## Contents

1 Table of contents 3  
  1.1 Getting started ........................................... 3  
  1.2 FRETbursts Reference Manual .......................... 7  
  1.3 Compiling Cython code .................................. 52  

2 Indices and tables 55  

Python Module Index 57
FRETBursts is an open-source toolkit for analysis of single-molecule FRET data acquired by single and multi-spot confocal systems.

For more general info you may want to visit FRETBursts Homepage. The source code can be found in the FRETBursts repository on GitHub. A set of IPython Notebook tutorials can be found in the FRETBursts_notebooks repository.

In the following pages you can find installation instructions and the reference and API documentation.
1.1 Getting started

1.1.1 Installation

FRETBursts is distributed as source code. The installation consists in installing a scientific python distribution, downloading FRETBursts sources, and running the Installation notebook.

Windows Quick Installation steps

If you are not familiar with Python and Git and you are using MS Windows follows these steps:

1. Install Continuum Anaconda (a scientific python distribution).
2. Install SourceTree (GUI for Git).
3. Download FRETBursts using SourceTree (see below Downloading FRETBursts).

4. Open FRETBursts folder, enter the notebooks folder and double-click “Launch IPython Notebook Here.bat”.

5. A new browser windows will open. Click on FRETBursts Installation and run the notebook. Follow the instruction there to run the tutorials notebooks.

For more info and customizations continue reading.
Installing Python for scientific computing

On all the main platforms, the easiest way to install python and all the scientific packages is using a python distribution like Continuum Anaconda or Enthought Canopy. Both distributions include, in the free version, all the needed software (and much more).

A pure-python dependency, lmfit, is not installed by default by Anaconda and will be installed by FRETBursts Installation notebook.

FRETBursts has been tested on Anaconda 1.9 or newer.

Manual Dependencies

If you prefer a manual installation, FRETBursts dependencies are:

- Python 2.7
- Numpy/Scipy (any version from 2013 on)
- Matplotlib (1.3.x or greater) with QT4 backend (either PyQt4 or PySide).
- IPython 1.x (2.x recommended)
- PyTables 3.x (optional). To load/save the HDF5 Ph-Data.
- lmfit: (version 0.8rc3 or higher) for powerful and user-friendly fitting
- Pandas (optional) currently used only by the fitting framework
- a modern browser (Chrome suggested)

For developing FRETBursts you should also install

- sphinx 1.2.2 with the napoleon extension (sphinxcontrib-napoleon) to build this documentation.
- pytest to execute the unit tests.

Installing Git (optional)

FRETBursts uses Git as revision control system. Even if not necessary, we strongly recommend installing it because FRETBursts notebooks can keep track of the FRETBursts software revision. Furthermore, Git will make easy downloading future updates.

Unless you are familiar with Git it is preferable to install a graphical interface like SourceTree.

On Windows, install SourceTree and, when asked, select the single-user installation and choose to download the embedded Git. Alternatively, for an independent system-wide Git installation, download the windows binaries from the Git Homepage.

On Mac OSX, install SourceTree and configure it to use a system-wide Git installation. Git can be installed system-wide using the homebrew package manager.

On Linux Git is easily installed and usually comes with the gitk graphical interface.

Downloading FRETBursts

You can download a simple ZIP-ball containing FRETBursts by clicking on Download ZIP on FRETBursts Homepage on GitHub.

However the preferred way is downloading FRETBursts through Git (in other words “cloning FRETBursts”). In this case copy the clone URL is:
https://github.com/tritemio/FRETBursts.git

When using SourceTree, click on Clone/New -> Clone Repository and paste the clone URL in Source Path/URL. You can choose where to put the sources.

From the command line, type:

git clone https://github.com/tritemio/FRETBursts.git

Installing FRETBursts

Strictly speaking FRETBursts is not installed as it runs from the folder where you download it. However some optional dependencies and a configuration file is created by running the “FRETBursts Installation” notebook that you find in the notebooks folder.

To run the FRETBursts Installation notebook:

- On windows, click on “Launch IPython Notebook Server Here.bat” (inside the notebooks folder) and then click on “FRETBursts Installation”.
- On the other platforms:

cd notebook_folder
ipython notebook

Note: Once the configuration is done, you can load FRETBursts in any notebook by running run load_fretbursts. Note that you need a copy of the load_fretbursts.py script in the notebook folder.

Running FRETBursts

We recommend starting by running the tutorial notebooks. The easiest way to perform a new analysis is to modify (or copy) one of the notebooks.

To run the FRETBursts notebooks, download and decompress the ZIP-ball in a folder and launch an IPython Notebook server inside that folder. For more details see [IPython Notebook Startup Folder](http://fretbursts.readthedocs.org/installation.html#ipython-notebook-startup-folder) on FRETBursts documentation.

On the first run, the tutorial notebooks will automatically download some public datasets of smFRET measurements that are provided for testing demonstration.

These datasets are free to use for any purposes (CC0 license). If you use these datasets please cite as:

- Ingargiola, Antonino; Chung, Sangyoon (2014): smFRET example datasets for the FRETBursts software. figshare. DOI 10.6084/m9.figshare.1019906

IPython Notebook startup folder

To use the IPython Notebook you have to launch a local notebook server in the folder containing the notebooks files (or in a parent folder).

On windows (Anaconda), you can copy and modify the IPython launcher you find in the start menu. To change the startup folder right click on the IPython Notebook icon -> Properties, and set the new folder in the Start in field.

On all the platforms, you can start IPython Notebook from the terminal (cmd.exe on Windows) with:

1.1. Getting started
cd notebook_folder
ipython notebook

Note: The preferred browser is Chrome or Firefox. The use of MS Explorer is discouraged as its implementation of web standards is incomplete and not compliant.

Installing a compiler (obsolete)

Warning: This paragraph is retained for historical reasons and because it may be useful for some user. However with recent versions of Anaconda the compiler is included and these steps are not necessary anymore.

Some core burst-search core functions can be optionally compiled to gain significant execution speed. This process requires a compiler to be installed.

On Linux the preferred compiler is GCC, that is easily available for any distribution.

On Windows, the MS Visual Studio compiler is preferred. To install it search on internet for the files VS2008ExpressWithSP1ENUX1504728.iso and GRMSDKX_EN_DVD.iso.

On Mac OSX you should install the LLVM compiler included in Xcode.

See also:

- Compiling Cython code

1.1.2 Notebook-based workflow

IPython Notebook is the recommended environment to perform interactive analysis with FRETBursts.

Tutorials for FRETBursts are provided as IPython Notebooks.

Typically, a new analysis is performed making a copy on an existing notebook (used as a template) and applying all the needed modifications.

The FRETBursts notebooks are “revision-control aware”, meaning that the exact FRETBursts revision used during each execution is stored (and displayed) at load time. Saving the software revision together with analysis commands and results allows long term reproducibility and provides a lightweight approach for regression testing.

The following sections describe how to configure and use the IPython notebooks to perform analysis with FRETBursts.

Loading FRETBursts from a notebook

From inside a notebook, FRETBursts is loaded running a small script (load_fretbursts.py) placed in the notebooks folder. For this reason, a typical FRETBursts notebook always starts with the line:

%run load_fretbursts

The script switches from notebook folder to FRETBursts source folder and loads the software.

Before the first execution, you have to tell the script where the FRETBursts source folder is. You can either paste the folder name in load_fretbursts.py or set an environment variable named FRETBURSTS_DIR containing the path.
1.2 FRETBursts Reference Manual

Contents:

1.2.1 Loader functions

This 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 file 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.

```python
def fretbursts.loader.hdf5(fname):
    # Load a data file saved in HDF5-Ph-Data format version 0.2 or higher.
    # Any fretbursts.burstlib.Data object can be saved in HDF5 format using fretbursts.hdf5.store().
    # For description and specs of the HDF5-Ph-Data format see: https://github.com/tritemio/FRETBursts/wiki/HDF5-Ph-Data-format-0.2-Draft
```

```python
def fretbursts.loader.hdf5_legacy(fname):
    # Load a data file saved in the old HDF5 smFRET format version 0.1.
    # For new data use hdf5() instead.
```

```python
def fretbursts.loader.multispot48(fname, leakage=0, gamma=1.0, reprocess=False, i_start=0, i_stop=None, debug=False):
    # Load a 48-ch multispot file and return a Data() object.
```

```python
def fretbursts.loader.multispot48_simple(fname, leakage=0, gamma=1.0, i_start=0, i_stop=None, debug=False):
    # Load a 48-ch multispot file and return a Data() object.
```

```python
def fretbursts.loader.multispot8(fname, bytes_to_read=-1, swap_D_A=True, leakage=0, gamma=1.0):
    # Load a 8-ch multispot file and return a Data() object. Cached version.
```

```python
def fretbursts.loader.multispot8_core(fname, bytes_to_read=-1, swap_D_A=True, leakage=0, gamma=1.0):
    # Load a 8-ch multispot file and return a Data() object.
```

```python
def fretbursts.loader.nsalex(fname, leakage=0, gamma=1.0):
    # Load a nsALEX 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))
    # nsalex_plot_alternation(d)
    # If the plot looks good apply the alternation with:
    # loader.nsalex_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 with \texttt{nsalex()} and then to set the alternation parameters using \texttt{d.add()}. 

The pattern to load nsALEX data is the following:

\begin{verbatim}
d = loader.nsalex(fname=fname)
d.add(D_ON=(2850, 580), A_ON=(900, 2580))
salex_plot_alternation(d)
\end{verbatim}

If the plot looks good apply the alternation with:

\begin{verbatim}
loader.nsalex_apply_period(d)
\end{verbatim}

Now \(d\) is ready for further processing such as background estimation, burst search, etc...

fretbursts.loader.usalex \((fname, leakage=0, gamma=1.0, header=166, bytes\_to\_read=-1, BT=None)\)
Load a usALEX 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:

\begin{verbatim}
d = loader.usalex(fname=fname)
d.add(D_ON=(2850, 580), A_ON=(900, 2580), alex_period=4000)
plot_alternation_hist(d)
\end{verbatim}

If the plot looks good apply the alternation with:

\begin{verbatim}
loader.usalex_apply_period(d)
\end{verbatim}

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 with \texttt{usalex()} and then to set the alternation parameters using \texttt{d.add()}. 

The pattern to load usALEX data is the following:

\begin{verbatim}
d = loader.usalex(fname=fname)
d.add(D_ON=(2850, 580), A_ON=(900, 2580), alex_period=4000)
plot_alternation_hist(d)
\end{verbatim}

If the plot looks good apply the alternation with:

\begin{verbatim}
loader.usalex_apply_period(d)
\end{verbatim}

Now \(d\) is ready for further processing such as background estimation, burst search, etc...

### 1.2.2 The “Data()” class

The \texttt{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

- The “Data()” class
  - “Data()” class: description and attributes
  - Analysis methods
  - Basic info methods
  - Burst corrections methods
  - Other burst methods
  - Utility methods
  - Photon selection

“Data()” class: description and attributes

A description of the Data class and its main attributes.

```python
class fretbursts.burstlib.Data(**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 by some methods (.copy(), .fuse_bursts(), etc...) or by a “burst selection” with Sel() function.

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**
  
  string
  
  measurements file name

- **nch**
  
  int
  
  number of channels

- **clk_p**
  
  float
  
  clock period in seconds for timestamps in ph_times_m

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

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

float or array of floats

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

gamma

float or array of floats

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

D_em

list of boolean arrays

[ALEX-only] boolean mask for \( .ph\times_m[i] \) for donor emission

D_ex, A_ex (list of boolean arrays): **[ALEX-only]**

boolean mask for \( .ph\times_m[i] \) during donor or acceptor excitation

D_ON, A_ON (2-element tuples of int ): **[ALEX-only]**

start-end values for donor and acceptor excitation selection.

alex_period

int

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

Background Attributes

These attributes contain the estimated background rate. Each attribute is a list (one element per channel) of arrays. Each array contains several rates computed every \( \text{time}_s \) seconds of measurements. \( \text{time}_s \) is an argument passed to \( .\text{calc}_bg() \) method. Each \( \text{time}_s \) measurement slice is here called background 'period'.

bg

list of arrays

total background (donor + acceptor) during donor excitation

bg_dd

list of arrays

background of donor emission during donor excitation

bg_ad

list of arrays

background of acceptor emission during donor excitation

bg_aa

list of arrays

background of acceptor emission during acceptor excitation

nperiods

int

number of periods in which timestamps are split for background calculation

bg_fun

function

function used to compute the background rates

Lim

list
each element of this list is a list of index pairs for \texttt{.ph\_times\_m[i]} for \textbf{first} and \textbf{last} photon in each period.

\texttt{Ph\_p}
\begin{itemize}
  \item \texttt{list}
\end{itemize}

each element in this list is a list of timestamps pairs for \textbf{first} and \textbf{last} photon of each period.

\texttt{bg\_ph\_sel}
\begin{itemize}
  \item \texttt{Ph\_sel object}
\end{itemize}

photon selection used by Lim and \texttt{Ph\_p}. See \texttt{fretbursts.ph\_sel.Ph\_sel} for details.

Other attributes (per-ch mean of \texttt{bg, bg\_dd, bg\_ad} and \texttt{bg\_aa}):

\begin{itemize}
  \item \texttt{rate\_m: array of bg rates for D+A channel pairs (ex. 4 for 4 spots)}
  \item \texttt{rate\_dd: array of bg rates for D em (and D ex if ALEX)}
  \item \texttt{rate\_da: array of bg rates for A em (and D ex if ALEX)}
  \item \texttt{rate\_aa: array of bg rates for A em and A ex (only for ALEX)}
\end{itemize}

\textbf{Burst search parameters (user input)}

These are the parameters used to perform the burst search (see \texttt{burst\_search\_t()}).

\texttt{ph\_sel}
\begin{itemize}
  \item \texttt{Ph\_sel object}
\end{itemize}

photon selection used for burst search. See \texttt{fretbursts.ph\_sel.Ph\_sel} for details.

\texttt{m}
\begin{itemize}
  \item \texttt{int}
\end{itemize}

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

\texttt{L}
\begin{itemize}
  \item \texttt{int}
\end{itemize}

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

\texttt{P}
\begin{itemize}
  \item \texttt{float, probability}
\end{itemize}

valid values [0..1]. Probability that a burst-start is due to a Poisson background. The employed Poisson rate is the one computed by \texttt{.calc\_bg()}. 

\texttt{F}
\begin{itemize}
  \item \texttt{float}
\end{itemize}

\((F * \text{background\_rate})\) is the minimum rate for burst-start

\textbf{Burst search data (available after burst search)}

When not specified, parameters marked as (list of arrays) contains arrays with one element per bursts. \texttt{mburst} arrays contain one “row” per burst. \texttt{TT} arrays contain one element per period (see above: background attributes).

\texttt{mburst}
\begin{itemize}
  \item \texttt{list of arrays}
\end{itemize}

list of 2-D arrays containing burst data. Each row contains data for 1 burst. Each column contains burst fields like burst start, end, duration, size and indexes. The proper way to access these fields is through the function \texttt{b\_*} (ex: \texttt{b\_start, b\_end, b\_width, b\_size, etc...}). For more details see \texttt{fretbursts.burstsearch.burstsearchlib}. 

1.2. FRETBursts Reference Manual 11
**TT**

*list of arrays*

List of arrays of \( T \) values (in sec.). A \( T \) value is the maximum delay between \( m \) photons to have a burst-start. Each channel has an array of \( T \) values, one for each background “period” (see above).

**T**

*array*

Per-channel mean of \( TT \)

**nd, na (list of arrays): number of donor or acceptor photons during donor excitation in each burst**

**nt**

*list of arrays*

Total number of photons (\( nd + na + nna \))

**aaaa**

*list of arrays*

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

**bp**

*list of arrays*

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

**bg bs**

*list*

Background rates used for threshold computation in burst search (is a reference to \( bg, bg_{dd}, \) or \( bg_{ad} \)).

**fuse**

*None or float*

If not None, the burst separation in ms below which bursts have been fused (see \( .fuse\_burst() \)).

**E**

*list*

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

**S**

*list*

Stoichiometry value for each burst: \( S = (\gamma nd + na) / (\gamma nd + na + nna) \)

---

**Analysis methods**

List of Data methods used to perform different analysis.

**class fretbursts.burstlib.Data**

**calc bg** \((fun, time_s=60, tail_min_us=500, F_bg=2)\)

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 is multiplied to compute the threshold.

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

**Example**

Compute background with `bg.exp_fit`, every 20s, with a threshold of 200us for photon waiting times:

```python
d.calc_bg(bg.exp_fit, time_s=20, tail_min_us=200)
```

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

**burst_search_t** (*L=10, m=10, P=None, F=6.0, min_rate_cps=None, nofret=False, max_rate=False, dither=False, ph_sel=Ph_sel(Dex='DAem', Aex='DAem'), verbose=False, mute=False, pure_python=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 minimum rate can be explicitly specified (`min_rate`) or computed as a function of the background rate (using `F` or `P`).

**Parameters**

• **m** (*int*) – number of consecutive photons used to compute the photon rate.

• **L** (*int*) – minimum number of photons in burst

• **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 meaningful.

• **F** (*float*) – if `P` is None: min.rate/bg.rate ratio for burst search else: `P` refers to a Poisson rate = `bg_rate*F`

• **min_rate** (*float or list/array*) – min. rate in cps for burst start. If not None has the precedence over `P` and `F`. If non-scalar, contains one rate per each channel.

• **ph_sel** (*Ph_sel object*) – object defining the photon selection used for burst search. Default: all photons. See `fretbursts.ph_sel.Ph_sel` for details.

• **pure_python** (*bool*) – if True, uses the pure python functions even when the optimized Cython functions are available.

**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_t(L=10, m=10, F=6)

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

calc_fret  (count_ph=False, corrections=True, dither=False, mute=False, pure_python=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 Ac-
ceptor photons, perform corrections (background, bleed-through), 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 Accep-
tor 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.

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 nna, nda) in self. Returns None.

fuse_bursts  (ms=0, process=True)
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.

Basic info methods

List of Data methods that output basic information.

class fretbursts.burstlib/Data
**time_max()**
Return the measurement time (last photon) in seconds.

**num_bursts()**
Return an array with the number of bursts in each channel.

**burst_sizes_ich(ich=0, gamma=1.0, gamma1=None, add_naa=False)**
Return gamma corrected burst sizes for channel ich.

The gamma corrected size is computed as:
1) \( nd + na/gamma \) (so th1 is the min. burst size for donly bursts)
2) \( nd \times gamma1 + na \) (so th1 is the min. burst size for high FRET bursts)

If \( gamma1 \) is not None, the definition (2) is used. If d.ALEX and add_naa are True, naa is added to the burst size.

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

**status(add='', noname=False)**
Return a string with burst search, corrections and selection info.

**name()**
Return short filename (last subfolder + fname with no extension)

**Name(add='')**
Return short filename + status information.

---

**Burst corrections methods**

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

**class fretbursts.burstlib.Data**

**update_leakage(leakage)**
Apply/update leakage (or bleed-through) correction.

**update_dir_ex(dir_ex)**
Apply/update direct excitation correction with value dir_ex.

**update_gamma(gamma)**
Change the gamma value and recompute FRET.

**background_correction_t(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 sizes (nd, na,...). lsb is the amplitude of dithering (if lsb=2 then noise is in -1..1).

---

**Other burst methods**

List of **Data** methods not falling in previous categories.

**class fretbursts.burstlib.Data**
calc_sbr \( (\text{ph\_sel=Ph\_sel(Dex='DAem', Aex='DAem')}, \text{gamma}=1.0) \)

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

**Parameters**

- **\text{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 \text{Ph\_sel('all')}, \text{Ph\_sel(Dex='Dem')}, \text{Ph\_sel(Dex='Aem')}. See \text{fretbursts.ph\_sel.Ph\_sel} for details.
- **\text{gamma (float)}** – gamma value used to compute corrected burst size in the case \text{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 \text{sbr attribute}.

**calc_max_rate** \( (m, \text{ph\_sel=Ph\_sel(Dex='DAem', Aex='DAem'))} \)

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

**Parameters**

- **\text{m (int)}** – number of timestamps to use to compute the rate
- **\text{ph\_sel (Ph\_sel object)}** – object defining the photon selection. See \text{fretbursts.ph\_sel.Ph\_sel} for details.

**fit_E_generic** \( (E1=-1, E2=2, \text{fit\_fun=<function two\_gaussian\_fit\_hist at 0x7f1a5ed94488>}, \text{weights=None, gamma=1.0}, \text{**fit\_kwargs}) \)

Fit E in each channel with \text{fit\_fun} using burst in [E1,E2] range. All the fitting functions are defined in \text{fretbursts.fit.gaussian\_fitting}.

**Parameters**

- **\text{weights (string or None)}** – specifies the type of weights If not None \text{weights will be passed to fret\_fit.get\_weights()}. \text{weights can be not-None only when using fit functions that accept weights (the ones ending in _hist or _EM)}
- **\text{gamma (float)}** – passed to \text{fret\_fit.get\_weights()} to compute weights

All the additional arguments are passed to \text{fit\_fun}. For example \text{p0 or mu\_fix} can be passed (see \text{fit.gaussian\_fitting} for details).

**Note:** Use this method for CDF/PDF or hist fitting. For EM fitting use \text{fit\_E\_two\_gauss\_EM}.

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

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):

\text{fit\_E\_m(weights='size')}
\text{fit\_E\_minimize(kind='E\_size', weights='sqrt')}

However \text{fit\_E\_minimize()} does not provide a model curve.

**fit_E\_ML\_poiss** \( (E1=-1, E2=2, \text{method=1}, \text{**kwargs}) \)

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

**fit_E\_minimize** \( (\text{kind='slope'}, E1=-1, E2=2, \text{**kwargs}) \)

Fit E using method \text{kind} ('slope' or 'E\_size') and bursts in [E1,E2] If \text{kind} is 'slope' the fit function is \text{fret\_fit.fit\_E\_slope()} If \text{kind} is 'E\_size' the fit function is \text{fret\_fit.fit\_E\_E\_size()} Additional arguments in \text{kwargs} are passed to the fit function.
**Utility methods**

List of Data methods used to get (or iterate over) the different arrays of timestamps or burst data.

```python
class fretbursts.burstlib.Data

get_ph_times (ich=0, ph_sel=Ph_sel(Dex='DAem', Aex='DAem'))
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.Ph_sel for details.

iter_ph_times (ph_sel=Ph_sel(Dex='DAem', Aex='DAem'))
Iterator that returns the arrays of timestamps in .ph_times_m.

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

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.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.Ph_sel for details.

expand (ich=0, alex_naa=False, width=False)
Return per-burst D and A sizes (nd, na) and background (bg_d, bg_a).

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 ()
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)

### Photon selection

The class `Ph_sel` is used to specify which sub-set of photons/timestamps are “selected” (i.e. all-photons, Donor-excitation-period photons, etc...).

```python
class fretbursts.ph_sel.Ph_sel
Class that describes a selection of photons.
```

This class takes two arguments ‘Dex’ and ‘Aex’. Valid values for the arguments are ‘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.

**Example**

- `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.
- `Ph_sel(Dex='Aem', Aex='Aem')` selects all the acceptor photons.

### 1.2.3 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 function `Select_bursts()` (or its alias `Sel()`). This function requires a `Data` object and a `selection function` as parameters. `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 just start from one of the functions in select_bursts.py and modify it to obtain a different criterium.

The reference documentation for the selection functions follows.

```python
fretbursts.burstlib.Select_bursts(d_orig, filter_fun, negate=False, nofret=False, **kwargs)
Uses filter_fun to select a sub-set of bursts from d_orig.
```

**Parameters**

- `d_orig` *(Data object)* – the original Data() object to filter
- `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.
- `nofret` *(boolean)* – If True do not recompute burst correction and FRET efficiencies in the new returned object. Default False.

**Kwargs:** Any additional keyword argument is passed to `filter_fun()`.
Returns \(d_{\text{new}}\) – a new Data() object containing the selected bursts.

**Warning:** This function is also available with the shortcut name `Sel`.

**Note:** In order to save RAM \(\text{ph\_times\_m}\) is shared between \(d_{\text{orig}}\) and \(d_{\text{new}}\). However, all the bursts data (\(\text{mburst}, \text{nd}, \text{na}, \text{etc.}\)) are new objects.

---

```python
fretbursts.burstlib.Sel()

Alias for `Select_bursts()`
```

```python
fretbursts.burstlib.Sel_mask(d_orig, filter_fun, negate=False, return_str=False, **kwargs)

Returns a list of nch masks to select bursts according to `filter_fun`. The passed ‘filter_fun’ is used to compute the mask for each channel.

Use this function only if you want to apply a selection from one object to a second object. Otherwise use `Select_bursts()`.

See also:

`Select_bursts()`, `Sel_mask_apply()`
```

```python
fretbursts.burstlib.Sel_mask_apply(d_orig, Masks, nofret=False, str_sel='')

Returns a new Data object with bursts select according to Masks. Note that ‘\(\text{ph\_times\_m}\)’ is shared to save RAM, but \(\text{mburst}, \text{nd}, \text{na}, \text{nt}, \text{bp}\) (and \(\text{naa}\) if ALEX) are new objects.

Use this function only if you want to apply a selection from one object to a second object. Otherwise use `Select_bursts()`.

See also:

`Select_bursts()`, `Sel_mask()`
```

---

**List of selection functions**

Functions to select bursts according to different criteria.

These functions are usually passed to the `burstlib.Sel()` as a second parameter. For example:

```python
ds = Sel(d, 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).

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

Select bursts with \(E\) between \(E1\) and \(E2\).
```

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

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

When \(\text{rect}\) is True the selection is rectangular otherwise is elliptical.
```

```python
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.
```

```python
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.
```

```python
fretbursts.select_bursts.S(d, ich=0, S1=-inf, S2=inf)

Select bursts with \(S\) between \(S1\) and \(S2\).
```
fretbursts.select_bursts.attached \((d, ich=0)\)
Select the first burst of consecutive bursts.

fretbursts.select_bursts.attached2 \((d, ich=0)\)
Select the second burst of consecutive bursts.

fretbursts.select_bursts.for_bt_fit \((d, ich=0, BT=None)\)
Select bursts for more accurate BT fitting (select before BG corr.)

fretbursts.select_bursts.fret_value \((d, ich=0, E=0.5, P_{th}=0.01)\)
Select bursts with prob. > \(P_{th}\) to have FRET equal to \(E\).

fretbursts.select_bursts.na \((d, ich=0, th1=20, th2=\infty)\)
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 \((d, ich=0, th1=20, th2=\infty)\)
Select bursts with \((naa >= th1)\) and \((naa <= th2)\).

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 \((d, ich=0, th1=20, th2=\infty)\)
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 \((d, ich=0, q=50, low=False, gamma=1.0, gamma1=None, add_naa=False)\)
Select bursts with SIZE >= q-percentile (or <= if low is True)
\(gamma\), \(gamma1\) and \(add_naa\) are passed to \fretbursts.burstlib.Data.burst_sizes_ich()\) to compute the burst size.

fretbursts.select_bursts.nearby \((d, ich=0, ms=0.2, clk_p=1.25e-08)\)
Select the first burst of bursts disting less than “ms” millisecond.

fretbursts.select_bursts.nearby2 \((d, ich=0, ms=0.2, clk_p=1.25e-08)\)
Select the second burst of bursts disting less than “ms” millisecond.

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.period \((d, ich=0, bp1=0, bp2=None)\)
Select bursts from period \(bp1\) to period \(bp2\) (included).

fretbursts.select_bursts.sbr \((d, ich=0, th1=0, th2=\infty)\)
Select bursts with SBR between \(th1\) and \(th2\).
fretbursts.select_bursts.single(d, ich=0, th=1)
  Select bursts that are at least th millisecond apart from the others.

fretbursts.select_bursts.size(d, ich=0, th1=20, th2=inf, gamma=1.0, gamma1=None, add_naa=False)
  Select bursts with burst sizes between th1 and th2.

  The parameters gamma, gamma1 and add_naa are passed to fretbursts.burstlib.Data.burst_sizes_ich() to compute the burst size.

fretbursts.select_bursts.size_noise(d, ich=0, th=2)
  Select bursts w/ size th times above the noise on both D and A ch.

fretbursts.select_bursts.size_noise_or(d, ich=0, th=2)
  Select bursts w/ size th times above the noise on D or A ch.

fretbursts.select_bursts.str_G(gamma, gamma1)
  Return a string to indicate if and how gamma or gamma1 were used.

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.topNnda(d, ich=0, N=500, gamma=1.0, gamma1=None, add_naa=False)
  Select the N biggest bursts in the channel.

  gamma, gamma1 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=1)
  Select bursts with width between th1 and th2 (ms).

1.2.4 Background estimation

background.py

Routines to compute the background from an array of timestamps. This module is imported as bg by burstlib.py.

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)
  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

**Returns** Estimated background rate in cps.

See also:

exp_cdf_fit(), exp_hist_fit()

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

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

**Returns** Estimated background rate in cps.

See also:

exp_fit(), exp_hist_fit()

```python
fretbursts.background.exp_hist_fit(ph, tail_min_us, binw=5e-05, clk_p=1.25e-08, weights='hist_counts')
```

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.

**Returns** Estimated background rate in cps.

See also:

exp_fit(), exp_cdf_fit()

### 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)

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 exponentially-distributed samples
- $s_{\text{min}}$ (float) – all samples $< s_{\text{min}}$ are discarded ($s_{\text{min}}$ must be $\geq 0$).

**Returns** The lambda parameter (1/life-time) of the exponential.

fretbursts.fit.exp_fitting.expon_fit_cdf(s, s_min=0, offset=0.5)

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 exponentially-distributed samples
- $s_{\text{min}}$ (float) – all samples $< s_{\text{min}}$ are discarded ($s_{\text{min}}$ must be $\geq 0$).

**Returns** The lambda parameter (1/life-time) of the exponential.

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

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

**Parameters**

- $s$ (array) – array of exponentially-distributed samples
- $\text{bins}$ (float or array) – if float is the bin width, otherwise is the array of bin edges (passed to numpy.histogram)
- $s_{\text{min}}$ (float) – all samples $< s_{\text{min}}$ are discarded ($s_{\text{min}}$ must be $\geq 0$).
- $\text{weights}$ (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.

**Returns** The lambda parameter (1/life-time) of the exponential.

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
- $\text{offset}$ (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
- $\text{tau}_{\text{fit}}$ (float) – mean waiting-time of the exponential distribution to use as reference
- $\text{offset}$ (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 $\tau_{\text{fit}}$.

1.2.5 Low-level burst search functions

The module `burstsearch.burstsearchlib` provides the low-level (or core) burst search and photon counting functions.

All the burst search functions return a 2-D array (burst array) of shape $N \times 6$, where $N$ is the number of bursts. The 6 columns contain the burst data.

In order to select one or more bursts, the burst array can be indexed or sliced along the first dimension (row wise or axis=0). The second dimension (columns) of the burst-array should be considered opaque, and the burst-data functions (starting with `b_`) should be used to access it (instead of the indexing the column). Using the `b_`* functions is both clearer and less bug-prone than using a column index.

The list of burst-data functions is: `b_start()`, `b_end()`, `b_size()`, `b_width()`, `b_istart()`, `b_iend()`, `b_rate()`, `b_separation()`. You can note that they are more than 6 because some of them return additional quantities derived from the burst array columns.

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

```python
mburst10 = mburst[:10]  # new array with burst data of the first 10 bursts
```

To obtain the burst start of all the bursts:

```python
b_start(mbursts)
```

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

```python
b_size(mbursts[10:20])
```

```python
fretbursts.burstsearch.burstsearchlib.b_end(b)  # Time of last ph in burst
fretbursts.burstsearch.burstsearchlib.b_iend(b)  # Index of last ph in burst
fretbursts.burstsearch.burstsearchlib.b_istart(b)  # Index of 1st ph in burst
fretbursts.burstsearch.burstsearchlib.b_rate(b)  # Mean rate of ph in burst
fretbursts.burstsearch.burstsearchlib.b_separation(b)  # Separation between nearby bursts
fretbursts.burstsearch.burstsearchlib.b_size(b)  # Number of ph in the burst
fretbursts.burstsearch.burstsearchlib.b_start(b)  # Time of 1st ph in burst
fretbursts.burstsearch.burstsearchlib.b_width(b)  # Burst width in clk cycles
fretbursts.burstsearch.burstsearchlib.bsearch(t, L, m, T, label='Burst search', verbose=True)
```

Sliding window burst search. Pure python implementation.
Finds bursts in the array $t$ (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**

- $t$ (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
- `label` (string) – a label printed when the function is called
- `verbose` (bool) – if False, the function does not print anything.

**Returns** 2D array of burst data, one row per burst, shape (N, 6), type int64. Each row contains (in the order): time of burst start, burst duration, number of photons, index of start time, time of burst end. To extract burst information it’s safer to use the utility functions $b_*$ (i.e. $b\_start()$, $b\_size()$, $b\_width()$, etc...).

```python
generate_bursts(t, L, m, T, label='Burst search', verbose=True)
```

Sliding window burst search. Pure python implementation.

Finds bursts in the array $t$ (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**

- $t$ (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
- `label` (string) – a label printed when the function is called
- `verbose` (bool) – if False, the function does not print anything.

**Returns** 2D array of burst data, one row per burst, shape (N, 6), type int64. Each row contains (in the order): time of burst start, burst duration, number of photons, index of start time, time of burst end. To extract burst information it’s safer to use the utility functions $b_*$ (i.e. $b\_start()$, $b\_size()$, $b\_width()$, etc...).

```python
generate_bursts(t, L, m, T, label='Burst search', verbose=True)
```

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.

The format of both input and output arrays is “burst-array” as returned by $b\_search()$.

**Parameters**

- `mburst_d` (array) – burst-array 1
- `mburst_a` (array) – burst array 2. The number of burst in each of the input array can be different.
**Returns** Burst-array representing the intersection (AND) of overlapping bursts.

```python
def count_ph_in_bursts_py(bursts, mask):
    # Counts number of photons in each burst counting only photons in mask.
    # This function takes a burst-array 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.

    # This is a reference implementation. In practice the multi-channel is always used instead (see mch_count_ph_in_bursts_py()).
```

**Parameters**

- `Mburst (list of 2D arrays, int64)` – a list of burst-arrays, 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 arrays, each containing the number of photons in the photon selection mask.

```python
def mch_count_ph_in_bursts(Mburst, Mask):
    # Counts number of photons in each burst counting only photons in Mask.
    # This multi-channel function takes a list of burst-arrays and a list of photon masks and compute the number of photons selected by the mask. It is used, for example, to count donor and acceptor photons in each burst.

    # Parameters
    #
    # - `Mburst (list of 2D arrays, int64)` – a list of burst-arrays, 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 arrays, each containing the number of photons in the photon selection mask.
```

**1.2.6 Fit framework**

This page contains only a general description of FRETBursts fitting functionalities. The content of this page is:
For the reference documentation for fitting multi-channel histograms see:

**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
- MultiFitter reference documentation
  - The MultiFitter class
  - Model factory functions

#### The MultiFitter class

```python
class fretbursts.mfit.MultiFitter(data_list)
```

A class for histogram, fit, KDE multiple datasets contained in a list.

Starting from the datasets in the `data_list` this class allows to conveniently compute histogram and KDE.

The histograms can be then fitted with an model (**lmfit.Model**). From the KDEs we can fit the peak position in a range.

Optionally weights can be assigned to each element in a dataset. To assign weights assigning 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.

```python
calc_kde(bandwidth=0.03)
```

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

```python
find_kde_max(x_kde, xmin=None, xmax=None)
```

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

Results are saved in the list `.kde_max_pos`.

```python
fit_histogram(model=None, pdf=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_kwvars** (**dict**) – keyword arguments passed to `model().fit`.

```python
histogram(bin_width=0.03, bins=None, verbose=False, **kwvars)
```

Compute the histogram of the data for each channel.

If `bins` is `None`, `bin_width` determines the bins array (saved in `self.hist_bins`). If `bins` is not `None`, `bin_width` is ignored and `self.hist_bin_width` is computed from `self.hist_bins`.
All the kwargs (except for bin_width) are passed to numpy.histogram.

```python
def 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)` – keywork arguments to be passed to `weight_func`.

### Model factory functions

```python
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.

```python
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.

```python
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.
```

**Parameters**

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

The other arguments are initial values for the model parameters.

**Returns**

An lmfit.Model object with all the parameters already initialized.

```python
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.

```python
fretbursts.mfit.factory_three_gaussians(p1_center=0.0, p2_center=0.5, p3_center=1, sigma=0.05)
    Return a 2-Gaussian + (optional) bridge model that can fit data.
```

**Parameters**

- `add_bridge (bool)` – if True adds a bridge function between the two gaussian peaks. If False the model has only two Gaussians.
The other arguments are initial values for the model parameters.

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

For some legacy modules used for gaussian and exponential fitting see:

### 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.

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

Returns val clipped inside the interval bounds.

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

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

```python
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.

```python
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.
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). kwars 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.

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

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). kwars 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.

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.

Return the normal pdf evaluated at x.

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

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

Mixture of two Gaussians with no area constrain.

PDF for the distribution of a mixture of two Gaussians.

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

Fit the sample s with two gaussians using Expectation Maximization.

This version 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 weights, same size as s (for ex. 1/sigma^2 ~ nt).

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

FretBursts Documentation, Release 0.4.dev

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 weights, 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 weights, same size as s (for ex. 1/sigma^2 ~ nt).

dep (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_a=[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)
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)

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: \( \mu_1, \sigma_1, a_1, \mu_2, \sigma_2, a_2 \).

**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**

```python
def expon_fit(s, s_min=0, offset=0.5):
    # 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 exponentially-distributed samples
    • **s_min** *(float)* – all samples < s_min are discarded (s_min must be >= 0).
    
    Returns The lambda parameter (1/life-time) of the exponential.
```

```python
def expon_fit_cdf(s, s_min=0, offset=0.5):
    # 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 exponentially-distributed samples
    • **s_min** *(float)* – all samples < s_min are discarded (s_min must be >= 0).
    
    Returns The lambda parameter (1/life-time) of the exponential.
```

```python
def expon_fit_hist(s, bins, s_min=0, weights=None, offset=0.5):
    # Fit of an exponential model to the histogram of s using least squares.
    
    Parameters
    • **s** *(array)* – array of exponentially-distributed samples
    • **bins** *(float or array)* – if float is the bin width, otherwise is the array of bin edges (passed to `numpy.histogram()`)
```

34 Chapter 1. Table of contents
• **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.

**Returns** The lambda parameter (1/life-time) of the exponential.

```
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.

**Overview**

FRETBursts uses of the powerful lmfit library for most fittings (like E or S histogram fitting). Lmfit is installed when running the FRETBursts Installation notebook (see Installation).

FRETBursts requires lmfit version 0.8rc3 or higher.

**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. While sensible defaults are applied 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).

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 haveing 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.

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.

1.2.7 Legacy Fit functions

These are legacy functions used in versions of FRETBursts < 0.4. This function are retained for backward compatibility.

A set of generic fit function is provided in fretbursts/fit. 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.

- `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_kwarg={}, 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.

- `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, mu0=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\texttt{.two_gauss_mix_ab}(x, p)
Mixture of two Gaussians with no area constrain.

fretbursts.fit.gaussian_fitting\texttt{.two_gauss_mix_pdf}(x, p)
PDF for the distribution of a mixture of two Gaussians.

fretbursts.fit.gaussian_fitting\texttt{.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\texttt{.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 version allows to optionally fix mean or std. dev. of each component.

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

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

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

Returns Array of parameters for the 2-gaussians (5 elements)
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 weights, same size as \( s \) (for ex. \( 1/sigma^2 \sim nt \)).
- \( debug \) (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)

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)

Fit a 2-gaussian mixture to the \((x, y)\) curve. \( \text{kargs} \) 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 Documentation, Release 0.4.dev**

### fretbursts.fit.gaussian_fitting\_two\_gaussian\_fit_hist(s, \(b\)ins=\([-0.5, -0.499, -0.498, \ldots, 1.497, 1.498, 1.499]\), \(w\)eights=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, \(b\)ounds=None, \(m\)ethod='L-BFGS-B', \(b\)ins=\([-0.5, -0.499, -0.498, \ldots, 1.497, 1.498, 1.499]\), \(w\)eights=None, \(p0=[0.2, 1, 0.8, 1, 0.3]\), \(fix\_mu=[0, 0]\), \(fix\_sig=[0, 0]\), \(fix\_a=False\), \(v\)erbose=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)

\[ \text{fretbursts.fit.gaussian_fitting.two gaussian fit hist min ab} \]

\[
\begin{align*}
&\text{s, bounds=None,} \\
&\text{method='L-BFGS-B', bins=\text{array}\{[-0.5, -0.499, -0.498, \ldots, 1.497, 1.498, 1.499]\},} \\
&\text{weights=None,} \\
&\text{p0=[0.2, 1, 0.8, 1, 0.3], fix mu=[0, 0], fix sig=[0, 0], fix a=[0, 0], verbose=False}
\end{align*}
\]

Histogram fit of sample \( s \) with 2-gaussian functions.

Uses \text{scipy.optimize.minimize} allowing constrained minimization. Also each parameter can be fixed.

The order of the parameters is: \( \mu_1, \sigma_1, a_1, \mu_2, \sigma_2, a_2 \).

Parameters

- **s** (array) – population of samples to be fitted
- **method** (string) – one of the methods accepted by \text{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 \text{L-BFGS-B, TNC} or \text{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 \text{np.histogram()}
- **weights** (array) – optional weights passed to \text{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 \text{FRET}Bursts workflow they are passed to higher level methods.

See also:

- **Background estimation**

\[ \text{fretbursts.fit.exp_fitting.expon fit} \]

\[
\text{s, s min=0, offset=0.5}
\]

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 exponentially-distributed samples
• **s_min (float)** – all samples < s_min are discarded (s_min must be >= 0).

**Returns** The lambda parameter (1/life-time) of the exponential.

\[
\text{fretbursts.fit.exp_fitting.expon_fit_cdf}(s, s_min=0, offset=0.5)
\]

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 exponentially-distributed samples
• **s_min (float)** – all samples < s_min are discarded (s_min must be >= 0).

**Returns** The lambda parameter (1/life-time) of the exponential.

\[
\text{fretbursts.fit.exp_fitting.expon_fit_hist}(s, bins, s_min=0, weights=None, offset=0.5)
\]

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

**Parameters**

• **s (array of floats)** – sample
• **bins (float or array)** – if float is the bin width, otherwise is the array of bin edges (passed to \texttt{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.

**Returns** The lambda parameter (1/life-time) of the exponential.

\[
\text{fretbursts.fit.exp_fitting.get_ecdf}(s, offset=0.5)
\]

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

\[
\text{fretbursts.fit.exp_fitting.get_residuals}(s, tau_fit, offset=0.5)
\]

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 \texttt{get_ecdf()} for details.

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

The module `burtlib_ext.py` (usually imported as `bext`) contains extensions to `burstslib.py`. It can be thought 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.

- `bursts_fitter()` and `fit_bursts_kde_peak()` help to build and fit histograms and KDEs for E or S.
- `burst_search_and_gate()` performs the AND-gate burst search taking intersection of the bursts detected in two photons streams.
- `calc_mdelays_hist()` computes the histogram of the m-delays distribution of photon intervals.
- `burst_data_period_mean()` computes a mean of any “burst data” for each background period.
- `join_data()` joins different measurements to create a single “virtual” measurement from a series of measurements.

Finally a few functions deal with burst timestamps:

- `get_burst_photons()` returns a list of timestamps for each burst.
- `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.

```python
fretbursts.burstlib_ext.asymmetry(dx, ich=0, func=<function mean at 0x7f1a6cf6e938>, drop-nan=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).

```python
fretbursts.burstlib_ext.burst_data(dx, ich=0, include_bg=False, include_ph_index=False)
```

Return a pandas Dataframe (one row per bursts) with all the burst data.

```python
fretbursts.burstlib_ext.burst_data_period_mean(dx, burst_data)
```

Compute mean `bursts_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)

```python
def fretbursts.burstlib_ext.burst_search_and_gate(dx, F=6, m=10, ph_sel1=Ph_sel(Dex='DAem', Aex=None), ph_sel2=Ph_sel(Dex=None, Aex='Aem'), 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) -- Burst search parameter F.
    # • m (int) -- Burst search parameter m.
    # • 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_t().
```

```python
def fretbursts.burstlib_ext.bursts_fitter(dx, burst_data='E', save_fitter=True, weights=None, gamma=1, add_naa=False):
    # 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.
    # Returns The mfit.MultiFitter object with the specified burst-size weights.
```

```python
def fretbursts.burstlib_ext.calc_bg_brute(dx, min_ph_delay_list=None, return_all=False):
    # 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.

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.

```python
fretbursts.burstlib_ext.calc_mdelays_hist(d, ich=0, m=10, bp=(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.

- **bp** (*int or 2-element tuple*) – index of the period to use. If tuple, the period range between `bp[0]` and `bp[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.

```python
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.

```python
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.

**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.Ph_sel` for details.

**Returns** A list of arrays of photon timestamps (one array per burst).

```python
fretbursts.burstlib_ext.histogram_mdelays(d, ich=0, m=10, period=None, ph_sel=Ph_sel(Dex='DAem', Aex='DAem'), bin_width=0.001, dt_max=0.01, bins=None, bursts=False, pdf=True)
```

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)* – index of the period to use.

• **ph_sel** *(Ph_sel object)* – photon selection to use.

**Returns** Two arrays for the histogram counts and the bins centers. If **bursts** is True the counts arrays contains a second row that is the histogram for only the photons inside bursts.

```python
fretbursts.burstlib_ext.join_data(d_list, gap=1)
```

Joins burst data of different measurements in a single Data object.

This function requires that all the passed data objects use the same period (bg_time_s). For each measurement, the time of burst start is offset by the duration of the previous measurement + an additional **gap**

The index of the first/last photon in the burst (returned by `b_istart()` and `b_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 error when calling functions that require the timestamps data.

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.

Burst widths and sizes are kept unchanged.

A new attribute **(i_origin)**, containing for each burst the index of the original data object in the list, is saved in the returned object.
**Returns** A `Data` object containing bursts from the all the objects in `d_list`.

```python
def ph_burst_stats(d, ich=0, func=<function mean at 0x7f1a6cf6e938>, 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`.

**Returns** An array containing per-burst timestamp statistics.

### 1.2.9 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.

**What is HDF5?**

HDF5 is standard and general-purposes container-format for binary data (see also [HDF on Wikipedia](https://en.wikipedia.org/wiki/HDF5)). 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.

**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`. 
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.

HDF5 in FRETBursts

FRETBursts allows saving and loading smFRET data from and to an HDF5-based file format called **HDF5-Ph-Data**.

The **HDF5-Ph-Data** is basically a pre-defined layout to be used with smFRET and other data involving time-series of photon-data.

A description of the HDF5-Ph-Data format and its specifications can be found in **HDF5-Ph-Data format 0.2 Draft**.

For documentation on using the HDF5-Ph-Data format in **FRETBursts** see:

HDF5-based smFRET file format

We developed an HDF5-based format called **HDF5-Ph-Data** for smFRET and other measurements involving series of photon timestamps. The specifications of the HDF5-Ph-Data format can be found in **HDF5-Ph-Data format 0.2 Draft**.

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.hdf5()**.

Any measurements data loaded in a **burstlib.Data** object can be saved in HDF5-Ph-Data format by using the **hdf5.store()** function.

FRETBursts hdf5 module

The module **fretbursts.hdf5** provides the **hdf5.store()** function and other utility functions to quickly print structure (hierarchy) and attributes (metadata) of HDF5 files. This module contains a function to store **fretbursts.burstlib.Data** objects to disk in **HDF5-smFRET** format.

Utility functions to print the HDF5 file structure and data-attributes are also provided.

**fretbursts.hdf5.print attrs** *(data_file, node_name=’/’, which=’user’)*

Print the HDF5 attributes for node_name.

**Parameters**

- **data_file** *(pytables HDF5 file object)* – the data file to print
• **node_name** (*string*) – name of the path inside the file to be printed. Can be either a group or a leaf-node. Default: ‘/’, the root node.

• **which** (*string*) – Valid values are ‘user’ for user-defined attributes, ‘sys’ for pytables-specific attributes and ‘all’ to print both groups of attributes. Default ‘user’.

```python
fretbursts.hdf5.print_children(data_file, group='/')
```

Print all the sub-groups in `group` and leaf-nodes children of `group`.

**Parameters**

• **data_file** (*pytables HDF5 file object*) – the data file to print

• **group** (*string*) – path name of the group to be printed. Default: ‘/’, the root node.

```python
fretbursts.hdf5.store(d, compression={'complevel': 6, 'complib': 'zlib'}, h5_fname=None)
```

Saves the `Data` object `d` in the HDF5-Ph-Data format. As a side effect the `d` object is modified by adding the attribute `data_file` that contains a reference to the pytables file.

**Parameters**

• **d** (*Data object*) – the Data object containing the smFRET measurement.

• **compression** (*dict*) – a dictionary containing the compression type and level. Passed to `pytables.tables.Filters()`.

• **h5_fname** (*string or None*) – if not None, contains the file name to be used for the HDF5 file. If None, the file name is generated from `d.fname`, by replacing the original extension with ‘.hdf5’.

For description and specs of the HDF5-Ph-Data format see: [https://github.com/tritemio/FRETBursts/wiki/HDF5-Ph-Data-format-0.2-Draft](https://github.com/tritemio/FRETBursts/wiki/HDF5-Ph-Data-format-0.2-Draft)

### 1.2.10 FRET Correction Formulas

The `fretmath` module contains functions to compute corrected FRET efficiency from the proximity ratio and vice-versa.

For derivation see notebook: “Algebra of FRET corrections.ipynb” ([link](https://github.com/tritemio/FRETBursts/wiki/HDF5-Ph-Data-format-0.2-Draft)).

```python
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 efficiency
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.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)

1.2.11 Description of the files

Here a brief list of the main FRETBursts files.

load_fretbursts.py

This is a small script in the notebook folder that is run in an IPython Notebook to import FRETBursts (run load_fretbursts).

The script performs 3 basic operations:
• Find the FRETBursts source folder (either from an environment variable or from a default value set in the script) and set two variables (NOTEBOOK_DIR and FRETBURSTS_DIR) to easily switch between the notebooks folder and the FRETBursts source folder.

• Load FRETBursts.
  • If git is found, it displays the current FRETBursts revision (and eventual files modified since last revision)

To quickly switch between the notebooks dir and the FRETBursts source dir, use:

%cd $NOTEBOOK_DIR

or:

%cd $FRETBURSTS_DIR

burstlib.py

This is the main library to import (or run) in order to load all the analysis functions.
burstlib.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”.

loader.py

This 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 file 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.

burst_plot.py

This module defines all the plotting functions for the 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)
background.py

Routines to compute the background from an array of timestamps. This module is imported as bg by burstlib.py. 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.

select_bursts.py

Functions to select bursts according to different criteria.

These functions are usually passed to the burstlib.Sel() as a second parameter. For example:

ds = Sel(d, 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).

burstsearch (folder)

This folder contains the core burst search functions. Also core functions to obtain basic burst information are defined here.

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.

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.

fit (folder)

This folder contains generic fit functions for Gaussian and exponential fit of a sample.

See Fit framework.

1.3 Compiling Cython code

Cython is a tool that, among other things, allows to translate 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 python 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.
To compile the cython functions into machine code, enter the folder `burstsearch` and type:

```
python setup.py build_ext --inplace
```

This command requires that both cython and a standard compiler are installed.

### 1.3.1 Installing a compiler (optional)

**Linux**

On **Linux** the preferred compiler is GCC, that is easily available for any distribution.

**Windows**

On **Windows**, the MS Visual Studio v7 compiler is the preferred compiler. In principle, installing only MS Windows SDK v7.0 (GRMSDKX_EN_DVD.iso) should be sufficient. However, sometime it maybe necessary to install the full MS Visual Studio Express 2008 (VS2008ExpressWithSP1ENUX1504728.iso). Both downloads are free.

After installation and a reboot, the cython version installed by python distributions like Anaconda should automatically find the compiler.

**Mac OSX**

On **Mac OSX** the LLVM compiler included in Xcode should be installed (untested).
Indices and tables

• genindex
• modindex
• search
fretbursts, 48
fretbursts.background, 52
fretbursts.burst_plot, 51
fretbursts.burstlib, 51
fretbursts.burstlib_ext, 43
fretbursts.burstsearch, 52
fretbursts.burstsearch.burstsearchlib, 24
fretbursts.dataload, 52
fretbursts.fit, 52
fretbursts.fit.exp_fitting, 41
fretbursts.fit.gaussian_fitting, 36
fretbursts.fretmath, 49
fretbursts.hdf5, 48
fretbursts.loader, 7
fretbursts.mfit, 27
fretbursts.ph_sel, 18
fretbursts.select_bursts, 52