References

The following work wouldn’t have been possible without relying on the foundational work of Jean-Yves Tinevez (https://github.com/tinevez/msdanalyzer, GitHub), who wrote the MATLAB class @msdanalyzer already used in a previous study (Tarantino *et al.*, 2014) and well explained in details in a chapter of the book Bioimage Data Analysis Workflows (Miura & Sladoje, 2020).

- Jean-Yves Tinevez (2021). Mean square displacement analysis of particles trajectories (https://github.com/tinevez/msdanalyzer), GitHub. Retrieved July 27, 2021

- Nadine Tarantino, Jean-Yves Tinevez, Elizabeth Faris Crowell, Bertrand Boisson, Ricardo Henriques, Musa Mhlanga, Fabrice Agou, Alain Israël, and Emmanuel Laplantine. TNF and IL-1 exhibit distinct ubiquitin requirements for inducing NEMO-IKK supramolecular structures. J Cell Biol (2014) vol. 204 (2) pp. 231-45

- Miura, K. and Sladoje, N., 2020. *Bioimage Data Analysis Workflows* (p. 170). Springer Nature. DOI 10.1007/978-3-030-22386-1

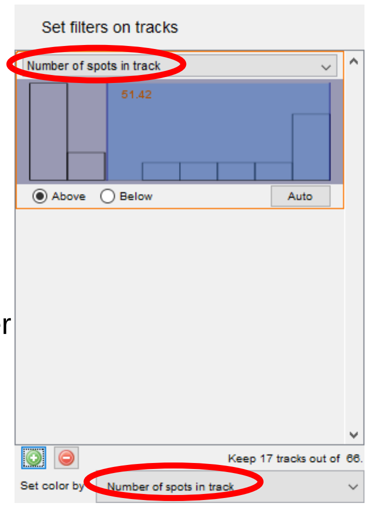
Trackmate_Parameters used on GA fibrils

1) Open video
2) Separate Green and Red channels (we're only interested in the Red)
3) Finally, open TrackMate:

- **LoG detector:** estimated blob diameter=0.4-**0.5** um; threshold=300-**400**; no median filter; do sub-pixel localization
  You can put a filter "Signal to noise ratio" for eliminating the formation of double spots on a single particle. Also the "Quality" filter is good.
- **Simple LAP tracker:** Linking max distance=**1**-1.5 um; Gap-closing max distance=1-**3** um; Gap-closing max frame gap=3-**5**

  Just to remind what are these parameters:
  - *Linking max distance* = the maximal distance (um) to link two particles from one frame to the next
  - *Gap-closing max distance* = for a trajectory, it is the maximal distance (um) to bridge over missing detections
  - *Gap-closing max frame gap* = for a trajectory, the maximal frame gap (=number of consecutive frames) to bridge over missing detections

- A good strategy is now to **filter tracks based on the number of spots they contain**. It has a side benefit: the MSD analysis we will perform later requires the tracks to be long for accuracy. Since the histogram for the number of spots in tracks have a large peak at low values that precede a gap at ***N = 10 Minimum!***, we can use this value as a threshold. Long trajectories are better for MSD. And if you use less than 10 you'll have errors when you do log-log fit later on; as in this thread: https://forum.image.sc/t/error-in-generating-and-reading-trackmate-xml-file/27385/21



- Go until the end of TrackMate algorithm and "select an action" → Export tracks to XML → Save it as "name**.xml**"
- Join the content of the three XML files together in order to be able to upload all the tracks in MATLAB.

STEPS (MATLAB R2018b)

Have everything ready to start

Open Matlab and locate your Desktop folder by typing on the "Command window":

```
>> cd /Users/mdp18pm/Desktop
```

Right click< *give access to folders and subfolders*

Make sure that the following files are in the **same folder** (and that this folder is added to your path and is on the Desktop):

- Tracks.XML file (this is the file that contains the information of both "xy trajectories" and the "time"; derived from TrackMate analysis). Note: make sure this file name terminates with ".xml"
- Matlab script namely "importTrackMateTracks" (this is a prewritten matlab script to allow reading of the Tracks.XML file into Matlab)

Also we're going to use the @msdanalyzer, which is a MATLAB class. So you have to put the @msdanalyzer folder in the MATLAB path. This class contains all the script to perform the MSD analysis.
To add it to the path: On the Home tab, in the Environment section, click Set Path. The Set Path dialog box appears. Add the folder location of the file @msdanalyzer. This can also be achieved by:

```
>> addpath("C:/Users/mdp18pm/Desktop/MSDtime Analysis/tinevez-msdanalyzer-da0bdaa/")
```

Read your tracks

```
>> track_file = '/Users/mdp18pm/Desktop/MSDtime Analysis/ProximalRegionSoma_fibrils/video1.xml'
>> tracks1 = importTrackMateTracks(track_file, true, false)
>> track_file2 = '/Users/mdp18pm/Desktop/MSDtime Analysis/ProximalRegionSoma_fibrils/video2.xml'
>> tracks2 = importTrackMateTracks(track_file, true, false)
>> track_file3 = '/Users/mdp18pm/Desktop/MSDtime Analysis/ProximalRegionSoma_fibrils/video3.xml'
>> tracks3 = importTrackMateTracks(track_file, true, false)
```

The first flag true is used to specify that we do not need to import the Z coordinates, and the second flag false is used to specify that we want a time interval in integer units of frames.

Import all tracks:

```
>> track_file1 = '/Users/mdp18pm/Desktop/MSDtime Analysis/ALL MERGED_ALL TRACKS/all tracks/image1.xml'
>> tracks1 = importTrackMateTracks(track_file1, true, false)
>> track_file2 = '/Users/mdp18pm/Desktop/MSDtime Analysis/ALL MERGED_ALL TRACKS/all tracks/image2.xml'
>> tracks2 = importTrackMateTracks(track_file2, true, false)
>> track_file3 = '/Users/mdp18pm/Desktop/MSDtime Analysis/ALL MERGED_ALL TRACKS/all tracks/image3.xml'
>> tracks3 = importTrackMateTracks(track_file3, true, false)
>> track_file4 = '/Users/mdp18pm/Desktop/MSDtime Analysis/ALL MERGED_ALL TRACKS/all tracks/image4.xml'
>> tracks4 = importTrackMateTracks(track_file4, true, false)
>> track_file5 = '/Users/mdp18pm/Desktop/MSDtime Analysis/ALL MERGED_ALL TRACKS/all tracks/image5.xml'
>> tracks5 = importTrackMateTracks(track_file5, true, false)
>> track_file6 = '/Users/mdp18pm/Desktop/MSDtime Analysis/ALL MERGED_ALL TRACKS/all tracks/image6.xml'
>> tracks6 = importTrackMateTracks(track_file6, true, false)
>> track_file7 = '/Users/mdp18pm/Desktop/MSDtime Analysis/ALL MERGED_ALL TRACKS/all tracks/image7.xml'
>> tracks7 = importTrackMateTracks(track_file7, true, false)
>> track_file8 = '/Users/mdp18pm/Desktop/MSDtime Analysis/ALL MERGED_ALL TRACKS/all tracks/image8.xml'
>> tracks8 = importTrackMateTracks(track_file8, true, false)
>> track_file9 = '/Users/mdp18pm/Desktop/MSDtime Analysis/ALL MERGED_ALL TRACKS/all tracks/image9.xml'
>> tracks9 = importTrackMateTracks(track_file9, true, false)
>> track_file10 = '/Users/mdp18pm/Desktop/MSDtime Analysis/ALL MERGED_ALL TRACKS/all tracks/image10.xml'
>> tracks10 = importTrackMateTracks(track_file10, true, false)
>> track_file11 = '/Users/mdp18pm/Desktop/MSDtime Analysis/ALL MERGED_ALL TRACKS/all tracks/image11.xml'
>> tracks11 = importTrackMateTracks(track_file11, true, false)
>> track_file12 = '/Users/mdp18pm/Desktop/MSDtime Analysis/ALL MERGED_ALL TRACKS/all tracks/image12.xml'
>> tracks12 = importTrackMateTracks(track_file12, true, false)
```

You can import more files, assign them into variables and then concatenate them orizzontally:
Example of the final **concatenation**:

```
>> distal_axon = cat(1, tracks, tracks2, tracks3)     %1 means I want to concatenate orizzontally
```

Concatenate all tracks:

```
>> all_tracks = cat(1, tracks1, tracks2, tracks3, tracks4, tracks5, tracks6, tracks7, tracks8,
tracks9, tracks10, tracks11, tracks12)
```

We can create this next variable to look into the content of one of the arrays (each array is a N x 3 array, because it contains information about x coordinate, y coordinate and time):

```
>> tracks_explore = tracks{1,1}
```

Let's now visualize one trajectory (tracks1):

```
>> x = tracks(:,2)

>> y = tracks(:,3)

>> plot(x,y, 'ko-', 'MarkerFaceColor', 'w'), axis equal, box off
```

Feed the data to the @MSD Analyzer

We need to create an empty analyzer first, as a container for the data that we'll put inside later on:

```
>> ma = msdanalyzer(2, 'um', 'frames')
```

Now ma is an empty msdanalyzer object, set to operate for 2D data (this is the meaning of the '2' as first argument), using µm as spatial units ('um' because MATLAB does not handle UTF8 characters very well) and frames as time units.

This object is empty, and we have to feed it the tracks with the ***addAll( )* method**. Luckily for us, as you can read in the help of the addAll method, it expects the tracks to be formatted exactly in the shape we have. So we can run directly:

```
>> ma = ma.addAll(distal_axon)

>> ma = ma.addAll(all_tracks)

>> ma = ma.addAll(only_axon)
```

Plot a preliminary "track overview" with the @MSD Analyzer

```
>> ma.plotTracks        % Plot the tracks
>> ma.labelPlotTracks        % Add labels to the axis
>> set(gca, 'YDir', 'reverse')
>> set(gca, 'Color', [0.5 0.5 0.5])
>> set(gcf, 'Color', [0.5 0.5 0.5])
```

Compute all the MSDs with the @MSD Analyzer

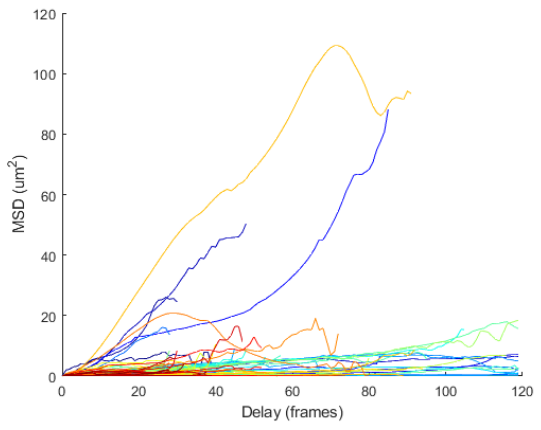
The **@msdanalyzer** class automates the calculation. Using the object we prepared in the previous step, calculating MSD is as simple as:

```
>> ma = ma.computeMSD
```

Notice that now the field of the **object "ma"** has some content. However interpreting it is not trivial. The plot of the individual MSD curves look like this if we type:

```
>> ma.plotMSD
```

#if the graph comes out with the Y axis non correctly oriented, just close the image of this graph. And then re-type the command **ma.plotMSD** → now it should work and look like this:



A little bit of theory:

Let’s consider particles undergoing Brownian motion. Let’s suppose that all the parti- cles were released from a single point at t = 0, that r is the distance to this point, and that D is the diffusion coefficient of all these particles in the medium they diffuse in. We can find the equation for their density for instance in one of Einstein’s historical papers (Einstein (1905)):

$$\rho(r,t) = \rho_0 \exp\left(-\frac{r^2}{4Dt}\right)$$

Using this formula, one can derive the mean square displacement (MSD) for such parti- cles. After a delay τ, the mean-square displacement of the particle ensemble is:

$$\mathrm{MSD}(\tau) = \left\langle r^2 \right\rangle = 2dD\tau$$

The plot of the MSD value as a function of time delay τ should be a **straight line in the case of simple freely diffusing movement**. We therefore have a way to check what is the motion type of the particles. If the MSD is a line, then it is diffusing, and the slope gives us the diffusion coefficient.
On the contrary, if the MSD increases faster than at linear rate, then it must be transported, because Brownian motion could not take it away that fast.

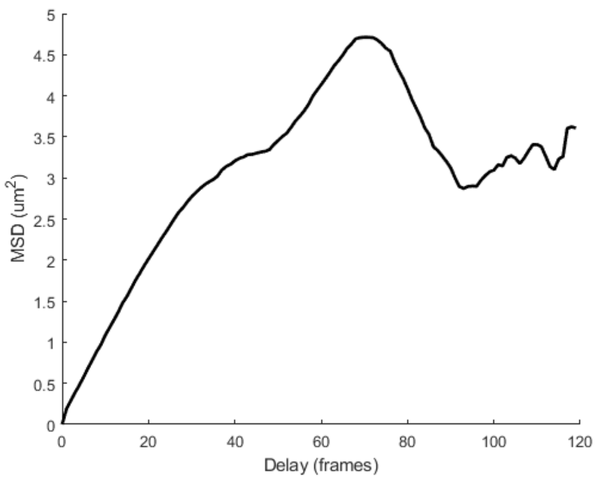
Let’s plot the average MSD considering all particles:

Experimentally, the MSD for a single particle is also taken as a mean. If the process is stationary (that is: the “situation”, experimental conditions, etc… do not change over time) and spatially homogeneous, the ensemble average can be taken as a time average for a single trajectory, and MSD for a single particle i can be calculated as:

$$r_i^2(t,\tau) = \left(r_i(t+\tau) - r_i(t)\right)^2$$

We then average over overall possible t for a given delay τ to yield MSDi (), t and then average the resulting MSDi over all particles:

```
>> ma.plotMeanMSD
```
#before typing this, you might have to close the previous graph



Better data visualization for a better understanding:

The previous graph (generated with ma.plotMeanMSD) is not enough for us to conclude whether the predominant behaviour of these particles is Brownian diffusion or instead directed transport.

A cell is a complex environment and each particle might have different properties that are confused in the ensemble mean plotted above. We therefore turn to another strategy:

Log-Log Fit of the Mean-Square Displacement

Let's consider a single particle diffusing freely. If we compute the logarithm of $\mathrm{MSD}(\tau) = \langle r^2 \rangle = 2dD\tau$ we get:

$$\log\left(\mathrm{MSD}_{\mathtt{diff}}(\tau)\right) = \log(\tau) + \mathcal{C}$$

Let us now consider a particle that moves with a nearly constant velocity vector. In that case, r varies linearly with τ and MSD varies with the square of τ. We then can write:

$$\log\left(\mathrm{MSD}_{\mathtt{trans}}(\tau)\right) = \boxed{2} \times \log(\tau) + \mathcal{C}$$

So **in a log-log plot**, the MSD curves can be approximated by straight lines of:

- slope 1 for diffusion motion
- slope 2 for transported motion
- less than 1 for constrained motion

We can therefore turn this into a test to determine the motion type of our dots.

Application of the "Log-Log fit" function:

% Get the description of the log-log fit function

```
>> help ma.fitLogLogMSD
```

% Perform the fit:

```
>> ma = ma.fitLogLogMSD
```

% Note that now the loglogfit field of the analyzer is not empty anymore:

```
>> ma.loglogfit
```

We are interested **in the slope** alpha, but first we want to remove all fits that had an R^2 value lower than 0.5. The R^2 values are stored in the *ma.loglogfit.r2fit* field.

% Logical indexing:

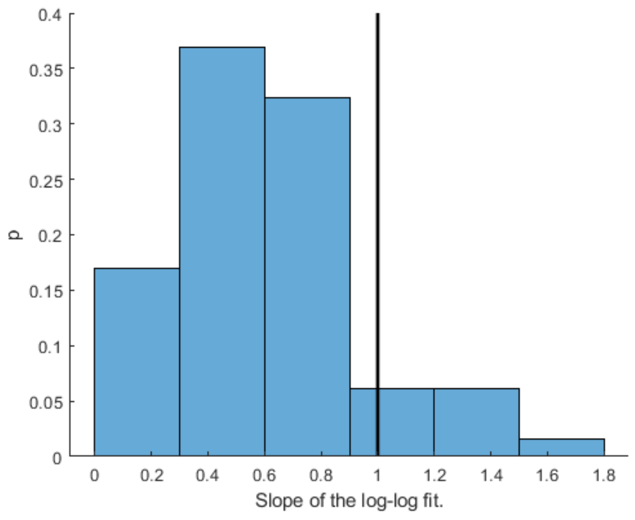
```
>> valid = ma.loglogfit.r2fit > 0.5;
>> fprintf('Retained %d fits over %d.\n', sum(valid), numel(valid))
```

#after this, in the command line, you'll see how many fits were retained.

Visualizing "Log-Log plots":

Now we can plot the histogram of slopes → **copy&paste all these lines together**:

```
>> histogram(ma.loglogfit.alpha( valid ), 'Normalization', 'probability')
>> box off
>> xlabel('Slope of the log-log fit.')
>> ylabel('p')
>> yl = ylim;
>> line( [ 1 1 ], [ yl(1) yl(2) ], 'Color', 'k', 'LineWidth', 2)
```



The histogram displayed above shows several peaks below 1 (around 0.4 and 0.8 judged from its shape). This suggests that there are mixed populations in our dataset, but **most of the particles probably have constrained diffusion (0<slope<1)**

Analysis of the "Log-Log fit":

T-test evaluation for my result:

The population average behavior can be assessed by computing the mean of this distribution and checking whether it is significantly lower than 1 based on a t-test evaluation:

```
>> fprintf('Mean slope in the log-log fit: alpha = %.2f +/- %.2f (N = %d).\n', ...
mean( ma.loglogfit.alpha(valid) ), std( ma.loglogfit.alpha(valid)), sum(valid))

>> [h, p] = ttest( ma.loglogfit.alpha(valid), 1, 'tail', 'left');
if (h)
fprintf('The mean of the distribution IS significantly lower than 1 with P = %.2e.\n', p)
else
fprintf('The mean of the distribution is NOT significantly lower than 1. P = %.2f.\n', p)
end
```

What are the Confidence intervals of my result?

We may ask how many particles have a constrained motility and if they are the majority. A way to assess this at the single particle level is to check the confidence interval for the value of the slope in the fit:

We state that <u>if the confidence interval of the slope value is below 1, then we CAN conclude that the particles have a constrained motility</u>. Again, things are made easy to us, as the confidence interval is also stored in the @msdanalyzer instance:

```
cibelow = ma.loglogfit.alpha_uci(valid) < 1;

ciin = ma.loglogfit.alpha_uci(valid) >= 1 & ma.loglogfit.alpha_lci(valid) <= 1;

ciabove = ma.loglogfit.alpha_lci(valid) > 1;

fprintf('Found %3d particles over %d with a confidence interval for the slope value below 1.\n', ...

sum(cibelow), numel(cibelow))

fprintf('Found %3d particles over %d with a slope of 1 inside the confidence interval.\n', ...

sum(ciin), numel(ciin))

fprintf('Found %3d particles over %d with a confidence interval for the slope value above 1.\n', ...

sum(ciabove), numel(ciabove) )
```

****

The end.