Fluorescence Correlation Data via MATLAB

Leland J. Jefferis*

July 1, 2013

1 Introduction

The paper [2] describes an algorithm for obtaining a correlation curve from *Asynchronous Single-Photon Counting* (ASPC) data. This document is a description of the various details which arose in the specific implementation of this method in MATLAB. In particular, there are parts in the description of the method in [2] which are either unclear or missing and here, I hope to fill in these gaps. My goal for this document is to fully equip any user of the MATLAB function "Correlate_ASPC" to modify the code to their needs. It is assumed, for the sake of this report, that the paper [2] has been read completely. I will also make reference to the paper [1], but only Section 3 of [1] is necessary for my discussion.

2 The Method

2.1 The correlation function

ASPC data appears simply as an increasing sequence of arrival times and due to the limited resolution of any particular photon detector, all arrival times will be integer multiples of a minimal detectable time δt. Then we construct the discreet intensity function

$$I(t) = \begin{cases} \frac{1}{\delta t} & \text{if a photon detected at time t} \\ 0 & \text{otherwise} \end{cases}$$

If the total time of the experiment is defined as T, then the total number of times for which $I(t)$ is defined is simply $T/\delta t$. Define the average of $I(t)$ as

$$\langle I(t) \rangle = \frac{\delta t}{T} \sum_t I(t) \tag{1}$$

where the sum is taken from the first arrival time to the final arrival time. As in [1], define

$$\delta I(t) = I(t) - \langle I(t) \rangle$$

so that the normalized autocorrelation function at a lag time τ (also an integer multiple of δt) is given by

$$G(\tau) = \frac{\langle \delta I(t) \cdot \delta I(t+\tau) \rangle}{\langle I(t) \rangle^2}. \tag{2}$$

A quick bit of algebra allows one to rewrite (2) as

$$G(\tau) = \frac{\langle I(t) \cdot I(t+\tau) \rangle}{\langle I(t) \rangle^2} - 1. \tag{3}$$

*University of Wisconsin - Madison, Department of Mathematics. Email: jefferis@math.wisc.edu

The numerator of the fraction in (3) may be written as

$$\langle I(t) \cdot I(t+\tau) \rangle = \frac{1}{T\delta t} \times \{number\ of\ collisions\}_\tau$$

where the *number of collisions* counts the number of times at which $I(t)$ and $I(t+\tau)$ are both non-zero. Note that the subscript τ in the number of collisions indicates that this quantity is dependent on the lag time τ. The denominator of the fraction in (3) may be written as

$$\langle I(t) \rangle^2 = \left(\frac{1}{T}\right)^2 \times \{number\ of\ data\ points\}^2$$

and *number of data points* is the number of data points in the file being analyzed. Thus finally, (2) becomes

$$G(\tau) = \frac{T}{\delta t} \times \frac{\{number\ of\ collisions\}_\tau}{\{number\ of\ data\ points\}^2} - 1. \tag{4}$$

This form for $G(\tau)$ makes clear that the number of collisions is the only term which varies with τ and is therefore the focus of the computation. Without time coarsening (introduced next) it is fairly straight forward to compute the number of collisions, but since this was discussed fully in [2] I will continue on.

2.2 Time coarsening

It is customary to display autocorrelation result on a log scale so it is desirable to choose a set of time lag values τ which are exponentially spaced. I chose to use the time lag values in the same way as [2] namely

$$\tau_j = \begin{cases} \delta t & \text{if } j = 1 \\ \tau_{j-1} + (\delta t) 2^{\lfloor (j-1)/B \rfloor} & \text{if } j > 1 \end{cases} \tag{5}$$

where we take $j = 1, ..., j_{\max}$ with $j_{\max} = n_{\text{casc}} B$. The introduced parameters B and n_{casc} are both integers and B may be thought of as the inverse exponential growth rate of the time lags τ_j and n_{casc} is the number of *cascades* of lag times. A *cascade* in this context is simply a set of lag times which are evenly spaced. Note also that in (5), $\lfloor \cdot \rfloor$ is the floor function.

In order to avoid a loss of resolution in our data when moving to increasingly larger spaced lag times, one employs a time coarsening step at the conclusion of each cascade of lag times. To do this, divide the set of arrival times by 2 and then round to the nearest lowest integer multiple of δt. Although not mentioned in [2] one must also do the same to the set of lag times $\{\tau_j\}$ since otherwise the computation would have two different and incompatible time scales present! This coarsening of the data will eventually create consecutive arrival times in the data which have become equal. Thus when counting collisions, as required in (4), these merged data points must count for more collisions than un-merged data points (because they represent several arrival times). This is dealt with in [2] by systematically eliminating duplicates and keeping track of an additional weight vector to account for these mergers. Since resizing a vector inside of a loop is highly inefficient in MATLAB, I have employed simple alternate method for keeping track of these mergers the details of which is discussed in a following section. I will end this section by mentioning that for each coarsening step, the final collision count must be scaled by whatever factor the data has been coarsened. In this case because of our chosen coarsening method, we must divide the resulting number of collisions at each lag time τ_j by a factor of $2^{\lfloor j/B \rfloor}$.

2.3 Computing number of collisions

With the previous discussion of time coarsening, I am prepared to discuss the computation of the *number of collisions* term appearing in (4). After some number of time coarsening steps, the data set of arrival times $\{t_1, t_2, ..., t_N\}$ as well as the shifted by τ_j arrival times $\{t'_1, t'_2, ..., t'_N\}$ may contain consecutive values which are equal. Note that contrary to [2] I am not removing arrival times which have become equal as per some time coarsening step. The algorithm proceeds as follows. First we progress through the arrival times until we find some $t_a \geq t'_1$. If $t_a = t'_1$, then we count how many subsequent t_i values (starting from t_a and

including t_a) are also equal to t_a and assign this value to w_1. We do the same for the subsequent t'_i values (starting from and including t'_1) and assign this value to w_2. Then the total collision count is increased by $w_1 \times w_2$ and the algorithm repeats using t_{a+w_1-1} and t'_{1+w_2} as the new starting values. The value of this approach is that it avoids keeping a separate weight vector as well as resizing vectors within a loop both of which have confirmed performance advantages in MATLAB. As a reminder, the total number of collisions should be divided by $2^{\lfloor j/B \rfloor}$ as discussed in the previous section.

3 Technical details

Already my method has been described in full, but there are a few implementation details which may be useful to future users and modifiers of the code.

3.1 Initial time coarsening step

In my implementation of the method, the parameter δt may be set manually. If one chooses to let δt be the actual smallest time resolution present in the ASPC data, then the code will do nothing to change the arrival data beyond the time coarsening. However, if δt is set to something larger, the code will perform an initial coarsening step on the data to transform the data to be integer multiples of this specified δt. The final result will be scaled accordingly, and this step serves as a starting point for the sequence of lag times. In the data sets I tested, it was not needed to use δt as small as the smallest time resolution so this feature was quite handy for producing usable results.

3.2 Time coarsening options

One may wish to choose time lags which are spaced in some different manner than chosen in this report. The current iteration of this code performs a time coarsening step each time the spacing between consecutive lag times. Although one is always free to choose whichever lag times they please, it is desirable that the coarsening exactly corresponds to the increased spacing of the chosen lag times. It is strictly required that if the ASPC data is coarsened by a factor of α (so that the arrival times are divided by a factor of α and then rounded), then the resulting number of collisions appearing in (4) must also be scaled by α (divided by α). A good rule of thumb then is that if the spacing between arrival times increases by a factor of α, then the ASPC data should be coarsened by a factor of α which then requires that the number of collisions be scaled by α also.

3.3 Rounding floats

There was one enduring error in my code during its development which was caused by how floating point numbers work in MATLAB. During the time coarsening step, I was performing a rounding step via the "floor" function and then doing some arithmetic on the rounded floating point number. In MATLAB, doing this may cause nonzero terms to appear in the 1e-14 place which can be devastating when one is trying to check if two arrival times are equal. To avoid this, [2] used the integer data type which they claimed was faster. However, in my experience with MATLAB, using integers which were large enough to deal with the size of the order of time resolution of the ASPC data was prohibitively slow. Thus my code still uses floats but I have taken care not to do arithmetic on values after they have been rounded.

3.4 Choice of returned lag times

There is a subtle issue resulting from the fact that lag times must be time coarsened in the same way the ASPC data is. One is faced with the question wether to return the originally generated lag times or the progressively time coarsened ones. It makes more sense to return the progressively time coarsened ones and this decision on my part appears to agree with the industry software so far as I can tell. My evidence is based on comparing results of both and noticing which agreed better.

3.5 Inline "if" statements

MATLAB has many quarks which cause it run very quickly in some cases and shocking slow in others. One of these which became apparent in my coding involved the break statements in my main loop. I have a few "if" statements which are of the form:

```
if (condition), break, end
```

The traditional way of writing this in MATLAB is

```
if (condition)
    break
end
```

which one would expect to be identical. However, the former is MATLABs notation for an "inline if" statement which is apparently much much faster since using the latter form will slow down code almost to a halt! I have no good understanding as to why this should be, but such is no less the case. In short, beware!

References

[1] Petra Schwille and Elke Haustein. Fluorescence correlation spectroscopy an introduction to its concepts and applications. *Spectroscopy*, 94 (3):1–33, 2009.

[2] Michael Wahl, Ingo Gregor, Matthias Patting, and Jörg Enderlein. Fast calculation of fluorescence correlation data with asynchronous time-correlated single-photon counting. *Optics Express*, 11 (26):3583–3591, 2003.