ImageJ macros code description

PreProcessMovie.ijm

The ImageJ macro used to stabilize movies (section 3.1) is fully described in this section.

```
// Dialog box
Dialog.create("Apply linear XY shift");
Dialog.addNumber("X shift (pix/frame)", -10);
Dialog.addNumber("Y shift (pix/frame)", 0);
Dialog.show();
XShift = Dialog.getNumber();
YShift      = Dialog.getNumber();

// Duplicate image
NewName = "Corrected_X"+d2s(XShift,0)+"_Y"+d2s(YShift,0)+"_"+getTitle();
run("Duplicate...", "title=["+NewName+"] duplicate");

// Shift images
Stack.getDimensions(width, height, channels, slices, frames);
for(f=0;f<frames;f++)
{
   for(z=1;z<=slices;z++)
   {
      Stack.setPosition(1, z, f+1);
       run("Translate...", "x="+d2s(XShift*f,0)+" y="+d2s(YShift*f,0)+"  interpolation=None slice");
   }
}

// Reset display
Stack.setPosition(1,1,1);
```

Full transcript of ImageJ macro PreProcessMovie.ijm

The first section of the code displays a dialog box from which the X / Y shifts (per time frame) to apply to the images are collected; these values are stored as variables ***XShift*** and ***YShift***. The default setting is -10 pix/frame along X dimension, the value most commonly used for the sample time-lapse images which were used when designing the macro.

Next, the image is copied and renamed by appending a text string with the X/Y correction as reference; the original image name is retrieved by the ImageJ macro function **getTitle()**.
The functional section of the code is the double loop over time frames and Z slices coming next: for each time frame, all Z slices are translated by calling ImageJ **Translate...** Since we plan to apply a linear correction to the images, the translation is simply computed by multiplying ***XShift*** and ***YShift*** by the frame index. The bounds of the loops (frames and Z slices) are set by first calling IJ macro function **Stack.getDimensions()** to retrieve the dimensions of the hypertstack. Finally, the hyperstack sliders are set to the first image.

ReviewMeasureTracks_simple.ijm ImageJ macro

The code of the macro **ReviewMeasureTracks.ijm** that was used in section 3.3 is a bit lengthy and technical to describe. To focus on the key concepts, we provide a simplified version in the software repository. This simplified version only implements a single starting bounding box selection and drops some visual features; it is fully described in this section.

Initialization

The first four sections of the macro initialize the workspace (closing windows, initializing variables, dialog box), we skip the details.

```
// Retrieve calibration and image dimensions
// Initialize windows
// Initialize variables
// Dialog box
```

Tracks selection: starting bounding box

The next section implements a user interaction and retrieves the position and size of a user drawn bounding box delimiting the starting positions of selected cells (tracks starting in first frame).
Upper left X and Y coordinates, width and height of the box are retrieved by the ImageJ macro function **getSelectionBounds()**, and stored as four variables.

```
// User defined track start bounding box
setTool("rectangle");
waitForUser("Draw a bounding box for starting positions");
getSelectionBounds(Bx, By, Bwidth, Bheight);
```

Read tracks from Trackmate table

For each track, we need to retrieve the nucleus chronological positions to compute its quantitative measurements (mean speed and directional persistence).

From inspection from the results table **Spots in tracks statistics** (Trackmate output) the tracks are identified by the column **TRACK_ID** (chronological order); they can hence be delimited by wandering the table row by row and monitoring **TRACK_ID** changes.

In the code, a loop over the rows of the results table implements this procedure: when a change is detected a sequence of operations is performed, if not the loop goes on and updates the variable **CurrentTRkLgth**, an incremental counter keeping track of the current track length. The track IDs are read from the results table by the ImageJ macro function **getResult(**"**Column header**", **row)**.

```
// Analyze tracks by sequentially reading track IDs
CurrentTrkLgth = 0;
CurrentTrackID = getResult("TRACK_ID", 0);
for(i=0;i<nResults;i++)
{
        TrkID = getResult("TRACK_ID", i);
      if(TrkID != CurrentTrackID)
      {
         .... Do something  ....
          CurrentTrkLgth = 0;
          CurrentTrackID = TrkID;
       }
       CurrentTrkLgth++;
}
```

Note: **CurrentTrackID** is initialized with the first track ID (first row in results table) before the loop starts. Also, a row is appended to the results table with a high track ID to ensure a change for the last track; this part of the code is not reproduced here.

Re-order nucleus time indexes from the track

For a given track, the frames are unfortunately not ordered chronologically in the results table. Once a track has been identified the positions hence have to be re-ordered before computing quantitative measurements.

From code inspection, the variable **StartRow** holds the results table row at which the current track starts. For each track, the following code section reads spot frames and stores them to an array **TPos**. Then we use the ImageJ macro function **Array.rankPositions** to compute **IndxTPos**, the index array re-ordering **TPos**. For instance, if **TPos** = [45 43 44] → **IndxTPos** = [1 2 0].

It is then easy to access the rows of the results table in chronological order by indexing with **IndxTPos[i]** for i = 0, 1, 2. This trick is used in next section.

```
// Re-order track position by ascending time frames
TPos = newArray(CurrentTrkLgth);
for(j=0;j<CurrentTrkLgth;j++)TPos[j]= getResult("FRAME",StartRow+j);
IndxTPos = Array.rankPositions(TPos);
```

Re-order spots chronologically in current track and cancel XY correction

The current track is processed only if it starts in first frame, that is if **TPos**[**IndxTPos**[0]] = 0 (first line of the following code section). If this condition is checked, the arrays **XPos**, **YPos**, **ZPos** and **TPos** are initialized to current track length. Then, spatial and temporal spot positions are read from results table in chronological order (trick from previous section).

Finally, XY shifts correction is subtracted to the spatial positions of the spots to restore their actual location. The pixel calibration (stored in **vxw** and **vhx**) is used to convert from pixel to physical units.

```
// Fill position arrays (re-order time points + correct motion)
if(TPos[IndxTPos[0]]==0)
{
  XPos = newArray(CurrentTrkLgth);
  for(j=0;j<CurrentTrkLgth;j++)
  {
     TPos[j] = getResult( "FRAME", StartRow + IndxTPos[j] );
     XPos[j] = getResult( "POSITION_X", StartRow + IndxTPos[j] ) - ShiftX * TPos[j] * vxw;
     ...
  }
  ...
}
```

Select valid tracks

This code section tests if the first track spot (**XPos[0]**, **YPos[0]**) is located inside the user defined bounding box. This is achieved by performing 4 tests that are combined two by two by logical AND operations (&&). As before, pixel calibration is used to convert from pixel to physical units.

```
// Only process track if first position inside starting box
if( ( XPos[0] >= Bx*vxw ) && ( XPos[0] <= (Bx+Bwidth)*vxw ) )
{
  if( ( YPos[0] >= By*vxh ) && ( YPos[0] <= (By+Bheight)*vxh ) )
  {
    ... Do something ...
  }
}
```

Store track spots to ROI Manager

Track spot positions are stored as point selection in the ROI Manager for their further inspection. Prior to this, the ROI Manager window is closed if opened (not reproduced here).

```
// Add track points
for(j=0;j<CurrentTrkLgth;j++)
{
  Stack.setPosition(1, 1+round(ZPos[j]/vxd), 1+TPos[0]+j);
  makePoint( round(XPos[j]/vxw), round(YPos[j]/vxh) );
  roiManager("add");
}
```

Measurements and statistics

If the user decides to keep the current track, its length is estimated by summing the lengths of its constitutive segments (Pythagorean theorem). Then, the track displacement is computed in the same way but by only considering first and last locations. Finally, mean speed and directional persistence are computed from the previous measurements and stored to a user defined table (code not reproduced here).

```
// Compute statistics
Lgth = 0;
for(j=1;j<CurrentTrkLgth;j++)
{
   Lgth = Lgth + sqrt(pow(XPos[j] - XPos[j-1],2) + pow(YPos[j] - YPos[j-1],2) + ...
}
Disp = sqrt(pow(XPos[CurrentTrkLgth-1] - XPos[0], 2) + ... ) ;
MeanSpeed = Lgth/(TimeStep*(CurrentTrkLgth-1));
Persistence = Disp/Lgth;
```

It is fairly simple to plug in any quantitative measurement you are interested into this code section, and then display the result to the user defined table.