Photobleaching Step Finding Code ---------- Very Short User Guide

Introduction:

1) This code is written in Python 2.7. It was my training project as I was teaching myself Python, so if you are an expert Python programmer you will surely find a lot that is redundant, inefficient or that you could have done so much better. Feel free to make any modifications you like and if they do improve accuracy or performance please share with the community.
2) The code assumes you have taken data until all fluorophores photobleach and for some time thereafter, so there are long stretches of data points where a single fluorophore is active, and when none are active. These stretches are critical because they allow the Kalafut-Visscher part of this algorithm to give accurate statistics of the background and single fluorophore emission intensity mean and variance. If your data does not last to total photobleaching, you must somehow estimate the mean and variance of your background and single fluorophore emission intensities and enter them in the code by hand at the appropriate points. The code also assumes your data goes from maximum fluorophores active to all photobleached. Do NOT pre-reverse your data; the code will do so automatically.
3) In this version of the code runtime is approximately 1 sec/data point. Good parameter choices can cut this down to half, bad choices can double or triple it.
4) There are a number of parameters, of lesser or greater importance, that have to be set manually. All these are marked in the code annotation as either “MANUALLY SET” or “MANUALLY PICK”. Pending development of a GUI (indefinitely postponed at this point) you must enter in line your choices and suitable parameter values. Below I broadly describe each part of the code in no particular order, pointing out where you must (or should) enter values by hand, where you must make a choice, and what the significance of your choices would be. I also offer some minor advice based on my experience working on experimental data sets.

Main

This is of course the part where the most parameters are and the most choices must be made. First of all:
0) Line 44: Here is where you enter your data set file name. The data should be in a plain text file, in the form of a single column of numerical values, assumed to be fluorescence intensities.

1) stat_flag (line 38). This tells the code how to learn the dataset statistics. You can enter 0 to enter all four values (background and single fluorophore emission intensity means and variances) by hand, 1 to manually enter the mean and variance of the background and let the code find the ones of the single fluorophore, 2 to enter manually the mean and variance of the single fluorophore and let the code find those of the background, and 3 to let the code find everything.
2) dropORpad (line 39). This parameter is of use only when you data set length is not an integer multiple of the window size (windz) you chose. For example, say you have a dataset of 750 points but have chosen a window of 70 points. Since 750%70=50, the code will produce 10 windows of 70 points each and must do something with the remaining 50 points. Either they must be discarded, or another 20 points must be entered to create an 11th window. Choosing 0 here picks the first option, whereas choosing 1 the second. Choosing 1 copies data points from the end of the trace and attaches them to the trace end, effectively duplicating a stretch of the data. To continue the above example, picking 1 would copy the last 20 points of the data set and add them to it creating a data set of 770 points and thus 11 windows. Since the data set is not yet reversed at this point, it is always the time-wise latest points that are affected. Be VERY careful how you use this in conjunction with *Kvflag* below.
3) interruptflag (line 40). This set to 0 allows the code to proceed normally. Set to 1 it interrupts the code after Step 2 and outputs Step 2 results plus a copy of the data the code was working on. The purpose of this is to allow you to see if your choice of window (windz) was the right one.
4) fluorupdateflag (line 41). A value of 2 here means that as each window (bar the first) is processed by Step 3, the code will take the initial fluorophore number from the results of Step2. A value of 3 means the code will take the initial fluorophore number from the estimate of fluorophores Step 3 gave in the previous window.
5) priorsliceflag (line 42). Set this to 0 if you want the code to find a λ effective automatically or to 1 if you want to enter a value manually.
6) KVflag (line 43). If you set this to 0 the code will feed into the KV algorithm all your data. If your data is large (>1000 data points), this may cause the code to become very slow (see *Known Issues*). To avoid such a problem, set this to something other than 0, plot your (not reversed!) data, eyeball the index of where the last step is (where the last fluorophore photobleached) and second to the last step is. Enter the index for the last step into the value for the variable *byhand1*. Enter a value into *byhand2;* this should be less than *byhand1* and should be large enough that it encompasses the value of the second to the last step (See example in *Known Issues)*. The code will then feed into the KV algorithm a slice of your data starting at index *byhand1-byhand2* and ending at index *byhand1+byhand2* or the end of the dataset, whichever is smaller.
7) windz (line 47). Easily the most important parameter, it sets the size of the window for Steps 2 & 3. **Under no circumstances should you ever use a window size larger than 200 data points**. (See Known Issues). Windows too small bring in small number statistics and the whole thing breaks down. Correct window choice is crucial for algorithm accuracy and speed: if the code runs too slow the most likely cause is a wrong window choice.

LeffFinder

There are no parameters to set or choices to make in these two mini-classes. Their job is to find a set of crude local approximations of the effective λ, the time-dependent Poisson rate with which photobleaching events occur. The code just sums the set of local rates and finds a very crude average of them. The point is that the effective λ plays only a very minor role in our criterion so a halfway decent estimate is still good enough.

Outputer

This is just a class meant to output the data in the desired format. The only choices to be made here are in the lines 25-27, where you need to pick the names of the files your outputed data will be stored in.

Kalafut

This runs the Kalafut-Visscher (KV) algorithm on the data. Its sole purpose is to find the location of the first step (remember the data trace is reversed so this is the final step time-wise) and calculate the mean and variance of the background and single fluorophore intensities. If you have these values from elsewhere you still need to run Kalafut to get the location of the step that corresponds to the photobleaching of the last fluorophore so the effective λ can be calculated.
There are two numbers you can set by hand here:
1) threshold (line 31). This sets the value below which a step is not accepted as such, on the grounds that the change in the mean fluorescence levels is too small. Too high a value for this and steps will be missed, too low and there will be spurious steps. However, we only care about the very first step so the code is robust to a considerable range of values for this parameter.
2) stat_window (line 86). This sets the size of the data set stretch that wil be used to calculate the mean and variance of the single fluorophore emission intensity. A larger window makes for a more accurate estimate, but if the window is so large that the second step is reached the values will be inaccurate. It is assumed that you have at least 50 data points between the bleaching of the second-to-the-last and last fluorophores.

Seeker

-*mSICer*: This class runs our criterion (our Step 3) on a data window. Important parameters here are:
1) gamma (line 26). Sets the cutoff γ_0 which constrains number of events to number of steps. You can experiment with its value to try to find what gives you better fits. I have been asked to place this value choice in the *Main*, but I am not doing it because I found when using this code that keeping it in *mSICer* prevented me from changing it until *after* I had played around with *windz* enough to have a good window size. Changing *gamma* much before you have a good window size may yield bad results.

2) limit (line 103). In this line you set a limit for how many steps past 2 the code will consider may exist within the window. The code always assumes there are at least two steps in every window and tries to find them. If there are more than two, the code will consider as many as the value you pick in this line, minus 1.
3) syms (line 202). These determine the maximum range of events we may have. For example, '6' means it is possible to have a step where as many as three fluorophores simultaneously reactivate. Eliminating one of these values means that a certain step size will no longer be considered, and will probably throw the code completely off. If you want to consider steps sizes greater than three minus or plus, you need to change the entire *dalistor* function. Because this function is by far the most computationally intensive and time-consuming step of the code, tampering with it may cause the code to run very slow.
-*Slicer:* This class runs Step 2 of our algorithm that is meant to limit the number of steps considered by Step 3. The only important parameter is:
1) evros (line 245). This determines how many distributions the class will consider. This is computationally cheap, so large numbers are not too much trouble. Of course picking a number greater than the maximum number of fluorophores you could have is just wasting time.

Suggestions:

The problem of always working with synthetic data, as I did when creating and benchmarking this code, is that you always know the optimal parameter choices and can choose values close to them to get very good results in minimum time. This is obviously not the case when working with experimental data. Choosing parameters at random until you hit on a choice that will give good results will make each instance of the code run slow and it will be hours or days before you have something decent to show for your labor. This is particularly so because the code runs much slower (lots more local searches) if the parameters are not optimal. The parameters that have the most impact on run time are *limit, windz* and *gamma.* The first affects run time in a straightforward way: the larger it is, the slower the code. The other two are much more unpredictable: the further away from their optimal values you are the slower the code – perhaps as much as one or two orders of magnitude slower. For this reason I suggest this tactic: pick a data set that is representative of the group of data sets you want to run. Get the statistics from the KV algorithm, then choose a value for *windz,* set *interruptflag* to 1, run the code and inspect the results. Typically, windows that are too big will be obvious, as you will see steps occurring in the middle of very regular stretches of data. Likewise, windows that are too small will give your results a "brush"-like appearance, with many ups-and-downs. Experiment with the value of *windz,* until you find one that looks good to you by eye – only then set *interruptflag* to 0 and run the full code. This helped me a lot.

Known Issues

1) Sometimes, for large data sets (over 1,000 data points) and more frequently if the signal-to-noise ratio is low, the code becomes extremely slow because the KV algorithm crawls along and takes "days" to finish. This is because KV gets confused by the high noise at the start of a data set and tries to place a step at each and every data point, creating numerical instabilities. If this happens it can be dealt with as follows: Plot your data and find the data set areas where the steps corresponding to when the last and second to the last fluorophores photobleached – these will almost certainly be easy to find by eye. Use the instructions in *KVflag* above on what to do then. For example, say you have 10,000 data points. By eye, it looks like the last step occurred around index 9800, and the second to the last around 8900. Set *byhand1* to 9800 and *byhand2* to 1200. Then KV will take a slice of your data set from 8600 to 10000 (it would be 11000, but as you only have 10,000 data points you cannot go above 10,000). In this 1400 data point set, say KV finds the statistics and decides the precise location of the last step is at 1140. 1140 in the 8600 to 10000 slice corresponds to 9740 in the real data set: this is your *tzerom*. Input the statistics by hand into line 73 of the main, change the value of *tzerom* to what you just found instead of "Try0.tzero" in line 75, comment out line 74, and run the code.
2) Never set your window size (*windz*) to anything greater than 200 data points. The function *multichoose* is not capable of treating windows larger than 200 data points in anything less than an inordinate amount of time, plus it requires huge amounts of memory to do so. If you pick a window size >200, the code will run very slow and may even stall altogether.
3) Because the code at Step 3 treats windows sequentially and largely independently, small number statistics can cause spurious steps to appear towards the end of a window. The best way to differentiate between spurious and real steps is to see if the steps persist when using window sizes of different (but close) sizes. If a step appears when you set your window size to, say, 80 but is no longer there if you change the window size to, say, 100, chances are it is spurious. Unfortunately this method does not work at the very start of the trace - so I strongly suggest you ignore as spurious any steps you find closer than 20 data points from your trace start (the earliest points time-wise, latest in the reversed data set).
We will add functionality in later versions of the code to automatically correct for spurious steps due to small number statistics.

Happy Step Finding!