Tutorial document written by Vincent Pelletier and Maria Kilfoil 2007.
Adapted for Python in July 2013 by Kevin Smith and Maria Kilfoil.

Overview

This code finds and tracks round features (usually microscopic beads as viewed in a microscope) and outputs the results in a convenient fashion for further analysis. The code is based on IDL code by John Crocker.

The features are first localized by matching finding high intensity regions on the image convolved with a disk. Then false features are weeded out based on their shape and intensity. Finally, the features are linked into trajectories by minimizing the total displacement of individual features between successive images. The algorithm can be made to allow features to skip one or more frames, with the trajectory continuing when they reappear.

By using the information of all the pixels of a round feature, its center can be determined to much better than pixel resolution. We routinely attain accuracy close to 5nm (about 0.03 pixel) for 1μm beads imaged at 0.16μm/pixel.

Code philosophy

All the code is concentrated in a single folder for ease of use and maintenance.

The functions expect data to be structured so that each experiment has one root folder, which contains the microscopy images in subfolders called "`fov#`". The image files are expected to be named "`fov#_####.tif`". The time information for the frames are expected to be found as a vector matrix called "`time`" saved as a NumPy file called "`fov#_times.mat`", located in the subfolder with the images. The feature finding data will be output in different subfolders to the experiment root.

It is convenient to create a variable containing the path to the root of the experiment. This variable will be referred to as `basepath` below, and should be a path, i.e. ending with '\\' (in Enthought, on a Windows system. This may vary for other IDEs).
Example:

```
basepath='C:\\myexperiments\\expt1\\';
```

There are rare instances where some quantities relevant to the analysis are hard-coded, but once these are set for a given experimental system, one should not need to modify them further. Any parameter likely to change from one experiment to another is passed as a function parameter.

The code assumes 2D image data, though the tracking algorithm is fully compatible with 3D spatial data.

For clarity, many of the functions exist within a module of the same name. For example, the feature2D function can be found within the function2D module, and can be called with feature2D.feature2D. This can be clumsy, so instead of importing the module feature2D, it is sometimes more elegant to import the feature2D. To do this, replace "import feature2D" with "from feature2D import feature2D". The function can then be called with "feature2D".

External Dependencies

In order to use the 2D feature finding/tracking codes, the following python packages and modules must be installed:
Numpy
Scipy
Matplotlib
PIL
tiff_file (included on the website)

NOTE: All of these packages, with the exception of tiff_file, can be downloaded from the Enthought Canopy Package Manager. If you are not using Enthought, these packages can be downloaded from the Python support site.

Feature finding

The feature finding code finds high intensity matches in the image as candidate features, none within a given distance from another candidate. At each of these local maxima, it computes the center of mass of the image intensity under a pixelated disk-shaped mask roughly the size of the features to be found and centered at the initial position, shifts the mask center to this new position in such a way that it is no longer in registry with the underlying image pixels, and re-computes the center of mass of the image intensity underneath the mask. This new center of mass of the intensity is the refined feature position. As it computes the center of mass, the code also computes other useful characteristics of the feature: its total intensity, its eccentricity and its radius of gyration squared.

The integrated intensity is simply the sum of the intensity value of all pixels under the mask. The eccentricity is a measure of how elliptical the feature is: an eccentricity of 0 is a perfectly circular disk, whereas a large eccentricity is a very elongated feature. The radius of gyration is a measure of the size of the feature, computed by averaging the square of the distance to the center position, weighted by pixel intensity. All three are useful to discriminate false features from real ones. Essentially, a bead tends to be circular, bright and of an expected size, whereas noise tends to be elliptical, less intense and/or much more extended.

The heart of the feature finding is in the function `feature2D`; the rest is a convenient front end.

The first step of the feature finding consists of finding the right rejection parameters to optimize the number of false features rejected and real features and accepted. To determine these, first call `mpretrack.test`, which is an interactive function within the `mpretrack` module that displays the result of a set of chosen rejection parameters on a single frame:

```
mpretrack.test(basepath, fovn, frame, numframes,
featuresize, masscut=0, Imin=0, barI=None, barRg=None,
barCc=None, IdivRg=None, field=2 );
```

Here, `fovn` is the batch number to process (for example, acquired from one field of view), `featuresize` is the radius of the features you are attempting to find (in pixel, integer); `barI` is the minimum integrated intensity that will be accepted; `barrRg` is the maximum radius of gyration squared that will be accepted (in pixel squared, float); `barCc` is the maximum eccentricity that will be accepted; `IdivRg` is the minimum ratio of integrated intensity to radius of gyration squared to be accepted; `Imin` is the minimum intensity of local maximum to be considered (larger values will reduce the number of candidate locations, useful in noisy images, set to 0 to use the default "top 30%" selection); and `masscut` is a parameter which defines a threshold for integrated intensity of features before position refinement, to speed up the code. These seven parameters are optimized interactively with this front-end. `fovn` is a number identifying the series of image files you are analyzing; `frame` is the frame number in which you are optimizing the parameters; and `field` is a parameter which is set to 0 or 1 if the image is only one field of an interlaced video frame, and 2 if it is the full frame. If you don't know what your camera outputs, find out.

The function `mpretrack.test` displays row-wise in the command window first the characteristics of all the accepted features, then of all the rejected features. All the features located are also stored in the output variable `M2` and the accepted features in `MT`. The data is organized as follows: columns 1 and 2 are the *x* and *y* positions (in pixels); column 3 is the integrated intensity; column 4 is the radius of gyration squared (pixels squared); and column 5 is the eccentricity. The data is also represented in an automatically-generated figure containing the micrograph superimposed with red dots representing rejected features and green circles representing accepted features.

Strategies to find the right parameters for the rejection step include looking at what features are accepted and rejected on the figure, and for accepted features, looking at the corresponding intensity, radius of gyration and eccentricity to gain intuition for what constitute good parameters. Note that the feature size parameter should be about the same size in pixels as the feature itself in the image. Another potentially useful strategy is to plot the characteristics one against another, for example eccentricity vs. radius of gyration, to look for clustering in parameter space corresponding to the real features. Often noise "features" will constitute a clear, distinct population from the real features.

Eventually you could update `mpretrack.test` and `mpretrack.run` to include more intricate means of distinguishing real features from false ones in your system, depending on your application.

Once the parameters do a satisfactory job at keeping real features, they can be fed into the actual feature finding loop function, `mpretrack.run`:

```
import mpretrack
for run=1:N
  mpretrack.run( basepath, fovn, numframes, featuresize,
masscut=0, Imin=0, barI=None, barRg=None, barCc=None,
IdivRg=None, field=2 );
end
```

where the parameters `featuresize`, `masscut`, `Imin`, `barCc`, `barRg`, `barI` and `IdivRg` are the same values determined by using `mpretrack.test`. `fovn` is again the batch number to and `numframes` is the number of frames which should be processed for that batch (from frame 1 to `numframes`). `field` is a parameter which is set to 0 or 1 if the image is only one field of an interlaced video frame, and 2 if it is the full frame.

The output is a file called `[basepath + '\\fov#\\MT_featsize_#.npy']` where the first # corresponds to `fovn` and the second one is the feature size (one might be interested in running the feature finding algorithm more than once per image if there are two or more distinct feature sizes to be found). That file contains a single matrix, `MT`, containing the information of all the features found (and not rejected) in that batch, arranged as follows: columns 0 and 1 are the *x* and *y* positions in pixels; column 2 is the integrated intensity; column 3 is the radius of gyration squared (pixel squared); column 4 is the eccentricity; column 5 is the frame number in which the feature was found; and column 6 is the time at which the image was recorded, as found in the `fov#_times.npy` file.

NOTES:
One parameter to be used in the `feature2D` function call that is hard-coded in `mpretrack.test` and `mpretrack.run` is the length scale of the noise in pixels, called `lnoise` (1 works great, any positive floating value is acceptable). You do not need to change the noise length scale parameter, though one can easily add it as an argument to the front ends if desired.

Module Dependencies:

```
mpretrack
    feature2D
        bpass
        localmax
```

```
    create_mask
rsqd
thetarr
fracshift
fieldof
```

Tracking

The tracking code links the positions found in successive frames into trajectories. The algorithm tries to minimize the sum of the distances between features in two successive frames. A maximum displacement limits the matching to only those features closer than what the user knows to be extremely unlikely distances the feature may travel between two frames. Any feature with no match in the successive frame will be assigned a distance of this maximum allowable displacement. The algorithm can be made to remember features over more than immediately successive frames, so that if a feature disappears momentarily (out of the focal plane, typically), it can be matched to its previous trajectory segment when it reappears.

The heart of the tracking is in the function `trackmem`; the rest is a front-end for convenient data loading and storage. A typical way to run the front-end is:

```
import fancytrack
for j=1:numFOV
  fancytrack.run( basepath, j, featsize, maxdisp,
goodenough, memory );
end
```

where the index `j` corresponds to the batch number to be tracked and `featuresize` is the feature size used in the feature finding: both are used to identify the file containing the actual feature data to be tracked, in the output format described above. `maxdisp` is the maximum displacement (in pixels) a feature may undergo between successive frames; `goodenough` is the minimum length requirement for a trajectory to be retained; and `memory` is how many consecutive frames a feature is allowed to skip. `maxdisp`, `goodenough` and `memory` are optional. Their default values are 2, 100 and 1 respectively, meaning that a feature is allowed to skip only one consecutive frame any number of times so long as the total trajectory has at least 100 time points, and that no features in different time frames will be matched into the same trajectory if they are greater than 2 pixels apart.

The dimensionality of the data is hard-coded in `fancytrack`, set to be 2. `trackmem` itself should be able to handle data of any dimension, though we have tested and used it only for data of 2 and 3 spatial dimensions.

The output of `fancytrack` is a file for each field of view called `Tracks_featsize_#.npy`, located in the `[basepath + '\\fov#']` folder.

This file contains a single matrix, `tracks`, similar to the MT matrix except for an added trajectory ID# column by which the data is sorted: columns 1 and 2 are the *x* and *y* positions (in pixels); column 3 is the integrated intensity; column 4 is the radius of gyration squared (pixel squared); column 5 is the eccentricity; column 6 is the frame number in which the feature was found; column 7 is the time at which the image was recorded; and column 8 is the trajectory ID number.

Module Dependencies:

```
fancytrack
    trackmem
        unq
        luberize
```