
Pycorrelate Documentation

Release 0.3+24.gf9e109e.dirty

Antonino Ingargiola

Aug 20, 2018

Contents

1 Documentation	3
Python Module Index	23

Pycorrelate computes fast and accurate cross-correlation over arbitrary time lags. Cross-correlations can be calculated on “uniformly-sampled” signals or on “point-processes”, such as photon timestamps. Pycorrelate allows computing cross-correlation at log-spaced lags covering several orders of magnitude. This type of cross-correlation is commonly used in physics or biophysics for techniques such as *fluorescence correlation spectroscopy (FCS)* or *dynamic light scattering (DLS)*.

Two types of correlations are implemented:

- **ucorrelate**: the classical textbook linear cross-correlation between two signals defined at **uniformly-spaced** intervals (both signals having the same interval size). Only positive lags are computed and a max lag can be specified. Thanks to the limit in the computed lags, this function can be much faster than `numpy.correlate`.
- **pcorrelate**: cross-correlation of discrete events in a point-process (i.e. a series of timestamps). Input arrays are timestamps (or “positions”) of events, for example **photon arrival times**. This function implements the algorithm in Laurence et al. *Optics Letters* (2006). This is a generalization of the multi-tau algorithm which retains high execution speed while allowing arbitrary time-lag bins.

Pycorrelate is implemented in Python 3 and operates on standard numpy arrays. Execution speed is optimized using `numba`.

- Free software: GNU General Public License v3
- Documentation: <https://pycorrelate.readthedocs.io>.

1.1 Installation

1.1.1 Stable release

To install Pycorrelate, run this command in your terminal:

```
$ pip install pycorrelate
```

This is the preferred method to install Pycorrelate, as it will always install the most recent stable release.

If you don't have `pip` installed, this [Python installation guide](#) can guide you through the process.

1.1.2 From sources

The sources for Pycorrelate can be downloaded from the [Github repo](#).

You can either clone the public repository:

```
$ git clone git://github.com/tritemio/pycorrelate
```

Or download the `tarball`:

```
$ curl -OL https://github.com/tritemio/pycorrelate/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

1.2 Usage

Imports:

```
import numpy as np
import pycorrelate as pyc
```

Create two arrays *t* and *u* of discrete events, exponentially correlated:

```
np.random.seed(1)
size = 10**4
t = np.sort(np.random.randint(0, 10**5, size=size))
u = np.sort(t + np.random.exponential(scale=10, size=t.size).astype(np.int64))
```

Compute correlation:

```
lags = np.arange(0, 201)
G = pyc.pcorrelate(t, u, lags)
```

G contains the cross-correlation of *t* and *u* at the defined *lags*.

For more examples see *Pycorrelate examples*.

1.3 API Reference

Quick links:

- `pcorrelate()`
- `pnormalize()`
- `make_loglags()`
- `ucorrelate()`

1.3.1 List of Pycorrelate functions

Functions to compute linear correlation on discrete signals (uniformly sampled in time) **or** on point-processes (e.g. timestamps of events).

`pycorrelate.pycorrelate.make_loglags(exp_min, exp_max, points_per_base, base=10, return_int=True)`

Make a log-spaced array useful as lag bins for cross-correlation.

This function creates an arrays of log-spaced time-lag bins to be used with `pcorrelate()`. By default it returns integer time-lag bins to avoid floating point inaccuracies in the correlation (showing up as higher noise at small time-lags).

Parameters

- **exp_min** (*int*) – exponent of the minimum value
- **exp_max** (*int*) – exponent of the maximum value
- **points_per_base** (*int*) – number of points per base (i.e. in a decade when *base = 10*)
- **base** (*int*) – base of the exponent. Default 10.

- **return_int** (*bool*) – if True (default) return integer bin edges to avoid floating point inaccuracies. If False, returned bin edges are float.

Returns Array of log-spaced values with specified range and spacing.

Example

Compute log₁₀-spaced bins with 5 bins per decade, starting from 1 (10⁰) and stopping at 10⁶:

```
>>> make_loglags(0, 6, 5)
array([[ 1,    2,    3,    4,    6,   10,   16,
        25,   40,   63,  100,  158,  251,  398,
        631, 1000, 1585, 2512, 3981, 6310, 10000,
        15849, 25119, 39811, 63096, 100000, 158489, 251189,
        398107, 630957, 1000000]])
```

Compute log₁₀-spaced bins with 2 bins per decade, starting from 10¹ and stopping at 10³:

```
>>> make_loglags(-1, 3, 2, return_int=False)
array([[ 1.00000000e-01,  3.16227766e-01,  1.00000000e+00,
        3.16227766e+00,  1.00000000e+01,  3.16227766e+01,
        1.00000000e+02,  3.16227766e+02,  1.00000000e+03]])
```

See also:

`pcorrelate()`

`pycorrelate.pycorrelate.pcorrelate`

Compute correlation of two arrays of discrete events (Point-process).

The input arrays need to be values of a point process, such as photon arrival times or positions. The correlation is efficiently computed on an arbitrary array of lag-bins. As an example, bins can be uniformly spaced in log-space and span several orders of magnitudes. (you can use `make_loglags()` to create log-spaced bins). This function implements the algorithm described in (Laurence 2006).

Parameters

- **t** (*array*) – first array of “points” to correlate. The array needs to be monotonically increasing.
- **u** (*array*) – second array of “points” to correlate. The array needs to be monotonically increasing.
- **bins** (*array*) – bin edges for lags where correlation is computed.
- **normalize** (*bool*) – if True, normalize the correlation function as typically done in FCS using `pnormalize()`. If False, return the unnormalized correlation function.

Returns Array containing the correlation of *t* and *u*. The size is `len(bins) - 1`.

See also:

`make_loglags()` to generate log-spaced lag bins.

`pycorrelate.pycorrelate.pnormalize`

Normalize point-process cross-correlation function.

This normalization is usually employed for fluorescence correlation spectroscopy (FCS) analysis. The normalization is performed according to (Laurence 2006). Basically, the input argument *G* is multiplied by:

$$\frac{T - \tau}{n(\{i \ni t_i \leq T - \tau\})n(\{j \ni u_j \geq \tau\})}$$

where $n(\{\})$ is the operator counting the elements in a set, t and u are the input arrays of the correlation, τ is the time lag and T is the measurement duration.

Parameters

- **G** (*array*) – raw cross-correlation to be normalized.
- **t** (*array*) – first input array of “points” used to compute G .
- **u** (*array*) – second input array of “points” used to compute G .
- **bins** (*array*) – array of bins used to compute G . Needs to have the same units as input arguments t and u .

Returns Array of normalized values for the cross-correlation function, same size as the input argument G .

`pycorrelate.pycorrelate.ucorrelate`

Compute correlation of two signals defined at uniformly-spaced points.

The correlation is defined only for positive lags (including zero). The input arrays represent signals defined at uniformly-spaced points. This function is equivalent to `numpy.correlate()`, but can efficiently compute correlations on a limited number of lags.

Note that binning point-processes with uniform bins, provides signals that can be passed as argument to this function.

Parameters

- **tx** (*array*) – first signal to be correlated
- **ux** (*array*) – second signal to be correlated
- **maxlag** (*int*) – number of lags where correlation is computed. If `None`, computes all the lags where signals overlap $\min(tx.size, tu.size) - 1$.

Returns Array contained the correlation at different lags. The size of this array is equal to the input argument *maxlag* (if defined) or to $\min(tx.size, tu.size) - 1$, and its *dtype* is the same as argument *t*'s.

Example

Correlation of two signals t and u :

```
>>> t = np.array([1, 2, 0, 0])
>>> u = np.array([0, 1, 1])
>>> pycorrelate.ucorrelate(t, u)
array([2, 3, 0])
```

The same result can be obtained with `numpy` swapping t and u and restricting the results only to positive lags:

```
>>> np.correlate(u, t, mode='full')[t.size - 1:]
array([2, 3, 0])
```

Also works with other types:

```
>>> t = np.array([1.2, 2.4, 0.5, 0.6])
>>> u = np.array([0, 1.2, 1.3])
>>> pycorrelate.ucorrelate(t, u)
array([3.53, 4.56, 1.56])
```

1.4 Pycorrelate examples

This notebook shows how to use pycorrelate as well as comparisons with other implementations.

```
In [1]: import numpy as np
import h5py
```

```
/home/docs/checkouts/readthedocs.org/user_builds/pycorrelate/conda/latest/lib/python3.6/importlib/_bo
return f(*args, **kwargs)
```

```
In [2]: # Tweak here matplotlib style
import matplotlib.pyplot as plt
import matplotlib as mpl
mpl.rcParams['font.sans-serif'].insert(0, 'Arial')
mpl.rcParams['font.size'] = 14
%config InlineBackend.figure_format = 'retina'
%matplotlib inline
```

```
In [3]: import pycorrelate as pyc
print('pycorrelate version: ', pyc.__version__)
```

```
pycorrelate version: 0.3+24.gf9e109e
```

1.4.1 Load Data

We start by downloading some timestamps data:

```
In [4]: url = 'http://files.figshare.com/2182601/0023uLRpitc_NTP_20dT_0.5GndCl.hdf5'
pyc.utils.download_file(url, save_dir='data')
```

```
URL: http://files.figshare.com/2182601/0023uLRpitc_NTP_20dT_0.5GndCl.hdf5
```

```
File: 0023uLRpitc_NTP_20dT_0.5GndCl.hdf5
```

```
File already on disk: data/0023uLRpitc_NTP_20dT_0.5GndCl.hdf5
Delete it to re-download.
```

```
In [5]: fname = './data/' + url.split('/')[-1]
h5 = h5py.File(fname)
unit = 12.5e-9
```

```
In [6]: num_ph = int(3e6)
detectors = h5['photon_data']['detectors'][:num_ph]
timestamps = h5['photon_data']['timestamps'][:num_ph]
t = timestamps[detectors == 0]
u = timestamps[detectors == 1]
```

```
In [7]: t.shape, u.shape, t[0], u[0]
```

```
Out[7]: ((839592,), (1844370,), 146847, 188045)
```

```
In [8]: t.max()*unit, u.max()*unit
```

```
Out[8]: (599.9994419125, 599.9984722875)
```

Timestamps need to be monotonic, let's test it:

```
In [9]: assert (np.diff(t) > 0).all()
assert (np.diff(u) > 0).all()
```

1.4.2 Log-scale bins (base 10)

Here we compute the cross-correlation on log10-spaced bins.

First we compute the array of lag bins using the function `make_loglags`:

```
In [10]: # compute lags in sec. then convert to timestamp units
        bins = pyc.make_loglags(-7, 0, 10, return_int=False) / unit
```

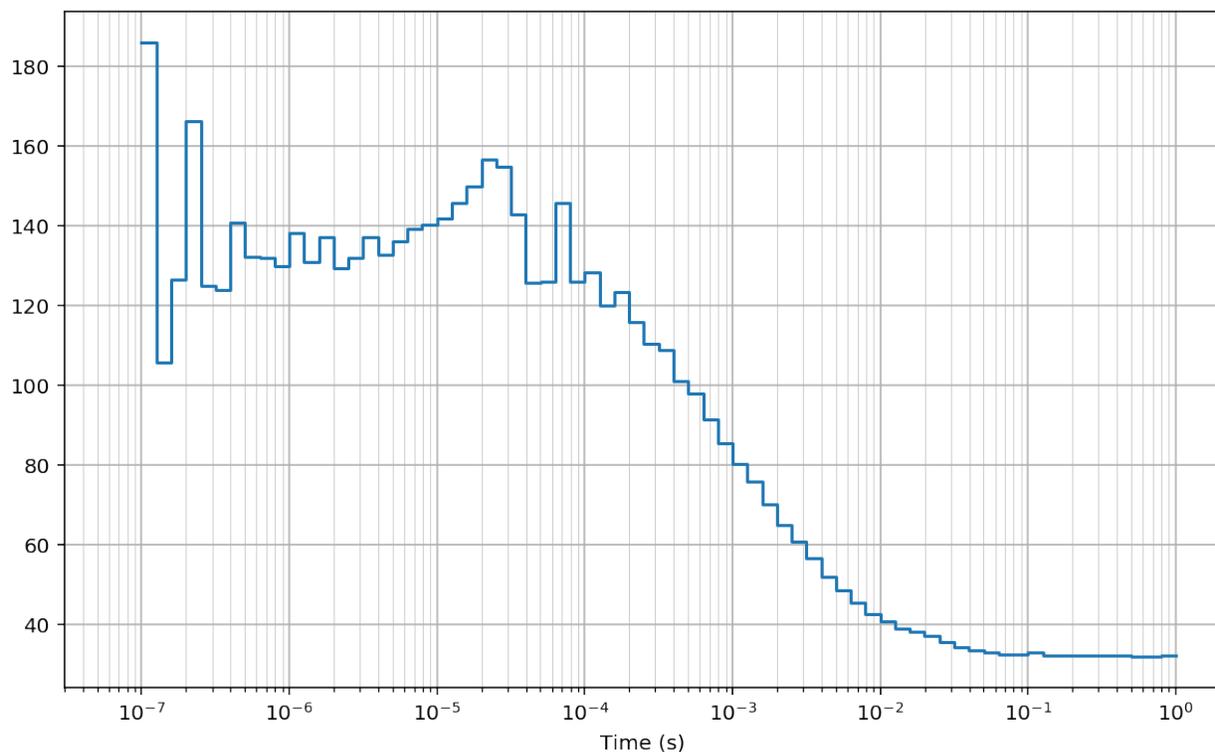
Then, we compute the cross-correlation using the function `pcorrelate`:

```
In [11]: G = pyc.pcorrelate(t, u, bins)
```

```
/home/docs/checkouts/readthedocs.org/user_builds/pycorrelate/conda/latest/lib/python3.6/importlib/_bo
return f(*args, **kwargs)
```

```
In [12]: fig, ax = plt.subplots(figsize=(10, 6))
        plt.plot(bins*unit, np.hstack((G[:1], G)), drawstyle='steps-pre')
        plt.xlabel('Time (s)')
        #for x in bins[1:]: plt.axvline(x*unit, lw=0.2) # to mark bins
        plt.grid(True); plt.grid(True, which='minor', lw=0.3)
        plt.xscale('log')
        plt.xlim(30e-9, 2)
```

```
Out [12]: (3e-08, 2)
```



1.4.3 Log-scale bins (base 2)

Here we compute the same cross-correlation on `log2`-spaced bins.

First we compute the array of lag bins using the function `make_loglags`:

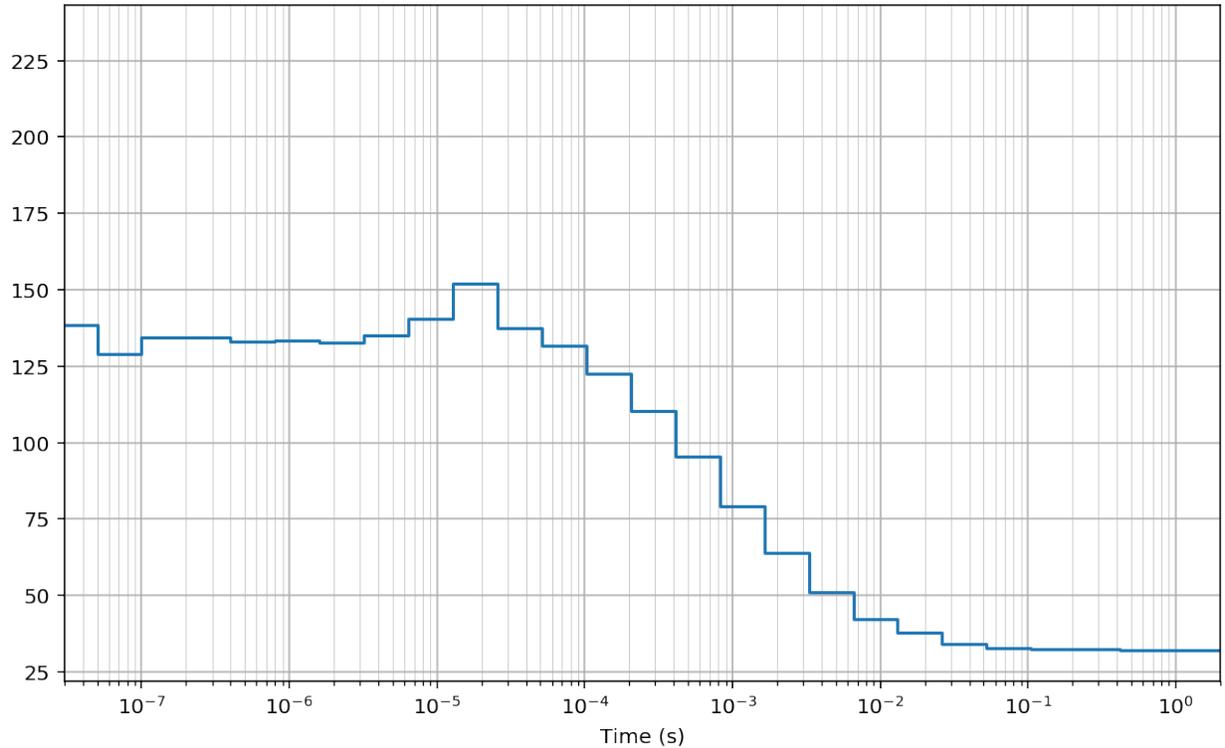
```
In [13]: # compute lags directly in timestamp units
        bins = pyc.make_loglags(0, 28, 1, base=2)
```

Then, we compute the cross-correlation using the function `pcorrelate`:

```
In [14]: G = pyc.pcorrelate(t, u, bins)
```

```
In [15]: fig, ax = plt.subplots(figsize=(10, 6))
plt.plot(bins*unit, np.hstack((G[:1], G)), drawstyle='steps-pre')
plt.xlabel('Time (s)')
#for x in bins[1:]: plt.axvline(x*unit, lw=0.2) # to mark bins
plt.grid(True); plt.grid(True, which='minor', lw=0.3)
plt.xscale('log')
plt.xlim(30e-9, 2)
```

```
Out[15]: (3e-08, 2)
```



1.4.4 Multi-tau bins

Finally, we compute the cross-correlation on arbitrarily-spaced bins. Similar to the multi-tau bins, here we use constant bin size for a number of bins (`n_group`), then we double the bin size and we keep it constant for another `n_group` and so on:

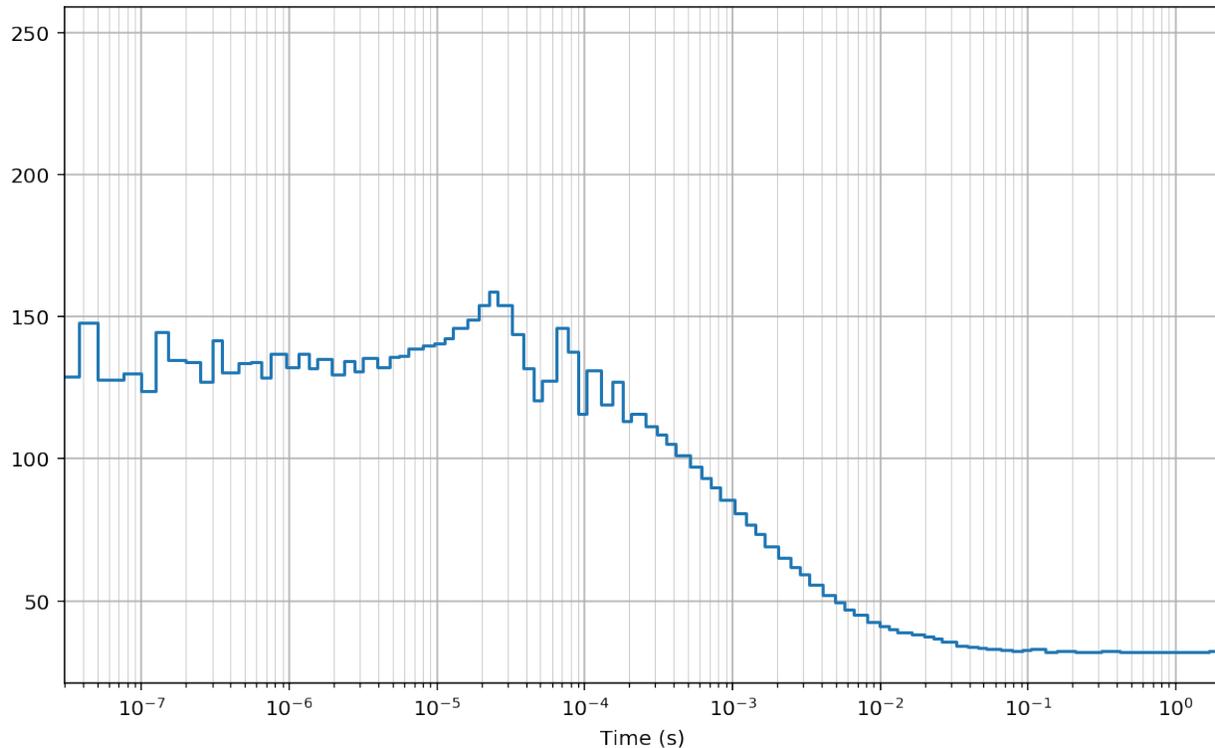
```
In [16]: n_group = 4
bin_widths = []
for i in range(26):
    bin_widths += [2**i]*n_group
np.array(bin_widths)
bins = np.hstack(([0], np.cumsum(bin_widths)))
```

Then, we compute the cross-correlation using the function `pcorrelate`:

```
In [17]: G = pyc.pcorrelate(t, u, bins)
In [18]: fig, ax = plt.subplots(figsize=(10, 6))
plt.plot(bins*unit, np.hstack((G[:1], G)), drawstyle='steps-pre')
plt.xlabel('Time (s)')
#for x in bins[1:]: plt.axvline(x*unit, lw=0.2) # to mark bins
plt.grid(True); plt.grid(True, which='minor', lw=0.3)
```

```
plt.xscale('log')
plt.xlim(30e-9, 2)
```

```
Out[18]: (3e-08, 2)
```



1.4.5 Test: comparison with np.histogram

For testing alternative (slower) implementations we use smaller input arrays:

```
In [19]: tt = t[:5000]
         uu = u[:5000]
```

The algorithm implemented in `pycorrelate.pcorrelate` can be re-written in a very simple way using `numpy.histogram`:

```
In [20]: # compute lags in sec. then convert to timestamp units
         bins = pyc.make_loglags(-7, 0, 10, return_int=False) / unit
```

```
In [21]: Y = np.zeros(bins.size - 1, dtype=np.int64)
         for ti in tt:
             Yc, _ = np.histogram(uu - ti, bins=bins)
             Y += Yc
         G = Y / np.diff(bins)
```

```
In [22]: assert (G == pyc.pcorrelate(tt, uu, bins)).all()
```

Test passed! Here we demonstrated that the logic of the algorithm is implemented as described in the paper (and in the few lines of code above).

1.4.6 Tests: comparison with np.correlate

The comparison with `np.correlate` is a little tricky. First we need to bin our input to create timetraces that can be correlated by linear convolution. For testing purposes, let's use a small portion of the timetraces:

```
In [23]: binwidth = 50e-6
        bins_tt = np.arange(0, tt.max()*unit, binwidth) / unit
        bins_uu = np.arange(0, uu.max()*unit, binwidth) / unit
```

```
In [24]: bins_tt.max()*unit, bins_tt.size
```

```
Out[24]: (4.13725, 82746)
```

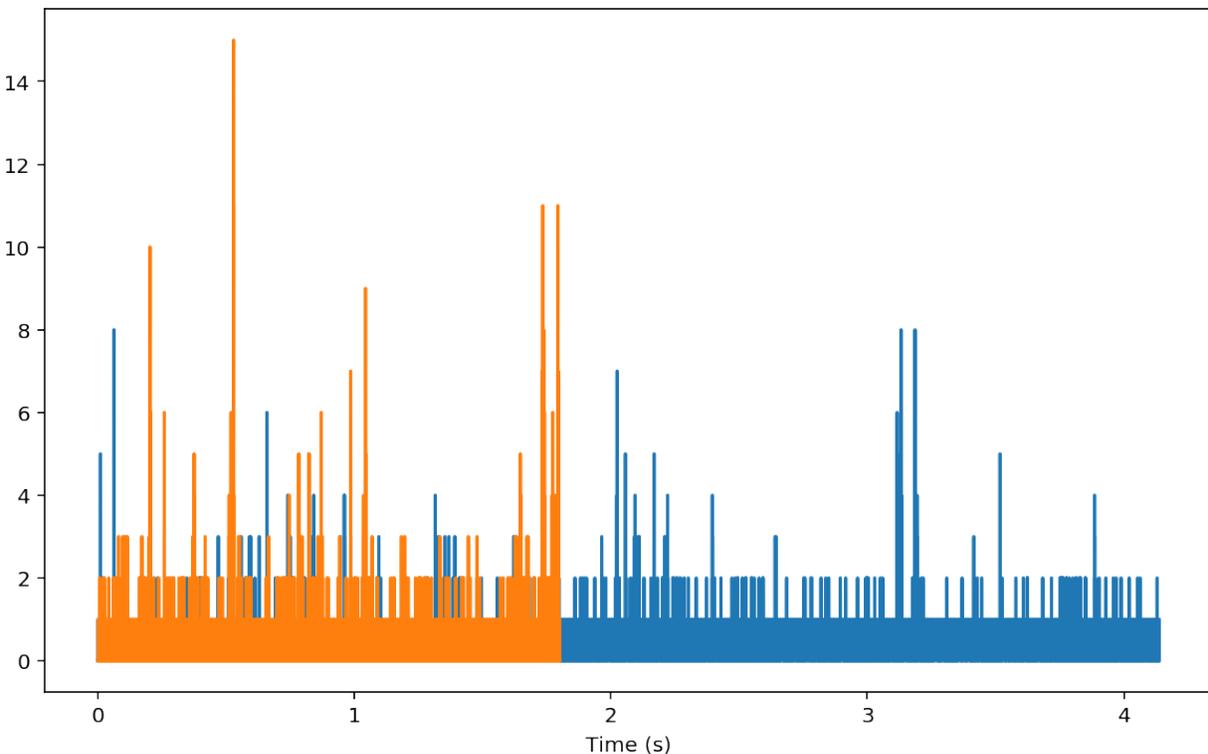
```
In [25]: bins_uu.max()*unit, bins_uu.size
```

```
Out[25]: (1.8020999999999998, 36043)
```

```
In [26]: tx, _ = np.histogram(tt, bins=bins_tt)
        ux, _ = np.histogram(uu, bins=bins_uu)
```

```
plt.figure(figsize=(10, 6))
plt.plot(bins_tt[1:]*unit, tx)
plt.plot(bins_uu[1:]*unit, ux)
plt.xlabel('Time (s)')
```

```
Out[26]: Text(0.5,0,'Time (s)')
```



The plots above are the two curves we are going to feed to `np.correlate`:

```
In [27]: C = np.correlate(ux, tx, mode='full')
```

We need to trim the result to obtain a proper alignment with the 0-time lag:

```
In [28]: Gn = C[tx.size-1:] # trim to positive time lags
```

Now, we can check that both `numpy.correlate` and `pycorrelate.ucorrelate` give the same result:

```
In [29]: Gu = pyc.ucorrelate(tx, ux)
         assert (Gu == Gn).all()
```

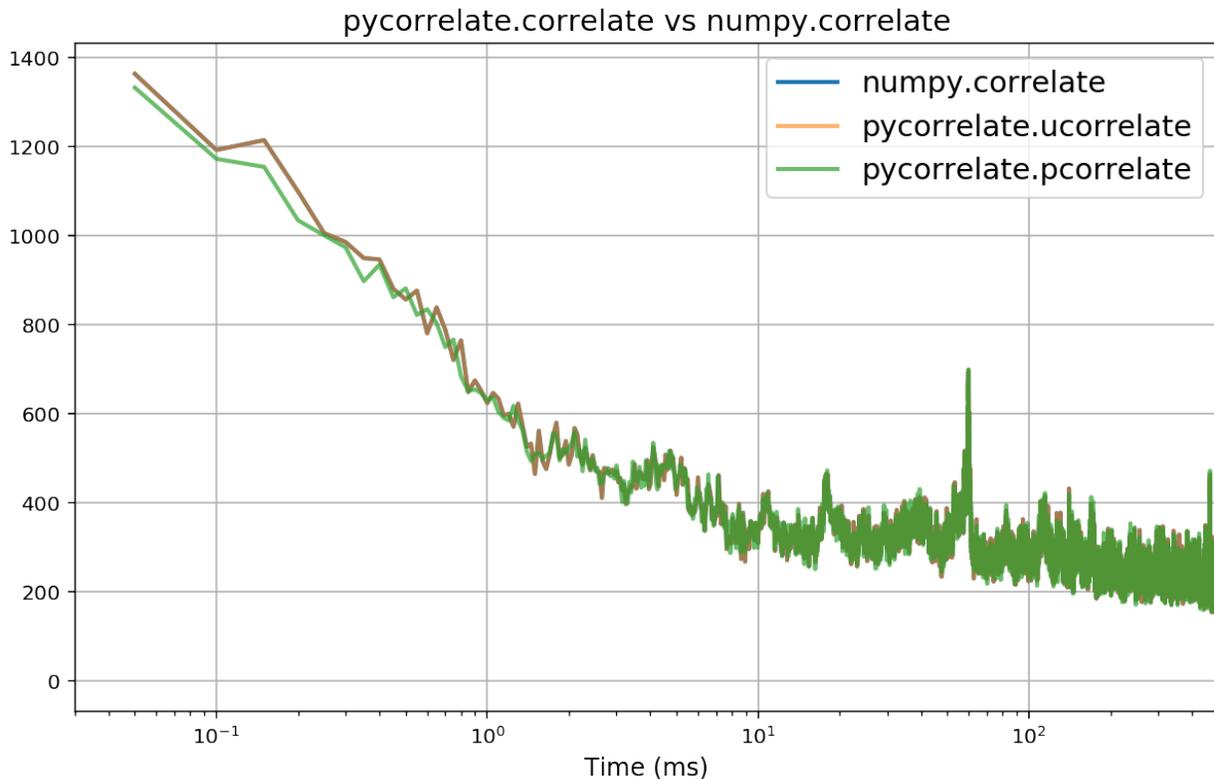
Now, let's compute the correlation also with `pycorrelate.pcorrelate`:

```
In [30]: maxlag_sec = 3.9
         lagbins = (np.arange(0, maxlag_sec, binwidth) / unit).astype('int64')
```

```
In [31]: Gp = pyc.pcorrelate(tt, uu, lagbins) * int(binwidth / unit)
```

Let's plot a comparison:

```
In [32]: fig, ax = plt.subplots(figsize=(10, 6))
         Gn_t = np.arange(1, Gn.size+1) * binwidth * 1e3
         Gu_t = np.arange(1, Gu.size+1) * binwidth * 1e3
         Gp_t = lagbins[1:] * unit * 1e3
         plt.plot(Gn_t, Gn, alpha=1, lw=2, label='numpy.correlate')
         plt.plot(Gu_t, Gu, alpha=0.6, lw=2, label='pycorrelate.ucorrelate')
         plt.plot(Gp_t, Gp, alpha=0.7, lw=2, label='pycorrelate.pcorrelate')
         plt.xlabel('Time (ms)', fontsize='large')
         plt.grid(True)
         plt.xlim(30e-3, 500)
         plt.xscale('log')
         plt.title('pycorrelate.correlate vs numpy.correlate', fontsize='x-large')
         plt.legend(loc='best', fontsize='x-large');
```



1.4.7 Conclusion

- `numpy.correlate` and `pycorrelate.ucorrelate` give identical results, with the latter being much faster. Note that the inputs are swapped between the two functions.
- `pycorrelate.ucorrelate` and `pycorrelate.pcorrelate` agree when using uniform time-lag bins.

1.5 Simple FCS example

This notebook shows howto compute and fit an FCS curve using pycorrelate.

1.5.1 Initial imports

```
In [1]: import numpy as np
import h5py
```

```
/home/docs/checkouts/readthedocs.org/user_builds/pycorrelate/conda/latest/lib/python3.6/importlib/_bo
return f(*args, **kwargs)
```

```
In [2]: # Tweak here matplotlib style
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
import matplotlib.pyplot as plt
import matplotlib as mpl
mpl.rcParams['font.sans-serif'].insert(0, 'Arial')
mpl.rcParams['font.size'] = 14
```

```
In [3]: import lmfit
import pycorrelate as pyc
```

```
print('lmfit version:      ', lmfit.__version__)
print('pycorrelate version: ', pyc.__version__)
```

```
/home/docs/checkouts/readthedocs.org/user_builds/pycorrelate/conda/latest/lib/python3.6/importlib/_bo
return f(*args, **kwargs)
```

```
/home/docs/checkouts/readthedocs.org/user_builds/pycorrelate/conda/latest/lib/python3.6/importlib/_bo
return f(*args, **kwargs)
```

```
lmfit version:      0.9.11
pycorrelate version: 0.3+24.gf9e109e
```

1.5.2 Load Data

We start downloading a sample dataset of a smFRET “measurement” with a single CW excitation laser and two detectors donor (D) and acceptor (A) (the data is actually a simulation performed with [PyBroMo](#)).

```
In [4]: url = 'http://files.figshare.com/4917046/smFRET_44f3da_P_20_s0_20_s20_D_6.0e11_6.0e11_E_75_30'
pyc.utils.download_file(url, save_dir='data')
```

```
URL: http://files.figshare.com/4917046/smFRET_44f3da_P_20_s0_20_s20_D_6.0e11_6.0e11_E_75_30_EmTot_200k_200k_BgD1500_BgA800_t_max_600s
```

```
File: smFRET_44f3da_P_20_s0_20_s20_D_6.0e11_6.0e11_E_75_30_EmTot_200k_200k_BgD1500_BgA800_t_max_600s
```

```
File already on disk: data/smFRET_44f3da_P_20_s0_20_s20_D_6.0e11_6.0e11_E_75_30_EmTot_200k_200k_BgD1500_BgA800_t_max_600s
Delete it to re-download.
```

```
In [5]: fname = './data/' + url.split('/')[-1]
h5 = h5py.File(fname)
unit = h5['photon_data']['timestamps_specs']['timestamps_unit']({})
unit
```

```
Out [5]: 5e-08
```

We can check that there are only two detectors:

```
In [6]: np.unique(h5['photon_data']['detectors'][:])
```

```
Out [6]: array([0, 1], dtype=uint8)
```

Then we load the timestamps in two arrays `t` and `u`:

```
In [7]: detectors = h5['photon_data']['detectors'][:]
        timestamps = h5['photon_data']['timestamps'][:]
        t = timestamps[detectors == 0]
        u = timestamps[detectors == 1]
```

```
In [8]: t.shape, u.shape, t[0], u[0]
```

```
Out[8]: ((1152331,), (755468,), 50, 128800)
```

```
In [9]: t.max()*unit, u.max()*unit
```

```
Out[9]: (599.999341, 599.9998935)
```

Timestamps need to be monotonic, let's check:

```
In [10]: assert (np.diff(t) >= 0).all()
         assert (np.diff(u) >= 0).all()
```

1.5.3 Compute CCF

To avoid afterpulsing, we can compute the cross-correlation function (CCF) between D and A channels.

We first create the array of time-lag bins using `make_loglags()`:

```
In [11]: # compute lags in timestamp units (not in seconds!)
        # to avoid floating point inaccuracies
        bins_per_dec = 10
        bins = pyc.make_loglags(1, 8, bins_per_dec)[bins_per_dec//2:]
        print(f'Number of time-lag bins: {bins.size}\n'
              f'First bin: {bins[0]*unit*1e9:.1f} ns \n'
              f'Last bin: {bins[-1]*unit:.2f} s')
```

```
Number of time-lag bins: 66
```

```
First bin: 1600.0 ns
```

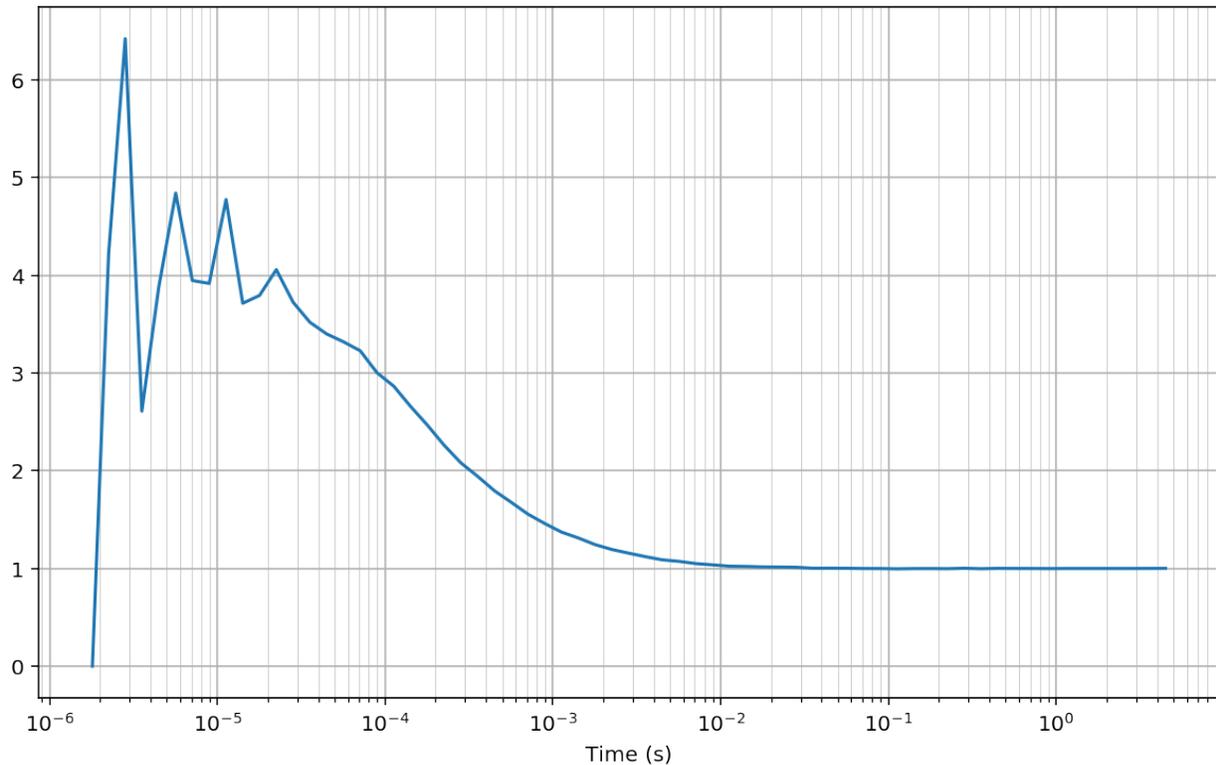
```
Last bin: 5.00 s
```

Then, we compute the cross-correlation with `pcorrelate`:

```
In [12]: Gn = pyc.pcorrelate(t, u, bins, normalize=True)
```

Plotting the CCF function `Gn` we observe the typical diffusion shape:

```
In [13]: fig, ax = plt.subplots(figsize=(10, 6))
        mean_lags = np.mean([bins[1:], bins[:-1]], 0)*unit
        plt.semilogx(mean_lags, Gn)
        #plt.semilogx(bins[1:]*unit, Gn, drawstyle='steps-pre')
        plt.xlabel('Time (s)')
        plt.grid(True); plt.grid(True, which='minor', lw=0.3);
```



1.5.4 Fit FCS model

The next step is fitting the computed CCF with a model. For freely-diffusing species under confocal excitation (and no photo-physics) the simplest model is the 2D model (i.e. the PSF z dimension is neglected):

$$G(\tau) = 1 + A_0 \left(1 + \frac{\tau}{\tau_D}\right)^{-1}$$

The full 3D model is just slightly more complicated:

$$G(\tau) = 1 + A_0 \left(1 + \frac{\tau}{\tau_D}\right)^{-1} \left[1 + \left(\frac{r}{z}\right)^2 \frac{\tau}{\tau_D}\right]^{-1/2}$$

There is a link between A_0 and concentration. Neglecting background, $A_0 = 1/N$ where N is the mean number of molecules in the excitation volume. The background makes $A_0 < 1/N$. For full expression see [Orrit 2002](#).

Here, for the sake of the example, we will just fit the simple 2D model.

Let's start defining the model functions and the array of time-lags τ :

```
In [14]: def diffusion_2d(timelag, tau_diff, A0):
         return 1 + A0 * 1/(1 + timelag/tau_diff)

         def diffusion_3d(timelag, tau_diff, A0, waist_z_ratio=0.1):
         return (1 + A0 * 1/(1 + timelag/tau_diff) *
                 1/np.sqrt(1 + waist_z_ratio**2 * timelag/tau_diff))

In [15]: tau = 0.5 * (bins[1:] + bins[:-1]) * unit
```

Now, we build a “fitting model” with `lmfit` and use it to fit the CCF curve G_n :

```
In [16]: model = lmfit.Model(diffusion_2d)
        params = model.make_params(A0=1, tau_diff=1e-3)
        params['A0'].set(min=0.01, value=1)
        params['tau_diff'].set(min=1e-6, value=1e-3)
        #params['waist_z_ratio'].set(value=1/6, vary=False) # 3D model only

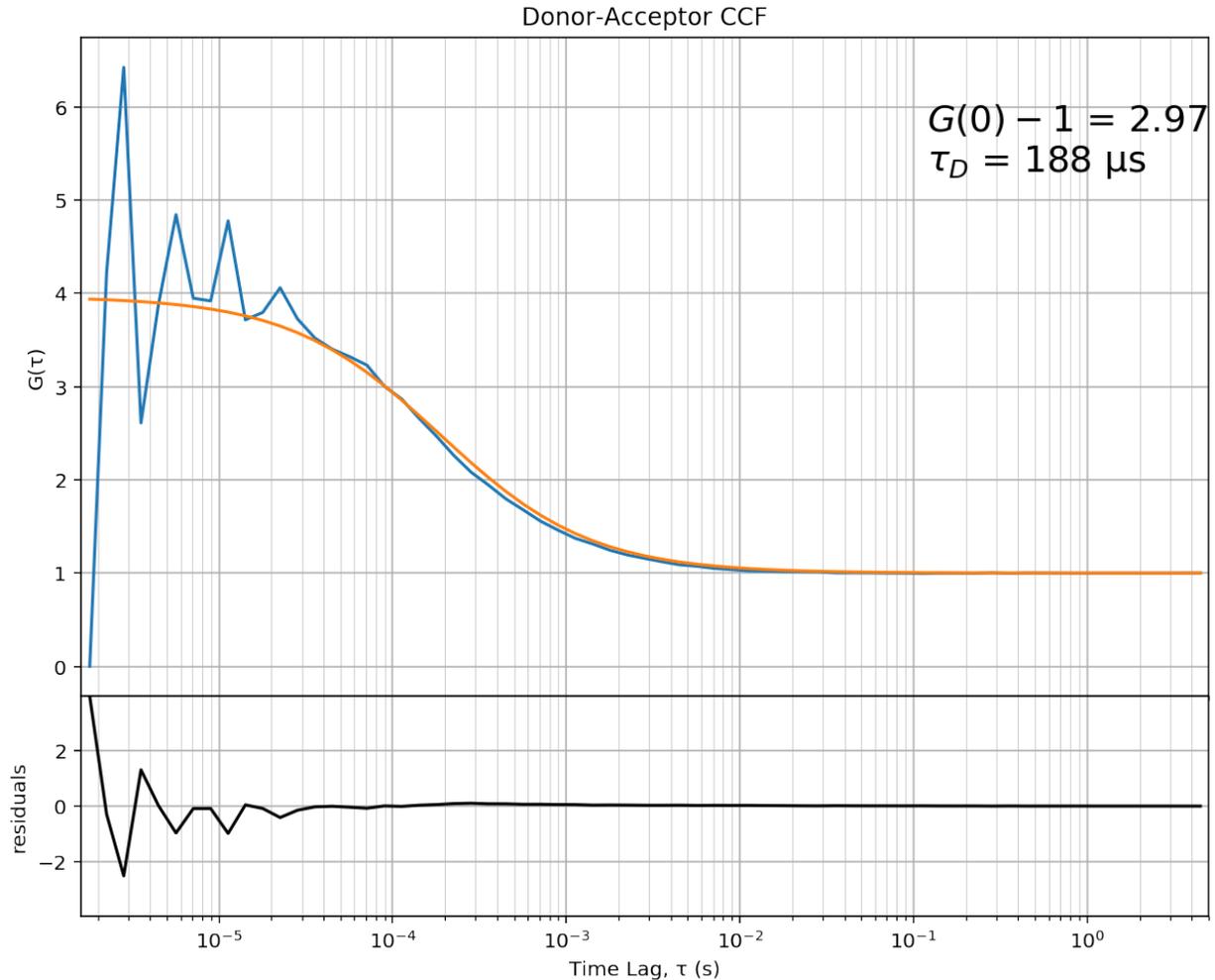
        weights = np.ones_like(Gn)
        #weights = np.log(np.sqrt(G*np.diff(bins))) # and example of using weights
        fitres = model.fit(Gn, timelag=tau, params=params, method='least_squares',
                          weights=weights)
        print('\nList of fitted parameters for %s: \n' % model.name)
        fitres.params.pretty_print(colwidth=10, columns=['value', 'min', 'max'])
```

List of fitted parameters for Model(diffusion_2d):

Name	Value	Min	Max
A0	2.966	0.01	inf
tau_diff	0.0001879	1e-06	inf

Finally, we plot fit results and residuals:

```
In [17]: fig, ax = plt.subplots(2, 1, figsize=(10, 8), sharex=True,
                               gridspec_kw={'height_ratios': [3, 1]})
        plt.subplots_adjust(hspace=0)
        ax[0].semilogx(tau, Gn)
        for a in ax:
            a.grid(True); a.grid(True, which='minor', lw=0.3)
        ax[0].plot(tau, fitres.best_fit)
        ax[1].plot(tau, fitres.residual*weights, 'k')
        ym = np.abs(fitres.residual*weights).max()
        ax[1].set_ylim(-ym, ym)
        ax[1].set_xlim(bins[0]*unit, bins[-1]*unit);
        tau_diff_us = fitres.values['tau_diff'] * 1e6
        msg = ((r'$G(0)-1$ = {A0:.2f}'+'\n'+r'$\tau_D$ = {tau_diff_us:.0f} $\mu$s')
              .format(A0=fitres.values['A0'], tau_diff_us=tau_diff_us))
        ax[0].text(.75, .9, msg,
                  va='top', ha='left', transform=ax[0].transAxes, fontsize=18);
        ax[0].set_ylabel('G( $\tau$ )')
        ax[1].set_ylabel('residuals')
        ax[0].set_title('Donor-Acceptor CCF')
        ax[1].set_xlabel('Time Lag,  $\tau$  (s)');
```



The flatness of the residual indicates a good fit. By changing the fitting function defined above (`diffusion_2d`), you can extend this example to more complex models.

1.6 Theory

1.6.1 Cross-correlation of point processes

In fluorescence correlation spectroscopy (FCS) the (normalized) cross-correlation function (CCF) of two continuous signals $I_1(t)$ and $I_2(t)$ is defined as:

$$G(\tau) = \frac{\langle I_1(t) I_2(t) \rangle}{\langle I_1(t) \rangle \langle I_2(t) \rangle}$$

The auto-correlation function (ACF) is just a special case where $I_1(t) = I_2(t)$.

In actual experiments, signals are not continuous but come from single-photon detectors that produce a pulse for each photon. These pulses are usually timestamped with $\sim 10\text{ns}$ resolution. The series of photon arrival times is used as input for ACF or CCF computations.

In principle, timestamps can be binned to produce a discrete-time signal. In signal processing, the (non-normalized)

cross-correlation of two real discrete-time signals $\{A_i\}$ and $\{B_i\}$ is defined as

$$c[k] = \sum_{i=0}^N A[i] B[i+k].$$

The previous formula is implemented by `ucorrelate()` and `numpy.correlate`. The difference is that `ucorrelate()` only computes positive lags and allows setting a max lag for efficiency.

Binning timestamps to obtain timetraces would be very inefficient for FCS analysis where time-lags span many orders of magnitude. It is much more efficient to directly compute the cross-correlation function from timestamps. The popular multi-tau algorithm allows computing the correlation directly from timestamps on a fixed arrangement of quasi-log-spaced bins. More generally, Laurence algorithm (Laurence et al. *Optics Letters* (2006)) allows computing cross-correlation from timestamps on arbitrary bins of time-lags, with similar performances as the multi-tau. Computing cross-correlation $C(\tau)$ from timestamps is fundamentally a counting task. Given two timestamps arrays t and u and considering the k -th time-lag bin $[\tau_k, \tau_{k+1})$, $C(k)$ is equal to the number of pairs where:

$$\tau_k \leq t_i - u_j < \tau_{k+1}$$

for all the possible i and j combinations. The full expression for $C(k)$ is:

$$C(k) = \frac{n(\{(i, j) \ni t_i < u_j - \Delta\tau_k\})}{\Delta\tau_k} \quad (1.1)$$

where $n(\{ \})$ is the operator counting the elements in a set, $\Delta\tau_k$ is the duration of the k -th time-lag bin and T is the measurement duration. For FCS we normally want the normalized CCF, that is:

$$G(k) = \frac{n(\{(i, j) \ni t_i < u_j - \Delta\tau_k\})}{n(\{i \ni t_i \leq T - \Delta\tau_k\}) n(\{j \ni u_j \geq \Delta\tau_k\})} \frac{(T - \Delta\tau_k)}{\Delta\tau_k} \quad (1.2)$$

Both eq. (1.1) and (1.2) are implemented by `pcorrelate()`. You can choose between the normalized and unnormalized version with the input argument `normalize`.

Note: In *Laurence 2006*, due to a typo, the expression for $G(k)$ (which they call $C_{AB}(\tau)$) is missing the term $\Delta\tau_k$ in the denominator.

References

- Laurence, T. A., Fore, S., Huser, T. (2006). Fast, flexible algorithm for calculating photon correlations. *Optics Letters*, 31 (6), 829–831. <https://doi.org/10.1364/OL.31.000829>
- Petra Schille and Elke Hausteин, *Fluorescence Correlation Spectroscopy An Introduction to its Concepts and Applications*
- Hausteин, E., Schille, P. (2003). Ultrasensitive investigations of biological systems by fluorescence correlation spectroscopy. *Methods*, 29 (2), 153–166. [https://doi.org/10.1016/S1046-2023\(02\)00306-7](https://doi.org/10.1016/S1046-2023(02)00306-7)

1.7 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

1.7.1 Types of Contributions

Report Bugs

Report bugs at <https://github.com/tritemio/pycorrelate/issues>.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” and “help wanted” is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with “enhancement” and “help wanted” is open to whoever wants to implement it.

Write Documentation

Pycorrelate could always use more documentation, whether as part of the official Pycorrelate docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at <https://github.com/tritemio/pycorrelate/issues>.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

1.7.2 Get Started!

Ready to contribute? Here’s how to set up *pycorrelate* for local development.

1. Fork the *pycorrelate* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/pycorrelate.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv pycorrelate
$ cd pycorrelate/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass tests (not yet, see #3), and that notebooks runs without errors.
6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

1.7.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests (for now see #3).
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.5+. Check https://travis-ci.org/tritemio/pycorrelate/pull_requests and make sure that the tests pass for all supported Python versions.

1.7.4 Tips

To run a subset of tests (not yet, see #3):

```
$ py.test tests.test_pycorrelate
```

1.8 Credits

1.8.1 Development Lead

- Antonino Ingargiola <tritemio@gmail.com>

1.8.2 Contributors

None yet. Why not be the first?

1.9 History

1.9.1 0.2.1 (2017-11-15)

- Added normalization for FCS curves (see [pnormalize](#)).
- Added example notebook showing how to fit a simple FCS curve
- Renamed [ucorrelate](#) argument from *maxlags* to *maxlag*.
- Added [theory page](#) in the documentation, showing the exact formula used for CCF calculations.

1.9.2 0.1.0 (2017-07-23)

- First release on PyPI.

1.10 Indices and tables

- [genindex](#)
- [modindex](#)
- [search](#)

p

`pycorrelate.pycorrelate`, 17

M

make_loglags() (in module pycorrelate.pycorrelate), 4

P

pcorrelate (in module pycorrelate.pycorrelate), 5

pnormalize (in module pycorrelate.pycorrelate), 5

pycorrelate.pycorrelate (module), 4, 17

U

ucorrelate (in module pycorrelate.pycorrelate), 6