pyFRET Documentation

Release 1.0

Rebecca R. Murphy

May 14, 2015

Contents

Contents:

CHAPTER 1

pyFRET Tutorial

1.1 Installing pyFRET

pyFRET is available as a module on PyPI, the Python Packagae Index.

If you are already familiar with python and PyPI:

- make sure you have numpy, scipy, matplotlib and scikit learn installed
- pip install pyFRET

If you are completely new to python, there are three stages to getting pyFRET up and running:

1. Getting Python
2. Getting Anaconda (extra packages for scientific computing)
3. Getting scikit learn
4. Getting pyFRET

Instructions for all three steps follow:

1.1.1 Getting Python

If you are not already using python for programming, you may need to install python. Here are some instructions:

- **For Unix.**
- **For iOS.**
- **For Windows.**

These pages also provide useful links to tutorials for programming in python.

Please note: pyFRET was written using python 2. The latest release of python 2 is python 2.7.6. The pyFRET library is also compatible with python 3, for which the latest release is python 3.4.

1.1.2 Getting Anaconda

Once you have python up and running, you need to make sure you have all the packages that pyFRET needs to work properly. Specifically, you will need:

- scipy
- numpy

- matplotlib
- scikit learn

If you have used python for scientific programming before, you may already have these installed. If so, you are ready to install pyFRET.

If you haven't used python much, you will need to get these packages. The easiest way is to download Anaconda, which will install 125 python packages used for scientific programming.

Some hints for installing Anaconda:

- Their instructions are very clear and easy to follow
- During the installation, you will be asked if you want to add the anaconda binary directory to your PATH environment variable. **Say yes**. This means that every time you use python, you will have acces to all the packages installed by Anaconda.

1.1.3 Getting Scikit learn

Scikit learn is not installed by default with Anaconda. However, once you have Anaconda installed, it is easy to install scikit learn. From the Anaconda terminal, type:

```
$ conda install scikit-learn
```

If you are not using Anaconda and you need to install scikit learn, there are comprehensive instructions on the scikit-learn website.

1.1.4 Getting pyFRET

Now you are ready to install pyFRET.

Open a terminal window and type:

```
$ pip install pyFRET
```

Installing pyFRET using pip will automatically detect whether you have the required packages and will install them for you at the same time as pyFRET is installed. Sadly, scipy and matplotlib don't install very nicely using pip install, so this will make a big mess.

If you need to install these packages, get them before you try downloading pyFRET. Scipy has dependencies on numpy, so you will need to install numpy first.

The best instructions for getting all of the required packages can be found on the Scipy installation page.

1.2 Using pyFRET

To use pyFRET to analyse your data, you must first import the module into your python program. You can use:

```
import pyfret
```

This will import the whole module. However, it is easier to import pyFRET and pyALEX separately:

```
from pyfret import pyFRET as pft
from pyfret import pyALEX as pyx
```

This will let you use their functions directly.

Sample code that uses pyFRET to analyse smFRET data can be found in ALEX_example.py and FRET_example.py. These programs use configuration files to load parameters for the analysis. These configuration files are ALEX_config.cfg and FRET_config.cfg. These files can be found in the /bin folder of the pyFRET download.

To provide further illustration of how pyFRET can be used, below are some examples of things that you can do using pyFRET.

1.3 Using pyFRET.pyFRET

Now that you have pyFRET imported into your program, you are ready to use it to analyse data. Let's start with a simple analysis of some FRET data.

First, you need to initialize a FRET data object to hold your data.

If you have a list of .csv files (here called file1, file2 and file3), you can do this:

```
my_directory = "path/to/my/files"
list_of_files = ["file1.csv", "file2.csv", "file3.csv"]
my_data = pft.parse_csv(my_directory, list_of_files)
```

This will store your data as two arrays of values, named donor and acceptor, in an object called my data. You can print these arrays like this:

```
print my_data.donor
print my_data.acceptor
```

Similarly, if you have a list of binary files with a .dat extension, like those used in the Klenerman group, you can do this:

```
my_directory = "path/to/my/files"
list_of_files = ["file1.dat", "file2.dat", "file3.dat"]
my_data = pft.parse_bin(my_directory, list_of_files)
```

A handy hint about files: A dataset typically consists of many files in the same directory, with the same base name but different file numbers. To quickly build a list of files to be parsed by one of these function, you can do something like this:

```
# The info you need
my_directory = "path/to/my/files"
no_files = 20       # how many files you have
files = []          # empty list to hold file names
name = "mydata"     # main part of file name
filetype = "dat"    # file extension

# Making your list of files
for n in range(no_files):
    # for n = 1, full_name = mydata0001.dat
    full_name = ".".join(["%s%04d" %(name, i), filetype])
    files.append(full_name)

# Reading your data
FRET_data = pft.parse_bin(my_directory, files)
```

Now you are ready to start manipulating the data.

To subtract background autofluorescence:

```
auto_donor = 0.5 # donor autofluorescence
auto_acceptor = 0.3 # acceptor autofluorescence
my_data.subtract_bckd(auto_donor, auto_acceptor)
```

To select bursts using AND thresholding:

```
threshold_donor = 20 # donor threshold
threshold_acceptor = 20 # acceptor threshold
my_data.threshold_AND(threshold_donor, threshold_acceptor)
```

To select bursts using SUM thresholding:

```
threshold = 30 # threshold
my_data.threshold_SUM(threshold)
```

To remove cross-talk from bursts:

```
cross_DtoA = 0.05 # fractional crosstalk from donor to acceptor
cross_AtoD = 0.01 # fractional crosstalk from acceptor to donor
my_data.subtract_crosstalk(cross_DtoA, cross_AtoD)
```

To calculate the FRET proximity ratio of bursts, you can use the proximity_ratio function:

```
gamma = 0.95 # instrumental gamma factor (default value 1.0)
E = my_data.proximity_ratio(gamma=0.95)
```

You can also build FRET histogram directly from the donor and acceptor data. To make a FRET histogram, use the function build_histogram, which will calculate the proximity ratio internally. This function has several optional additions.

The simplest option is just to make a histogram, and save the frequencies and bin centres in a .csv file:

```
filepath = "path/to/save/histogram"
csvname = my_histogram
g_factor = 0.95 # instrumental gamma factor
my_data.build_histogram(filepath, csvname, gamma=g_factor, bin_min=0.0, bin_max=1.0, bin_width=0.02)
```

You can also save an image of the histogram:

```
filepath = "path/to/save/histogram"
csvname = my_histogram
g_factor = 0.95 # instrumental gamma factor
my_data.build_histogram(filepath, csvname, gamma=g_factor, bin_min=0.0, bin_max=1.0, bin_width=0.02,
image = True, imgname = my_histogram, imgtype="png")
```

Finally, you can fit the histogram with one or more gaussian distributions and save the parameters of the fit in a csv file. This will also make an image of the histogram overlaid with the gauss fit.

```
filepath = "path/to/save/histogram"
csvname = my_histogram
g_factor = 0.95 # instrumental gamma factor
n_gauss = 2     # number of gaussians to fit
my_data.build_histogram(filepath, csvname, gamma=g_factor, bin_min=0.0, bin_max=1.0, bin_width=0.02,
image=True, imgname=my_histogram, imgtype="png", gauss=True, gaussname="gaussfit", n_gauss=n_gauss)
```

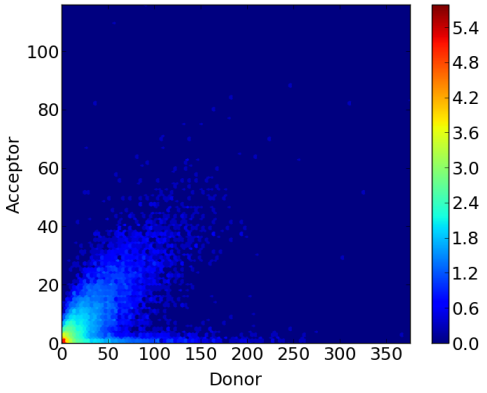
As well as making histograms, you can also make some other plots to display your data. For example, a heatmap or 3D plot of event frequencies:

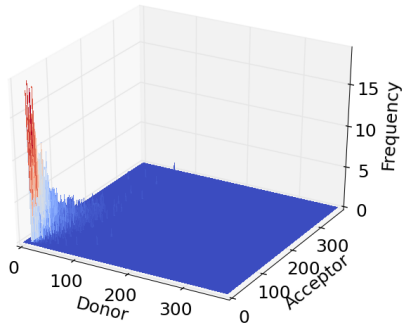
```
# make a heatmap
filepath = "path/to/save/image"
plotname = my_plot
```

```
my_data.make_hex_plot(filepath, plotname, imgtype="pdf", binning="log")

# make a 3D plot
filepath = "path/to/save/image"
plotname = my_3d_plot
my_data.make_3d_plot(filepath, plotname, imgtype="pdf")
```

This makes images like these:





For more information on the pyFRET library, more functions and more detail, please see the reference:

1.4 Using pyFRET.pyALEX

pyALEX is for data collected using alternating laser excitation. Many of the functions are similar to those used in pyFRET. Here is a quick oveview.

First, you need to initialize an ALEX data object to hold your data.

If you have a list of .csv files (here called file1, file2 and file3), you can do this:

```
my_directory = "path/to/my/files"
list_of_files = ["file1.csv", "file2.csv", "file3.csv"]
my_data = pft.parse_csv(my_directory, list_of_files)
```

This will store your data as four arrays of values (see below) in an object called my_data. The four data channels in an ALEX experiment are:

- D_D: Donor channel when the donor laser is on

- D_A: Donor channel when the acceptor laser is on
- A_D: Acceptor channel when the donor laser is on
- A_A: Acceptor channel when the acceptor laser is on

You can print these arrays like this:

```
print my_data.D_D
print my_data.D_A
print my_data.A_D
print my_data.A_A
```

Similarly, if you have a list of binary files with a .dat extension, like those used in the Klenerman group, you can do this:

```
my_directory = "path/to/my/files"
list_of_files = ["file1.dat", "file2.dat", "file3.dat"]
my_data = pft.parse_bin(my_directory, list_of_files)
```

A handy hint about files: A dataset typically consists of many files in the same directory, with the same base name but different file numbers. To quickly build a list of files to be parsed by one of these function, you can do something like this:

```
# The info you need
my_directory = "path/to/my/files"
no_files = 20        # how many files you have
files = []           # empty list to hold file names
name = "mydata"      # main part of file name
filetype = "dat"     # file extension

# Making your list of files
for n in range(no_files):
    # for n = 1, full_name = mydata0001.dat
    full_name = ".".join(["%s%04d" %(name, i), filetype])
    files.append(full_name)

# Reading your data
FRET_data = pft.parse_bin(my_directory, files)
```

Now you are ready to start manipulating the data.

To subtract background autofluorescence:

```
autoD_D = 0.5 # donor autofluorescence (donor laser)
autoD_A = 0.3 # donor autofluorescence (acceptor laser)
autoA_D = 0.5 # acceptor autofluorescence (donor laser)
autoA_A = 0.8 # acceptor autofluorescence (acceptor laser)
my_data.subtract_bckd(auto_donor, auto_acceptor)
```

To subtract leakage and direct excitation from the FRET channel:

```
l = 0.05 # fractional leakage from donor excitation into acceptor channel
d = 0.03 # fractional contribution of direct acceptor excitation by donor laser
my_data.subtract_crosstalk(l, d)
```

The key innovation of ALEX is to select events based on their photon stoichiometry – the fraction of all observed photons that were emitted by the acceptor dye.

Both the Proximity Ratio, E, and the stoichiometry, S, can be calculated explicitly:

```
    g_factor = 0.95 # the gamma factor
E = my_data.proximity_ratio(gamma=g_factor) # Proximity ratio
S = my_data.stoichiometry(gamma=g_factor) # Stoichiometry
```

Then you can remove singly-labelled molecules that give events with extreme stoichiometries:

```
S = my_data.stoichiometry(gamma=g_factor)
    S_min = 0.2 # minimum S
S_max = 0.8 # maximium S
S = my_data.stoichiometry_selectiony(S, S_min, S_max)
```

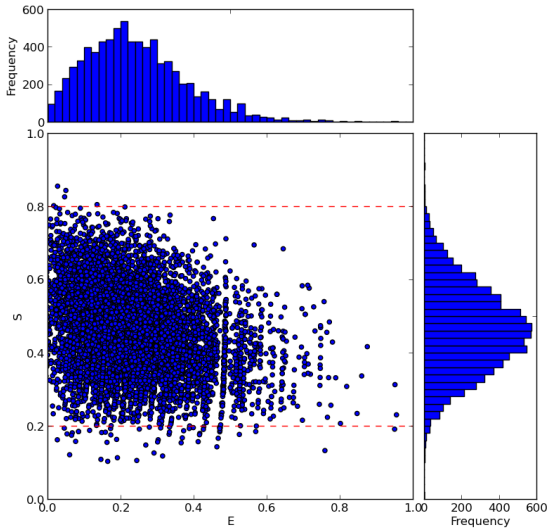
However, E and S can also be calculated in a single step, combined with burst selection based on S, using the scatter_hist function. This will also plot and optionally save a scatter plot of your data, with projections of E and S:

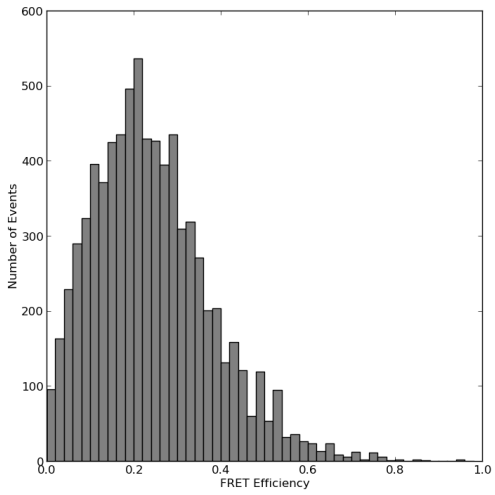
```
g_factor = 0.95 # the gamma factor
S_min = 0.2
S_max = 0.8
filepath = "path\to\my\file"
filename = "scatter_plot"
scatter_hist(self, S_min, S_max, gamma=1.0, save=True, filepath=filepath, imgname=filename, imgtype="
```

You can also make a separate histogram of the selected events. The histogram frequencies and bin centres will be saved in a csv file. You can also optionally save an image of the histogram and fit it to one or more gaussian distributions.

```
g_factor = 0.95 # the gamma factor
S_min = 0.2
S_max = 0.8
filepath = "path\to\my\file"
filename = "E_histogram"
img = "E_plot"
build_histogram(filepath, filename, gamma=1.0, bin_min=0.0, bin_max=1.0, bin_width=0.02, image = True
```

The scatterplot and FRET Efficency histograms look like this:





For more details about pyALEX, please see the detailed documentation:

1.5 Using The Burst Search Algorithms

pyFRET now includes burst search algorithms for both FRET and ALEX data.

The algorithms implemented are the All Photons Burst Search (APBS) AND Dual Channel Burst Search (DCBS) algorithms described in Nir et al.'s 2006 paper.

To use these algorithms, you will need time-binned data that has been binned on a timescale much shorter than the typical dwell time of a molecule in the confocal volume. A suitable bin-time would be 0.01 ms for freely diffusing molecules, although shorter bin times can also be used.

These burst search algorithms identify fluorescent bursts by grouping together photons that reach the photon counting devices within some short time window of each other. According to Nir et al:

> The start (respectively, the end) of a potential burst is detected when the number of photons in the averaging window of duration T is larger (respectively, smaller) than the minimum number of photons M.
>
> A potential burst is retained if the number of photons it contains is larger than a minimum number L.

In the APBS method, photons from all channels are summed to give a total number of photons which is then evaluated in each window. In the DCBS method, photons encountered during donor exciation periods are considered separately from those encountered during acceptor excitation periods. To be accepted as a burst, both donor and acceptor dyes must be active throughout the entire burst.

To use the burst search algorithms in pyFRET, it is necessary to first initialize a pyFRET data object. The burst search algorithm can then be used:

```
# The info you need
my_directory = "path/to/my/files"
no_files = 20        # how many files you have
files = []           # empty list to hold file names
name = "mydata"      # main part of file name
filetype = "dat"     # file extension

# Making your list of files
for n in range(no_files):
```

```
    # for n = 1, full_name = mydata0001.dat
    full_name = ".".join(["%s%04d" %(name, n), filetype])
    files.append(full_name)

# Reading your data
FRET_data = pft.parse_bin(my_directory, files)

# calling APBS algorithm
T1 = 50             # time window (bins)
L1 = 50             # first threshold
M1 = 30             # second threshold

bursts_APBS = FRET_data.APBS(T1, L1, M1)

# calling DCBS algorithm
T2 = 50             # time window (bins)
L2 = 25             # first threshold
M2 = 15             # second threshold

bursts_DCBS = FRET_data.APBS(T2, L2, M2)
```

According to Nir et al., appropriate values for the APBS algorithm are:

- T = 0.5 ms
- L = 50
- M = 30

Similarly for the DCBS algorithm:

- T = 0.5 ms
- L = 25
- M = 15

The value of T used in the pyFRET burst search algorithm depends on the bin-time used. If a 0.01 ms bin-time is used, then to achieve a window size of T = 0.5 ms, a value of T = 50 bins should be used in calling the burst search algorithm.

The burst search algorithm returns a FRET_bursts data object, which can be used in the same way as the original FRET data object. However, an additional denoise_bursts function is provided, which will denoise identified bursts in a manner proportional to the burst duration. Crosstalk subtraction is unchanged.

```
# subtract background
N_D = 0.005   # donor noise per bin
N_A = 0.004   # acceptor noise per bin
bursts_APBS.denoise_bursts(N_D, N_A)

cross_DtoA = 0.05 # fractional crosstalk from donor to acceptor
cross_AtoD = 0.01 # fractional crosstalk from acceptor to donor
bursts_APBS.subtract_crosstalk(cross_DtoA, cross_AtoD)

# plot FRET histogram
filepath = "path/to/save/histogram"
csvname = "my_histogram"
g_factor = 0.95 # instrumental gamma factor
bursts_APBS.build_histogram(filepath, csvname, gamma=g_factor, bin_min=0.0, bin_max=1.0, bin_width=0
```

The FRET_bursts data object has three extra attributes, burst_starts, burst_ends and burst_len, which are arrays of (respectively) the start time, end time and duration (in time bins) of each identified burst. The FRET_bursts object also

has an additional plotting function, that can be used to display the relationship between burst duration and brightness:

```
# the burst_len attribute
print bursts_APBS.burst_len

# plotting the relationship between burst duration and brightness
filepath = "path/to/save/plot"
imgname = "my_plot"
bursts_APBS.scatter_intensity(filepath, imgname, imgtype="pdf")
```

Similarly for ALEX data:

```
# The info you need
my_directory = "path/to/my/files"
no_files = 20       # how many files you have
files = []          # empty list to hold file names
name = "mydata"     # main part of file name
filetype = "dat"    # file extension

# Making your list of files
for n in range(no_files):
    # for n = 1, full_name = mydata0001.dat
    full_name = ".".join(["%s%04d" %(name, n), filetype])
    files.append(full_name)

# Reading your data
ALEX_data = pyx.parse_bin(my_directory, files)

# calling APBS algorithm
T1 = 50             # time window (bins)
L1 = 50             # first threshold
M1 = 30             # second threshold

bursts_APBS = ALEX_data.APBS(T1, L1, M1)

# calling DCBS algorithm
T2 = 50             # time window (bins)
L2 = 25             # first threshold
M2 = 15             # second threshold

bursts_DCBS = ALEX_data.APBS(T2, L2, M2)

# make scatterhist plot with projections
S_min = 0.2
S_max = 0.8
filepath = "path/to/save/plot"
ALEX_data.scatter_hist(S_min, S_max, gamma=1.0, save=True, filepath=filepath, imgname="scatterhist",
```

1.6 RASP: Recurrence Analysis of Single Particles

Finally, the FRET_bursts data can be analysed using the Recurrence Analysis of Single Particles (RASP) method described by Hoffmann et al. (Phys Chem Chem Phys. 2011 13(5):1857-1871). Fluorescent bursts that occur within a short time interval of each other have a high probability of having been generated by the same fluorescent molecule reentering the confocal volume. RASP can be used to identify bursts that occurred within a short time period of bursts with a specified FRET efficiency.

From Hoffmann et al,:

First, the bursts b2 must be detected during a time interval between t1 and t2 (the 'recurrence interval', T = (t1,t2)) after a previous burst b1 (the 'initial burst'). Second, the initial bursts must yield a transfer efficiency, E(b1), within a defined range, Delta E1 (the 'initial E range').

In pyFRET's implementation of RASP, t1 and t2 are named Tmin and Tmax. The initial E range is given by Emin and Emax:

```
# initial E range: 0.4 < E < 0.6
Emin = 0.4
Emax = 0.6

# Time interval for re-occurrence
# given in number of bins
Tmin = 1000
Tmax = 10000

# selcting re-ocurring bursts
recurrent_bursts = bursts_APBS.RASP(Emin, Emax, Tmin, Tmax)

# histogram of re-occurring bursts
recurrent_bursts.build_histogram(filepath, csvname, gamma=g_factor)
```

1.7 The Sample Data

Included in /bin is some sample data that you can use to check that your installation of pyFRET is working correctly.

To reproduce our data analysis, from the /bin folder in pyFRET, type:

```
$ python FRET_example.py 10bp_FRET_config.cfg
```

This will execute the program FRET_example.py using the parameters stored in the configuration file 10bp_FRET_config.cfg, to analyse smFRET data from dual labelled DNA duplex, with a 10 base-pair separation between the dye attachment sites. There are four more smFRET datasets to analyse (4, 6, 8 and 12 bp separations) with similar configuration files.

Similarly, you can reproduce our analysis of the equivalent ALEX data using:

```
$ python ALEX_example.py 10bp_ALEX_config.cfg
```

There is also some sample data and a sample script for burst search in FRET data:

```
$ python FRET_bursts_example.py FRET_bursts_config.cfg
```

Right now, the configuration file parser that is used in ALEX_example.py and FRET_example.py runs only with a python 2.x installation. We are working on making an equivalent set of files for use with python 3.x distributions.

To learn more about how configuration files work and how you can use them to analyse your own data please see the configparser documentation:

You can open both the configuration files and the example python scripts in a text editor (like Sublime or Gedit) to see how the configuration files are used by the python program:

CHAPTER 2

pyFRET Reference

class pyFRET.**FRET_bursts**(*donor*, *acceptor*, *burst_starts*, *burst_ends*)

This class holds single molecule burst data. Photon bursts are stored in numpy arrays. There is a separate array for each of the two photon streams, for the start and end of each burst and for the burst duration.

The two attributes corresponding to bursts from the four photon streams from a FRET experiment are numpy arrays:

- donor: The donor channel
- acceptor: The acceptor channel

The three further attributes, corresponding to burst duration, burst start time and burst end time are also numpy arrays:

- burst_len: Length (in bins) of each identified burst
- burst_starts: Start time (bin number) of each identified burst
- burst_ends: End time (bn number) of each identified burst

The class can be initialized directly from four lists or arrays: two of burst photon counts; the burst start times and the burst end times: bursts = FRET_bursts(donor_bursts, acceptor_bursts, burst_starts, burst_ends).

However, it is more typically achieved by running the APBS or DCBS algorithm that forms part of the FRET_data class.

RASP(*Emin*, *Emax*, *Tmin*, *Tmax*, *gamma=1.0*)

Recurrence Analysis of Single Particles analysis as implemented in Hoffmann et al. Phys Chem Chem Phys. 2011 13(5):1857-1871. Returns a FRET_bursts object.

Arguments: * Emin: minimum value of E (proximity ratio) to consider for initial bursts * Emax: maximum value of E (proximity ratio) to consider for initial bursts * Tmin: start time (in number of bins after the initial burst) to search for recurrent bursts * Tmax: end time (in number of bins after the initial burst) to search for recurrent bursts

Keyword Arguments: * gamma: value of instrumental gamma factor to use in calculating the proximity ratio. Default value = 1.0.

From Hoffmann et al.: First, the bursts b2 must be detected during a time interval between t1 and t2 (the 'recurrence interval', T = (t1,t2)) after a previous burst b1 (the 'initial burst'). Second, the initial bursts must yield a transfer efficiency, E(b1), within a defined range, Delta E1 (the 'initial E range').

In this implementation, Tmin and Tmax correspond to t1 and t2 respectively. The initial E range lies between Emin and Emax.

denoise_bursts(*N_D*, *N_A*)

Subtract background noise from donor and acceptor bursts.

Arguments:

•N_D: average noise per time-bin in the donor channel

•N_A: average noise per time-bin in the acceptor channel

scatter_intensity(*filepath, imgname, imgtype='pdf', labels=['Burst Duration', 'Burst Intensity']*)

Plot a scatter plot of burst brightness vs burst duration

Arguments: * filepath: file path to the directory in which the image will be saved * imgname: name under which the image will be saved

Keyword arguments: * imgtype: filetype of histogram image. Accepted values: jpg, tiff, rgba, png, ps, svg, eps, pdf * labels: labels for x and y axes, as a 2-element list of strings: ["x-title", "y-title"]. Default value: ["Burst Duration", "Burst Intensity"]

class pyFRET.**FRET_data**(*donor*, *acceptor*)

This class holds single molecule data.

It has two attributes, donor and acceptor to hold photon counts from the donor and acceptor channels respectively. These are numpy arrays.

It can be initialized from two lists or two arrays of photon counts: data = FRET_data(donor_events_list, acceptor_events_list)

APBS(*T*, *M*, *L*)

All-photon bust search algorithm as implemented in Nir et al. J Phys Chem B. 2006 110(44):22103-24. Returns a FRET_bursts object.

Arguments: * T: time-window (in bins) over which to sum photons * M: number of photons in window of length T required to identify a potential burst. * L: total number of photons required for an identified burst to be accepted.

From Nir et al.: The start (respectively, the end) of a potential burst is detected when the number of photons in the averaging window of duration T is larger (respectively, smaller) than the minimum number of photons M. A potential burst is retained if the number of photons it contains is larger than a minimum number L.

DCBS(*T*, *M*, *L*)

Dual-channel bust search algorithm as implemented in Nir et al. J Phys Chem B. 2006 110(44):22103-24. Returns a FRET_bursts object.

Arguments: * T: time-window (in bins) over which to sum photons * M: number of photons in window of length T required to identify a potential burst. * L: total number of photons required for an identified burst to be accepted.

From Nir et al.: The start (respectively, the end) of a potential burst is detected when the number of photons in the averaging window of duration T is larger (respectively, smaller) than the minimum number of photons M. A potential burst is retained if the number of photons it contains is larger than a minimum number L.

build_histogram(*filepath*, *csvname*, *gamma=1.0*, *bin_min=0.0*, *bin_max=1.0*, *bin_width=0.02*, *image=False*, *imgname=None*, *imgtype=None*, *gauss=True*, *gaussname=None*, *n_gauss=1*)

Build a proximity ratio histogram and save the frequencies and bin centres as a csv file. Optionally plot and save a graph and perform a simple gaussian fit.

Arguments:

•E: array of FRET efficiecies

•filepath: path to folder where the histogram will be saved (as a string)

•csvname: the name of the file in which the histogram will be saved (as a string)

Keyword arguments:

•gamma: Instrumental gamma factor. (float, default value 1.0)

•bin_min: the minimum value for a histogram bin (default 0.0)

•bin_max: the maximum value for a histogram bin (default 1.0)

•bin_width: the width of one bin (default 0.02)

•image: Boolean. True plots a graph of the histogram and saves it (default False)

•imgname: the name of the file in which the histogram graph will be saved (as a string)

•imgtype: filetype of histogram image. Accepted values: jpg, tiff, rgba, png, ps, svg, eps, pdf

•gauss: Boolean. True will fit the histogram with a single gaussian distribution (default False)

•gaussname: the name of the file in which the parameters of the Gaussian fit will be saved

•n_gauss: number of Gaussain distributions to fit. Default = 1

make_3d_plot (*filepath, imgname, imgtype='pdf', labels=['Donor', 'Acceptor', 'Frequency']*)

Make a 3D histogram of donor and acceptor photon counts.

Arguments:

•filepath: path to folder where data will be saved

•filename: name of image file to save plot

Keyword arguments:

•filetype: image type (as string). Default "pdf". Accepted values: jpg, tiff, rgba, png, ps, svg, eps, pdf

•labels: axes labels, list of strings ["Xtitle", "Ytitle", "Ztitle"]. Default ["Donor", "Acceptor", "Frequency"]

make_hex_plot (*filepath, imgname, imgtype='pdf', labels=['Donor', 'Acceptor'], xmax=None, ymax=None, binning=None*)

Make a 2D representation of donor and acceptor photon count frequencies.

Based on the matplotlib.pyplot construction "hexbin": http://matplotlib.org/api/pyplot_api.html

Arguments:

•filepath: path to folder where data will be saved

•imgname: name of image file to save plot

Keyword arguments:

•imgtype: image type (as string). Default "pdf". Accepted values: jpg, tiff, rgba, png, ps, svg, eps, pdf

•labels: axes labels, list of strings ["Xtitle", "Ytitle"]. Default ["Donor", "Acceptor"]

•xmax: maximum x-axis value. Default None (maximum will be the brightest donor event)

•ymax: maximum x-axis value. Default None (maximum will be the brightest acceptor event)

•**binning: type of binning to use for plot. Default: None (bin colour corresponds to frequency).** Accepted vals: "log" (bin colour corresponds to frequency), integer (specifies number of bins), sequence (specifies bin lower bounds)

proximity_ratio (*gamma=1.0*)

Calculate the proximity ratio (E) and return an array of values.

Arguments: None

Keyword arguments:

•gamma (default value 1.0): the instrumental gamma-factor

Calculation:

E = nA / (nA + gamma*nD) for nA and nD photons in the acceptor and donor channels respectively

subtract_bckd(*bckd_d*, *bckd_a*)
Subtract background noise from donor and acceptor channel data.

Arguments:

•bckd_d: average noise per time-bin in the donor channel

•bckd_a: average noise per time-bin in the acceptor channel

subtract_crosstalk(*ct_d*, *ct_a*)
Subtract crosstalk from donor and acceptor channels.

Arguments:

•ct_d: fractional cross-talk from donor to acceptor (float between 0 and 1)

•ct_a: fractional cross-talk from acceptor to donor (float between 0 and 1)

threshold_AND(*D_T*, *A_T*)
Select data based on the AND thresholding criterion.

Arguments:

•D_T: threshold for the donor channel

•A_T: threshold for the acceptor channel

An event is above threshold if nD > donor_threshold AND nA > acceptor_threshold for nD and nA photons in the donor and acceptor channels respectively

threshold_OR(*D_T*, *A_T*)
Select data based on the OR thresholding criterion.

Arguments:

•D_T: threshold for the donor channel

•A_T: threshold for the acceptor channel

An event is above threshold in nD > donor_threshold OR nA > acceptor_threshold for nD and nA photons in the donor and acceptor channels respectively

threshold_SUM(*T*)
Select data based on the SUM thresholding criterion.

Arguments: T: threshold above which a time-bin is accepted as a fluorescent burst

An event is above threshold in nD + nA > threshold for nD and nA photons in the donor and acceptor channels respectively

pyFRET.**fit_mixture**(*data*, *ncomp=1*)
Fit data using Gaussian mixture model

Arguments:

•data: data to be fitted, as a numpy array

Key-word arguments:

•ncomp (default value 1): number of components in the mixture model.

pyFRET.**parse_bin**(*filepath*, *filelist*, *bits=8*)

Read data from a list of binary files and return a FRET_data object.

Arguments:

- filepath: the path to the folder containing the files
- filelist: list of files to be analysed

Keyword arguments:

- bits (default value 8): the number of bits used to store a donor-acceptor pair of time-bins

Note: This file structure is probably specific to the Klenerman group's .dat files. Please don't use it unless you know you have the same filetype!

pyFRET.**parse_csv**(*filepath*, *filelist*, *delimiter=', '*)

Read data from a list of csv and return a FRET_data object.

Arguments:

- filepath: the path to the folder containing the files
- filelist: list of files to be analysed

Keyword arguments:

- delimiter (default ","): the delimiter between values in a row of the csv file.

This function assumes that each row of your file has the format: "donor_item,acceptor_item"

If your data does not have this format (for example if you have separate files for donor and acceptor data), this function will not work well for you.

CHAPTER 3

pyALEX Reference

class pyALEX.**ALEX_bursts**(*D_D*, *D_A*, *A_D*, *A_A*, *burst_starts*, *burst_ends*)

This class holds single molecule burst data. Photon bursts are stored in numpy arrays. There is a separate array for each of the four photon streams, for the start and end of each burst and for the burst duration.

The four attributes corresponding to bursts from the four photon streams from an ALEX experiment are numpy arrays:

- D_D: Donor channel when the donor laser is on
- D_A: Donor channel when the acceptor laser is on
- A_D: Acceptor channel when the donor laser is on
- A_A: Acceptor channel when the acceptor laser is on

The three further attributes, corresponding to burst duration, burst start time and burst end time are also numpy arrays:

- burst_len: Length (in bins) of each identified burst
- burst_starts: Start time (bin number) of each identified burst
- burst_ends: End time (bn number) of each identified burst

The class can be initialized directly from six lists or arrays: four of photon counts, the burst start times and the burst end times: bursts = ALEX_bursts(D_D_events, D_A_events, A_D_events, A_A_events, burst_starts, burst_ends).

However, it is more typically achieved by running the APBS or DCBS algorithm that forms part of the ALEX_data class.

denoise_bursts(*N_DD*, *N_DA*, *N_AD*, *N_AA*)

Subtract background noise from ALEX bursts.

Arguments:

- N_DD: average noise per time-bin in the channel D_D
- N_DA: average noise per time-bin in the channel D_A
- N_AD: average noise per time-bin in the channel A_D
- N_AA: average noise per time-bin in the channel A_A

scatter_intensity(*filepath, imgname, imgtype='pdf', labels=['Burst Duration', 'Burst Intensity']*)

Plot a scatter plot of burst brightness vs burst duration

Arguments: * filepath: file path to the directory in which the image will be saved * imgname: name under which the image will be saved

Keyword arguments: * imgtype: filetype of histogram image. Accepted values: jpg, tiff, rgba, png, ps, svg, eps, pdf * labels: labels for x and y axes, as a 2-element list of strings: ["x-title", "y-title"]. Default value: ["Burst Duration", "Burst Intensity"]

class pyALEX.**ALEX_data**(*D_D*, *D_A*, *A_D*, *A_A*)

This class holds single molecule data.

It has four attributes, corresponding to the four photon streams from an ALEX experiment. These are numpy arrays:

- D_D: Donor channel when the donor laser is on
- D_A: Donor channel when the acceptor laser is on
- A_D: Acceptor channel when the donor laser is on
- A_A: Acceptor channel when the acceptor laser is on

It can be initialized from four lists or four arrays of photon counts: data = FRET_data(D_D_events, D_A_events, A_D_events, A_A_events)

APBS(*T*, *M*, *L*)

All-photon bust search algorithm as implemented in Nir et al. J Phys Chem B. 2006 110(44):22103-24. Calls _runningMean and _APBS_bursts. Returns an ALEX_bursts object.

Arguments: * T: time-window (in bins) over which to sum photons (integer) * M: number of photons in window of length T required to identify a potential burst (integer) * L: total number of photons required for an identified burst to be accepted (integer)

From Nir et al.: The start (respectively, the end) of a potential burst is detected when the number of photons in the averaging window of duration T is larger (respectively, smaller) than the minimum number of photons M. A potential burst is retained if the number of photons it contains is larger than a minimum number L.

DCBS(*T*, *M*, *L*)

Dual-channel bust search algorithm as implemented in Nir et al. J Phys Chem B. 2006 110(44):22103-24. Returns an ALEX_bursts object.

Arguments: * T: time-window (in bins) over which to sum photons * M: number of photons in window of length T required to identify a potential burst. * L: total number of photons required for an identified burst to be accepted.

From Nir et al.: The start (respectively, the end) of a potential burst is detected when the number of photons in the averaging window of duration T is larger (respectively, smaller) than the minimum number of photons M. A potential burst is retained if the number of photons it contains is larger than a minimum number L.

build_histogram(*filepath*, *csvname*, *gamma=1.0*, *S_min=0.1*, *S_max=1.0*, *bin_min=0.0*, *bin_max=1.0*, *bin_width=0.02*, *image=False*, *imgname=None*, *imgtype=None*, *gauss=True*, *gaussname=None*, *n_gauss=1*)

Build a proximity ratio histogram and save the frequencies and bin centres as a csv file. Optionally plot and save a graph and perform a simple gaussian fit.

Arguments:

- filepath: path to folder where the histogram will be saved (as a string)
- csvname: the name of the file in which the histogram will be saved (as a string)

Keyword arguments:

- •gamma: Instrumental gamma factor. (float, default value 1.0)
- •S_min: the miniumum stoichiometric value for which to accept a burst (default 0.1)
- •S_max: the maximum stoichiometric value for which to accept a burst (default 0.9)
- •bin_min: the minimum value for a histogram bin (default 0.0)
- •bin_max: the maximum value for a histogram bin (default 1.0)
- •bin_width: the width of one bin (default 0.02)
- •image: Boolean. True plots a graph of the histogram and saves it (default False)
- •imgname: the name of the file in which the histogram graph will be saved (as a string)
- •imgtype: filetype of histogram image. Accepted values: jpg, tiff, rgba, png, ps, svg, eps, pdf
- •gauss: Boolean. True will fit the histogram with a single gaussian distribution (default False)
- •gaussname: the name of the file in which the parameters of the Gaussian fit will be saved
- •n_gauss: number of Gaussain distributions to fit. Default = 1

proximity_ratio(*gamma=1.0*)

Calculate the proximity ratio (E) and return an array of values.

Arguments: None

Keyword arguments: gamma (default value 1.0): the instrumental gamma-factor

Calculation: E = nA / (nA + gamma*nD) for nA and nD photons in the acceptor (A_D) and donor (D_D) channels respectively

scatter_hist(*S_min*, *S_max*, *gamma=1.0*, *save=False*, *filepath=None*, *imgname=None*, *imgtype=None*)

Plot a scatter plot of E (proximity ratio) vs S (stoichiometry) and projections of selected E and S values

Arguments:

- •S_min: minimum accepted value of S (float between 0 and 1)
- •S_max: maximum accepted value of S (float between 0 and 1)

Keyword arguments:

- •gamma: Instrumental gamma factor. (float, default value 1.0)
- •save: Boolean. True will save an image of the graph plotted (default False)
- •filepath: file path to the directory in which the image will be saved (default None)
- •imgname: name under which the image will be saved (default None)
- •imgtype: filetype of histogram image. Accepted values: jpg, tiff, rgba, png, ps, svg, eps, pdf

stoichiometry(*gamma=1.0*)

Calculate the stoichiometry (S) and return an array of values.

Arguments: None

Keyword arguments: gamma (default value 1.0): the instrumental gamma-factor

Calculation: S = (gamma*D_D + A_D) / (gamma*D_D + A_D + A_A)

stoichiometry_selection(*S*, *S_min*, *S_max*)

Select data with photons above a threshold.

Arguments:

- S: array of stoichiometry values calculated using the stoichiometry method
- S_min: minimum accepted value of S (float)
- S_max: maximum accepted value of S (float)

Event selection criterion: Smin < Sx < Smax, for Stoichiometry Sx of event x

subtract_bckd(*bckd_D_D*, *bckd_D_A*, *bckd_A_D*, *bckd_A_A*)
Subtract background noise from the four data channels.

Arguments:

- bckd_D_D: average noise per time-bin in the channel D_D
- bckd_D_A: average noise per time-bin in the channel D_A
- bckd_A_D: average noise per time-bin in the channel A_D
- bckd_A_A: average noise per time-bin in the channel A_A

subtract_crosstalk(*l*, *d*)
Subtract crosstalk from the FRET channel A_D

Arguments:

- l: leakage constant from donor channel D_D to acceptor channel A_D (float between 0 and 1)
- d: direct excitation of the acceptor by the donor laser (float between 0 and 1)

thresholder(*T_D*, *T_A*)
Select events that have photons above a threshold.

Arguments:

- T_D: threshold for photons during donor laser excitation
- T_A: threshold for photons during acceptor laser excitation

An event is above threshold if nA_D + nD_D > T_D AND nA_A > T_A for nA_D, nD_D and nA_A photons in the channels A_D, D_D and A_A respectively

pyALEX.**fit_mixture**(*data*, *ncomp=1*)
Fit data using Gaussian mixture model

Arguments:

- data: data to be fitted, as a numpy array

Key-word arguments:

- ncomp (default value 1): number of components in the mixture model.

pyALEX.**parse_bin**(*filepath*, *filelist*, *bits=16*)
Read data from a list of binary files and return an ALEX_data object.

Arguments: * filepath: the path to the folder containing the files * filelist: list of files to be analysed

Keyword arguments: * bits (default value 16): the number of bits used to store a donor-acceptor pair of time-bins

Note: This file structure is probably specific to the Klenerman group's .dat files. Please don't use it unless you know you have the same filetype!

pyALEX.**parse_csv**(*filepath*, *filelist*, *delimiter=', '*)
Read data from a list of csv and return a FRET_data object.

Arguments:

- filepath: the path to the folder containing the files

- filelist: list of files to be analysed

Keyword arguments:

- delimiter (default ","): the delimiter between values in a row of the csv file.

This function assumes that each row of your file has the format: "D_D,D_A,A_D,A_A"

If your data does not have this format (for example if you have separate files for donor and acceptor data), this function will not work well for you.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

p