated

**Parameters**
**d_list** (*list of Data*) – A list of data objects to be grouped into single data object

**Raises**

- **`RuntimeError`** – Mismatched fields indicating data are not technical repeats of each other
- **`ValueError`** – Inconsistent FRET correction factors

**Returns**
**new_d** (*Data*) – Data object of inputs grouped into new multi-spot measurment.

`fretbursts.burstlib_ext.`**`histogram_mdelays`**(*d*, *ich=0*, *m=10*, *ph_sel=Ph_sel(Dex='DAem', Aex='DAem')*, *binwidth=0.001*, *dt_max=0.01*, *bins=None*, *inbursts=False*)

Compute histogram of m-photons delays (or waiting times).

**Parameters**

- **dx** (*Data object*) – contains the burst data to process.
- **ich** (*int*) – the channel number. Default 0.
- **m** (*int*) – number of photons used to compute each delay.
- **ph_sel** (*Ph_sel object*) – photon selection to use.
- **inbursts** (*bool*) – if True, compute the histogram with only photons in bursts.

**Returns**
A `HistData` object containing the computed histogram.

`fretbursts.burstlib_ext.`**`join_data`**(*d_list*, *gap=0*)

Joins burst data of different measurements in a single `Data` object.

Merge a list of `Data` objects (i.e. a set of different measurements) into a single `Data` object containing all the bursts (like it was a single acquisition). The `Data` objects to be merged need to already contain burst data. The input `Data` objects are required to have undergone background estimation (all with the same background period) and burst search. For each measurement, the time of burst start is offset by the duration of the previous measurement + an additional `gap` (which is 0 by default).

The index of the first/last photon in the burst (`istart` and `iend`) are kept unmodified and refer to the original timestamp array. The timestamp arrays are not copied: the new `Data` object will not contain any timestamp arrays (ph_times_m). This may cause errors when calling functions that require the timestamps data such as burst search.

The background arrays (bg, bg_dd, etc…) are concatenated. The burst attribute `bp` is updated to refer to these new concatenated arrays. The attributes `Lim` and `Ph_p` are concatenated and left unchanged. Therefore different sections will refer to different original timestamp arrays. The retuned `Data` object will have a new attribute `i_origin`, containing, for each burst, the index of the original data object in the list.

**Parameters**

- **d_list** (*list of Data objects*) – the list of measurements to concatenate.
- **gap** (*float*) – the time delay (or gap) in seconds to add to each concatenated measurement.

**Returns**
A `Data` object containing bursts from the all the objects in `d_list`. This object will not contain timestamps, therefore it is possible to perform burst selections but not a new burst search.

---

**Example**

If `d1` and `d2` are two measurements to concatenate:

```
file_list = ['filename1', 'filename2']
d_list = [loader.photon_hdf5(f) for f in file_list]

for dx in d_list:
    loader.alex_apply_period(dx)
    dx.calc_bg(bg.exp_fit, time_s=30, tail_min_us='auto', F_bg=1.7)
    dx.burst_search()

d_merged = bext.join_data(d_list)
```

`d_merged` will contain bursts from both input files.

---

`fretbursts.burstlib_ext.moving_window_chunks(dx, start, stop, step, window=None, time_zero=0)`

Return a list of Data object, each containing bursts in one time-window.

Each returned Data object contains only bursts lying in the current time-window. Additionally, the start/stop values of current time-window are saved in `Data`'s attributes: name, slice_tstart and slice_tstop.

**Parameters**

- **dx** (*Data*) – the Data() object to be sliced with a moving window.
- **start, stop** (*scalars*) – time-range in seconds spanned by the moving window.
- **step** (*scalar*) – window time-shift at each step.
- **window** (*scalar*) – window duration. If None, window = step.
- **time_zero** (*scalar*) – shift the start/stop times saved in the Data attributes so that "time zero" falls at `time_zero` seconds. Default 0, no shift.

**Returns**

A list of Data objects, one for each window position.

See also: `moving_window_dataframe()`.

`fretbursts.burstlib_ext.moving_window_dataframe(start, stop, step, window=None, time_zero=0)`

Create a DataFrame for storing moving-window data.

Create and return a DataFrame for storing columns of moving-window data. Three columns are initialize with "time axis" data: 'tstart', 'tstop' and 'tmean'. The returned DataFrame is typically used to store (in new columns) quantities as function of the moving time-window. Examples of such quantities are number of bursts, mean burst size/duration, fitted E peak position, etc.

**Parameters**

- **start, stop** (*scalars*) – range spanned by the moving window.
- **step** (*scalar*) – window shift at each "step".
- **window** (*scalar*) – window duration. If None, window = step.

**Returns**

DataFrame with 3 columns (tstart, tstop, tmean), one row for each window position.

See also: `moving_window_chunks()`.

`fretbursts.burstlib_ext.moving_window_startstop(start, stop, step, window=None)`

Computes list of (start, stop) values defining a moving-window.

**Parameters**

- **start, stop** (*scalars*) – range spanned by the moving window.
- **step** (*scalar*) – window shift at each "step".
- **window** (*scalar*) – window duration. If None, window = step.

**Returns**

A list of (start, stop) values for the defined moving-window range.

`fretbursts.burstlib_ext.ph_burst_stats(d, ich=0, func=<function mean>, ph_sel=Ph_sel(Dex='DAem', Aex='DAem'))`

Applies function `func` to the timestamps of each burst.

**Parameters**

- **d** (*Data*) – Data() object

- **ich** (*int*) – channel index
- **func** (*function*) – a function that take an array of burst-timestamps and return a scalar. Default `numpy.mean`.
- **ph_sel** (*Ph_sel*) – photon selection. It allows to select timestamps from a specific photon selection. Default Ph_sel('all'). See `fretbursts.ph_sel` for details.

**Returns**
An array containing per-burst timestamp statistics.

## 3.12 Why an HDF5-based smFRET file format

In this page we briefly introduce what the HDF5 format is and why it is important for single-molecule FRET data.

### 3.12.1 What is HDF5?

HDF5 is standard and general-purposes container-format for binary data (see also HDF on Wikipedia). The format can store any number of multi-dimensional arrays with no size limit in a hierarchical fashion (i.e. arrays can be put in folders and subfolders called groups). Any dataset or folder can have metadata attached to it (for example a description, a date, or an array of parameters).

The format is self-describing, so any HDF5 compatible application can read the file content without knowing in advance the data-type (i.e. int32 or float) or the byte layout (i.e. big-endian little-endian).

HDF5 supports transparent data compression using the zlib algorithm or any third-party algorithm via plugins.

The format is an open standard supported by the non-profit organization HDFGroup. Open-sources libraries to read the format are available for all the main programming languages.

### 3.12.2 The HDF5 ecosystem

Numerous organizations use HDF5. Just as an example, the native MATLAB format (.mat) is HDF5-based from version 7.3 on.

Libraries to read the HDF5 format exist for the majority of programming languages. Among the others, FORTRAN, C, C++, C#, Java, MATLAB, Python, Mathematica, R have first-class support for the format.

LabView can read/write the format using h5labview.

Origin natively support HDF5 from version 8.1.

Open-source and multi-platform viewers/editors are also available (see HDFView and ViTables).

Python, in particular, has 2 libraries that allow handling HDF5 files:

- h5py
- pytables

FRETBursts uses **pyTables**.

### 3.12.3 Why HDF5 and smFRET?

Most of smFRET data around the world is acquired through a custom setup and custom software. As a result the number of file formats is almost as large as the number of existing setups.

A single, space-efficient and self-documenting file format like HDF5 is highly preferable to the Babel of formats used today.

Numerous advantages can be easily envisioned:

- **Efficiency**: HDF5 is highly efficient both for space and speed. Libraries to interoperate with the format are broadly used and heavily tested. Scientists don't need to reinvent the wheel and can leverage the already available state-of-the art software technologies.
- **Long-term persistence:** in 5-10-20 years the data can be always read without relying on obscure, poorly document, (or in some case vendor specific) binary formats.
- **Easy interoperability:** a single format lowers the barriers for data-exchange and collaboration. A single format makes easier to compare the output of different analysis software, encourages reproducibility and foster collaboration between different groups.

### 3.12.4 HDF5 in FRETBursts

FRETBursts allows saving and loading smFRET data from and to an HDF5-based file format called **Photon-HDF5**.

The **Photon-HDF5** is a pre-defined layout to be used with smFRET and other data involving time-series of photon-data.

A description of the Photon-HDF5 format and its specifications can be found in Photon-HDF5 format.

For documentation on using the Photon-HDF5 format in *FRETBursts* see:

#### HDF5-based smFRET file format

We developed an HDF5-based format called **Photon-HDF5** for smFRET and other measurements involving series of photon timestamps. The specifications of the Photon-HDF5 format can be found in Photon-HDF5 format.

For a general overview on the importance of a standard file format for smFRET see also *Why an HDF5-based smFRET file format*.

#### Read and write HDF5 smFRET files

To load a smFRET data contained in HDF5-Ph-Data use the function `loader.photon_hdf5()`.

You can convert files from any format to Photon-HDF5 by using phconvert (already pre-installed with FRETBursts).

## 3.13 FRET Correction Formulas

The `fretmath` module contains functions to compute corrected FRET efficiency from the proximity ratio and vice-versa.

For derivation see notebook: "Derivation of FRET and S correction formulas.ipynb" (link).

`fretbursts.fretmath.`**`correct_E_gamma_leak_dir`**(*Eraw*, *gamma=1*, *leakage=0*, *dir_ex_t=0*)

Compute corrected FRET efficiency from proximity ratio `Eraw`.

For the inverse function see *uncorrect_E_gamma_leak_dir()*.

**Parameters**

- **Eraw** (*float or array*) – proximity ratio (only background correction, no gamma, leakage or direct excitation)
- **gamma** (*float*) – gamma factor
- **leakage** (*float*) – leakage coefficient
- **dir_ex_t** (*float*) – coefficient expressing the direct excitation as n_dir = dir_ex_t * (na + gamma*nd). In terms of physical parameters it is the ratio of acceptor over donor absorption cross-sections at the donor-excitation wavelength.

**Returns**

Corrected FRET effciency

`fretbursts.fretmath.`**`correct_S`**(*Eraw*, *Sraw*, *gamma*, *leakage*, *dir_ex_t*)

Correct S values for gamma, leakage and direct excitation.

**Parameters**

- **Eraw** (*scalar or array*) – uncorrected ("raw") E after only background correction (no gamma, leakage or direct excitation).
- **Sraw** (*scalar or array*) – uncorrected ("raw") S after only background correction (no gamma, leakage or direct excitation).
- **gamma** (*float*) – gamma factor.
- **leakage** (*float*) – donor emission leakage into the acceptor channel.
- **dir_ex_t** (*float*) – direct acceptor excitation by donor laser. Defined as `n_dir = dir_ex_t * (na + g nd)`. The dir_ex_t coefficient is the ratio between D and A absorption cross-sections at the donor-excitation wavelength.

**Returns**

Corrected S (stoichiometry), same size as `Sraw`.

`fretbursts.fretmath.`**`dir_ex_correct_E`**(*Eraw*, *dir_ex_t*)

Apply direct excitation correction to the uncorrected FRET `Eraw`.

The coefficient `dir_ex_t` expresses the direct excitation as n_dir = dir_ex_t * (na + gamma*nd). In terms of physical parameters it is the ratio of acceptor over donor absorption cross-sections at the donor-excitation wavelength.

For the inverse see *dir_ex_uncorrect_E()*.

`fretbursts.fretmath.`**`dir_ex_uncorrect_E`**(*E*, *dir_ex_t*)

Reverse direct excitation correction and return uncorrected FRET.

For the inverse see *dir_ex_correct_E()*.

`fretbursts.fretmath.`**`gamma_correct_E`**(*Eraw*, *gamma*)

Apply gamma correction to the uncorrected FRET `Eraw`.

For the inverse see *gamma_uncorrect_E()*.

`fretbursts.fretmath.`**`gamma_uncorrect_E`**`(`*E*, *gamma*`)`

Reverse gamma correction and return uncorrected FRET.

For the inverse see *`gamma_correct_E()`*.

`fretbursts.fretmath.`**`leakage_correct_E`**`(`*Eraw*, *leakage*`)`

Apply leakage correction to the uncorrected FRET `Eraw`.

For the inverse see *`leakage_uncorrect_E()`*.

`fretbursts.fretmath.`**`leakage_uncorrect_E`**`(`*E*, *leakage*`)`

Reverse leakage correction and return uncorrected FRET.

For the inverse see *`leakage_correct_E()`*.

`fretbursts.fretmath.`**`test_fretmath`**`()`

Run a few consistency checks for the correction functions.

`fretbursts.fretmath.`**`uncorrect_E_gamma_leak_dir`**`(`*E*, *gamma=1*, *leakage=0*, *dir_ex_t=0*`)`

Compute proximity ratio from corrected FRET efficiency `E`.

This function is the inverse of *`correct_E_gamma_leak_dir()`*.

**Parameters**

- **E** (*float or array*) – corrected FRET efficiency
- **gamma** (*float*) – gamma factor
- **leakage** (*float*) – leakage coefficient
- **dir_ex_t** (*float*) – direct excitation coefficient expressed as n_dir = dir_ex_t * (na + gamma*nd). In terms of physical parameters it is the ratio of absorption cross-section at donor-excitation wavelengths of acceptor over donor.

**Returns**

Proximity ratio (reverses gamma, leakage and direct excitation)

`fretbursts.fretmath.`**`uncorrect_S`**`(`*E_R*, *S*, *gamma*, *L_k*, *d_dirT*`)`

Function used to test *`correct_S()`*.

## 3.14 Description of the files

Here a brief descriprion of the main FRETBursts files.

### 3.14.1 `burstlib.py`

This module contains all the main FRETBursts analysis functions.

`burstslib.py` defines the fundamental object `Data()` that contains both the experimental data (attributes) and the high-level analysis routines (methods).

Furthermore it loads all the remaining **FRETBursts** modules (except for `loaders.py`).

For usage example see the IPython Notebooks in sub-folder "notebooks".

### 3.14.2 loader.py

The `loader` module contains functions to load each supported data format. The loader functions load data from a specific format and return a new *fretbursts.burstlib.Data()* object containing the data.

This module contains the high-level function to load a data-file and to return a `Data()` object. The low-level functions that perform the binary loading and preprocessing can be found in the `dataload` folder.

### 3.14.3 select_bursts.py

See *fretbursts.select_bursts*.

### 3.14.4 burst_plot.py

This module defines all the plotting functions for the *fretbursts.burstlib.Data* object.

The main plot function is `dplot()` that takes, as parameters, a `Data()` object and a 1-ch-plot-function and creates a subplot for each channel.

The 1-ch plot functions are usually called through `dplot` but can also be called directly to make a single channel plot.

The 1-ch plot functions names all start with the plot type (`timetrace`, `ratetrace`, `hist` or `scatter`).

**Example 1** - Plot the timetrace for all ch:

```
dplot(d, timetrace, scroll=True)
```

**Example 2** - Plot a FRET histogramm for each ch with a fit overlay:

```
dplot(d, hist_fret, show_model=True)
```

For more examples refer to FRETBurst notebooks.

### 3.14.5 background.py

Routines to compute the background from an array of timestamps. This module is normally imported as `bg` when fretbursts is imported.

The important functions are *exp_fit()* and *exp_cdf_fit()* that provide two (fast) algorithms to estimate the background without binning. These functions are not usually called directly but passed to `Data.calc_bg()` to compute the background of a measurement.

See also *exp_hist_fit()* for background estimation using an histogram fit.

### 3.14.6 phtools (folder)

This folder contains the core functions to manipulate timestamps, including burst search and photon rates computations. Additionally, data structures for storing and manipulating bursts data are provided.

Burst search and photon counting functions (to count number of donor and acceptor photons in each burts) are provided both as a pure python implementation and as an optimized Cython (compiled) version. The cython version is usually 10 or 20 times faster. `burstlib.py` will load the Cython functions, falling back to the pure python version if the compiled version is not found.

### 3.14.7 `dataload` (folder)

This folder contains one file per each supported data file.

Each file contains the binary load and preprocessing functions needed for the specific format. Functions defined here are used by loader functions in `loaders.py` to properly initialize a `Data()` variable.

### 3.14.8 `fit` (folder)

This folder contains generic fit functions for Gaussian and exponential fit of a sample.

See *Fit framework*.

## 3.15 FRETBursts Cython extensions

Cython is a tool that, among other things, allows to translate annotated python code into C code. The C code can be then compiled into a dynamic library and transparently called from python like any other python library, but with the advantage of a much higher execution speed.

For some core burst-search functions FRETBursts includes both a pure pyhton and a cython version. At import time, the code looks for the compiled version and, if not found, falls back to the pure python version. Therefore, although the compiled cython version is completely optional, it allows to gain significant execution speed in core functions that are potentially executed many times.

Usually the cython extensions are compiled during installation. To manually build the extensions type:

```
python setup.py build
```

from the FRETBursts source folder.

# BIBLIOGRAPHY


[pax] Doose *et al.* European Biophysics Journal 36(6) p.669-674, **2007**. DOI:10.1007/s00249-007-0133-7

[48spot] Ingargiola *et al.* bioRxiv 156182, **2017**. DOI:10.1101/156182

# PYTHON MODULE INDEX

## f

# INDEX

## E



## F

# G



# H



# I



# J



# K



# L



# M

# N



# P



# R

# S



# T



# U



# W