Overview

This code will compute one point (1P) and two point (2P) microrheology on 2D tracked particle data. The 1P is computed from the autocorrelation of beads, by first calculating their mean squared displacement as function of lag time and then using the generalized Stokes-Einstein relation to compute the complex modulus as function of frequency.

The 2P is computed from the cross-correlation between pairs of beads in the same field of view. First the correlations in the direction parallel to the separation vector between pairs and perpendicular to that vector are calculated. Then we compute the effective mean squared displacement which arises from the similarity of the 2P formula relating the modulus to the correlation parallel to the separation vector. Finally, the complex modulus is computed from that effective mean squared displacement. Along the way, the Poisson ratio for the material is computed from the ratio of the correlation parallel and perpendicular to the separation vector.

Code philosophy

The code is spread in a number of folders, organized so that similar functions are grouped together. The actual location of the code does not matter aside from ease of organization, as long as any dependencies are located in the same folder or in the Matlab path.

The functions expect data to be structured so that each experiment has one root folder, which contains the results of the feature finding and tracking algorithms (also available on this site). One can use different algorithms as long as the output is in the same format as what the first functions in this package expect. Functions in this package will often create subfolder to that experiment root folder to store intermediate data, and sometimes will instead return the relevant data as variables. When an input is given in the command line directly from another function's output, the variables are identified in blue.

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. end by a '\' (on a Windows system).
Example:

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

There are rare instances where some quantities relevant to the analysis (for example, the conversion factor from pixel to micrometers) are hardcoded, but once these are set for a given setup, one should not need to modify them further. Any parameter which is likely to change from one experiment to another is passed as a function parameter.

Most of the code assumes 2D tracking data. Some of it can (in theory) accommodate 3D data, but little effort has been spent in our group to fully push the 3D microrheology capabilities.

Initial data processing

The assumption is that the data being processed here has already been through a feature finding and tracking process. To process from scratch (from raw images), the feature finding and tracking code is available here too.

The first step is to convert the positions from pixel to micrometers and to remove drift in the data if present. Ideally, drift should be avoided as much as possible when data is recorded, though we provide a way to remove any that is left here. Then the data is split into individual bead files to be processed by the rest of the analysis. The 1P and 2P microrheology calculations are independent of each other, each one starts from the end of this initial data processing.

In the code folder **Dedriftingandconversions**, there are two possible functions to call: `conversions_no_dd` which will not dedrift the data, or `dedrifting_and_conversions`, which will. Otherwise, the inputs and outputs are the same.

The input is a series of files stored in the `[basepath 'Bead_Tracking\res_files\']` folder, named `res_run##.mat` and containing each a Matlab matrix called `res`. Each row in that matrix should correspond to one bead position, sorted by bead and then by time. The information essential for this piece of code and the rest of the analysis is in columns 1, 2, 6 and 8. Columns 1 and 2: *x* and *y* positions in pixel; column 6: frame #; column 8: bead ID # for this field of view. The complete column information breakdown is: Columns 1 and 2: *x* and *y* positions in pixel; column 3: integrated intensity of the feature; column 4: radius of gyration of that feature; column 5: the eccentricity of that feature; column 6: frame #; column 7: the time at which the frame was recorded, in seconds; column 8: bead ID # for this field of view. The actual time information is lost here, and will be replaced by the average time interval between frames, to make the data processing easier (since we will be dealing with integer frame numbers instead of a continuum of time). It is therefore important to ensure that the frame rate is as constant as possible for the data processing to make sense.

A typical way to call these functions is:

From code root folder

```
cd Dedriftingandconversions

for j = 1:numFOV
  dedrifting_and_conversions( basepath, j, smoo );
% or
```

```
  conversions_no_dd( basepath, j );
end
```

where the `j` index runs over all run numbers for the files in the data folder, and the `smoo` parameter specifies over how many frames the drift is averaged before being subtracted from the bead position. Unless the number of beads per field of view is very large, `smoo` should be large to remove overall drift trends and not actual variations in the bead positions.

The main output is the (possibly dedrifted) converted data, in the `[basepath 'Bead_tracking\ddposum_files\']` folder. Any missing frame in a bead trajectory will be filled by extrapolating linearly from the two nearest frames. Also, .png plots of the drift and average dedrift value are saved in the same folder.

NOTE: the conversion factor is hardcoded in the `pixtomicro.m` file and should be adapted for the hardware setup the data was recorded from. That file is called directly by either front ends.

Dependencies:

```
dedrifting_and_conversions
    pixtomicro
    drift_loop
        from_8_columnns_to_4
        motion
            getdx
                gdxtrinterp
        mot_eintegrate
        smooth_d
        dedrift
        putting_in_missing_frames

conversions_no_dd
    from_8_columnns_to_4
    pixtomicro
    putting_in_missing_frames
```

Once the data is converted, it needs to be separated in individual bead track files so as to simplify the rest of the code. This is done by calling:

```
getting_individual_beads( basepath, FOVs );
```

where `FOVs` is a vector containing a list of the field of view numbers to be processed as one. That is, they need to have the same time separation between frames, and obviously be from the same sample. The input is simply the files output by calling one of the conversion functions.

The output is a series of files in the `[basepath 'Bead_tracking\ddposum_files\individual_beads\']` folder name `bead_#.mat`. Each file contains a matrix called `bsec` which consists of one row per frame, with columns 1 and 2: the *x* and *y* positions, in micrometers; column 3: the frame number; column 4: the bead number (in that frame); column 5: the frame number reset so that the first row is frame 1. Also, a `correspondence.m` file is created with a `correspondance` matrix to convert the global bead number to a field of view number and bead ID number. The columns are: column 1: field of view number; column 2: bead ID number in that FOV; column 3: global bead number.

Dependencies:

```
getting_individual_beads
```

The second step, optional, is to compute the radius of gyration of each trajectory. That information can be used later to filter out beads which have anomalous behavior, for example beads which fail to be caged by a polymer network will have a much larger radius of gyration than beads properly confined.

In the code folder **Radius_of_gyration**, there is one function to call: `rg_matrix_many_single_beads`. This function will take each individual bead trajectories and compute its radius of gyration. The calling sequence is:

From code root folder

```
cd Radius_of_gyration

rg_matrix_many_single_beads( basepath );
```

The inputs are the files output by `getting_individual_beads`. The output is a new `correspondance` matrix stored in a file called `correspondence_rg.m` and which has a 4th column, the value of the radius of gyration, in micrometers.

Dependencies:

```
rg_matrix_many_single_beads
    calc_rg_k
```

Another version of the function is included:

```
rg_matrix_many_single_beads_maxframes( basepath, maxframes
);
```

where `maxframes` sets a maximum number of frames over which the radius of gyration is calculated.

1P microrheology calculations

The data can now be analyzed to get the 1P mean squared displacement (MSD), computed as the sum of the *x* and *y* direction MSDs. The MSD will then be converted to the complex viscoelastic modulus.

First the MSD is computed for each possible lag time `tau` by averaging over all time windows of length tau for each trajectory and over all trajectories. Then a helper function can be called to pick out only a few of those points, spaced logarithmically in time, to make plots more readable and the modulus calculation quicker.

In the code folder **MSD**, the first function to call will be `Mean_SD_many_single_beads`, which does the MSD calculation. A typical calling sequence is:

From code root folder

```
cd MSD

[MSD, tau] = Mean_SD_many_single_beads( basepath, timeint,
number_of_frames, rg_cutoff );
```

where `timeint` is the time interval between frames, in seconds; `number_of_frames` is the maximum lag time to be considered (typically the number of frames recorded per FOV); and `rg_cutoff` (optional) is the maximum Rg value to be included, in micrometers (leave undefined or set to a negative value if the Rg was not calculated).

The input data is taken from the outputs of `getting_individual_beads` and `rg_matrix_many_single_beads`.

The output data are `MSD`, a vector containing the MSD in micrometers^2 for each lag time, and `tau`, a vector containing the list of lag times in seconds. The function also creates a folder `[basepath '\1pt_msd\']` which contains files with the individual *x* and *y* components of the MSD.

Then the MSD can be made to be spaced out logarithmically, with roughly 16 points per decade, using `making_logarithmically_spaced_msd_vs_tau`, which is typically called as follows:

```
[msdtau] = making_logarithmically_spaced_msd_vs_tau( MSD,
tau, maxtime );
```

where `maxtime` is the maximum time in seconds to be output. The inputs are all in the command line, and consist of vectors of `MSD` and `tau`.

The output is a matrix which contains the logarithmically spaced lag times in the first column and the corresponding MSD in the second column.

Dependencies:
`Mean_SD_many_single_beads`

`making_logarithmically_spaced_msd_vs_tau`

The second step consists in calculating the complex viscoelastic modulus using the generalized Stokes-Einstein relation. The conversion to and from the Laplace domain is done by fitting the data locally to a second order polynomial so that the first and second order log derivatives used along the way can be calculated easily. The values obtained for $G(s)$ and $G^*(\omega)$ should be accurate to 1% and 2% respectively using this method, as long as the second order log derivatives are not too large.

In the code folder **moduli**, the function to call is `calc_G`. This function does the modulus calculation and issues a warning if one of the second log derivatives exceeds a hardcoded threshold (0.15) above which calculation accuracy is no longer guaranteed. The polynomial fit is done by weighting the data around the point of interest by a Gaussian based on the distance from that point. Calling sequence:

From code root folder

```
cd Moduli

[omega,Gs,Gp,Gpp,dd,dda] = calc_G( msdtau(:,1),
msdtau(:,2), a, dim, T, clip, width );
```

where `a` is the bead radius, in micrometers; `dim` is the dimensionality of the data; `T` is the temperature in Kelvin; `clip` is a fraction of $G(s)$ below which $G'(\omega)$ and $G''(\omega)$ are meaningless; and `width` is the width of the Gaussian that is used for the polynomial fit. `dim` should be set to 2 in this calling sequence, since the MSD we compute is a 2D MSD (being the sum of the x and y components). `clip` is typically set to 0.03, as values of G' and G'' much lower than 0.03 $G(s)$ are unlikely to be meaningful unless the data is really clean. The `width` can be experimented with, and could be increased for noisier data from the recommended starting value of 0.7.

The input data is again entirely given in the command line, with the first parameter a vector of lag times in second and the second a vector of corresponding MSDs in micrometers^2.

The output is a series of vectors: `omega` is the list of frequencies, in second^-1, corresponding to the lag times in the input; `Gs` is the Laplace domain modulus, in Pascal; `Gp` is $G'(\omega)$, in Pascal; `Gpp` is $G''(\omega)$, in Pascal; and `dd` and `dda` are the second log derivatives computed. The most important variables are typically `omega`, `Gp` and `Gpp`. The second log derivatives can be used to display against `omega` if the warning was triggered. If the value barely goes above the 0.15 limit, then the accuracy may not be too compromised.

Dependencies:

```
calc_G
    logderive
        polyfitw
```

2P microrheology calculations

The data can be analyzed starting from the initial data processing directly to get the complex 2P viscoelastic modulus. First the displacement correlation between pairs of beads is computed. Due to the mathematical similarity between the equation relating the 2P complex modulus and the displacement correlation, one can define an effective 2P MSD. Thus we will compute this 2P MSD, and then process it in the same way as the 1P MSD to obtain the moduli.

In the code folder **Two_point**, the first function to call will be `separating_beads_by_FOV`, which does some basic data processing: it re-assembles the separated bead files into one large matrix. Including all fields of view, so that is can be processed all at once by the 2P algorithm. In order to prevent the 2P algorithm from trying to correlate beads from different fields of view, each field of view is shifted in space by 800 micrometers (hardcoded). Since our images are 212 by 162 micrometers, we limit the range of acceptable distances to 300um (roughly the image diagonal), and there is no risk of dubious cross-talk between fields of view since they are separated by 800 micrometers. This value should be changed if your field of view is larger than 200 micrometers. Another strategy could have been to shift everything in time so that there is no overlap between fields of view.

The calling sequence is:

From code root folder

```
cd Two_point

separating_beads_by_FOV( basepath, rg_cutoff );
```

where the `rg_cutoff` (optional) is the maximum Rg value to be included, in micrometers (leave undefined or set to a negative value if the Rg was not calculated).

The input data is taken from the outputs of `getting_individual_beads` and `rg_matrix_many_single_beads`.

The output data is a file called `beads_separated_by_FOV`, created in the `[basepath '2pt_msd\']` folder, and which contains the `posstot` matrix. That matrix consists of one row per frame, with columns 1 and 2: the *x* and *y* positions, in micrometers (shifted, depending on the field of view); column 3: the frame number; column 4: the global bead number.

Then the output from `separating_beads_by_FOV` should be loaded into Matlab and sent to the `twopoint` function, which does the 2P correlation function calculations. The calling sequence should be:

```
load( [basepath ' 2pt_msd\beads_separated_by_FOV.mat'] );
[data]=twopoint( posstot, [rmin, rmax, nbrbins, maxtime],
timestep, dim, mydts, dedrift );
```

where `rmin` is the minimum separation distance (in micrometers) to compute the correlation for; `rmax` is the maximum separation distance (in micrometers) to compute the correlation for; `nbrbins` is the number log-spaced bins between those two boundaries; `maxtime` is the maximum time (in frames) over which to compute the correlation; `timestep` is the time interval between frames, in seconds; `dim` is the dimensionality of the data (2 or 3); `mydts` is an optional vector of Δt's for which the correlations should be computed (set to 0 to let the program create a logarithmically spaced vector); and `dedrift` is a binary flag, set to 1 to dedrift the data or 0 to leave it as is. The largest range for `rmin and rmax` should be set to a few times the bead diameter and the image diagonal length, respectively. The code was originally written in IDL and we translated it in Matlab, though the 3D part of the code is not translated here, so `dim` should always be 2. Also, we are doing the dedrifting at a much earlier time in the processing, so `dedrift` should always be 0.

The first parameter is the data itself, and should be in a format similar what comes out of `separating_beads_by_FOV`.

The output is a matrix containing the values for the distance and time bins, as well as all the correlation data. It is organized as follows:
In 2 dimensions (polar coordinates):
`data(:,:,1)` contains the log-spaced 'dr' values in micrometers
`data(:,:,2)` contains the log-spaced lag time values in seconds
`data(:,:,3)` is the longitudinal, 'r-r' mean component (Drr) in micrometers^2
`data(:,:,4)` is the transverse, 'theta-theta' mean component (Dthetatheta) in micrometers^2
`data(:,:,5:6)` store the variances of the r and theta parts (in micrometers^2)
`data(:,:,7)` contains the total number of points used in each bin

The final step before computing the complex viscoelastic modulus is to convert the correlation data into a 2P MSD. The heart here is a function called `msddE`, which is surrounded by much front ends to deal with frequent data peculiarities. We occasionally see negative correlation points, coming from noisy 2P data, and those points have to be removed from the averaging, as they are unphysical and the averaging is done on the log of the correlation data, so the logarithm of a negative numbers introduces complex numbers and mess up the calculations. Also, the range over which the correlation goes as 1/r is often limited, and boundaries have to be set on the data points which will be used for the actual fitting.

Thus the main function to call is `calling_two_fitting_methods`. The correlation data will be multiplied by the separation distance *r* before it is: (1) averaged (on a log scale); (2) fitted to a straight line. The rationale behind the latter is that drift, if present, will introduce a distance independent correlation, and thus a term growing linearly with *r* after multiplication by *r*. Therefore the quantity we are after is the constant part of that fit. This method we do not routinely use, and may thus not be as robustly implemented as the average method.

A typical calling sequence would be:

```
minradius(1:size(data,1))=0;
maxradius(1:size(data,1))=rmax;
for j = 1:size(data,1)
  calling_two_fitting_methods( basepath, data, j,
minradius(j), maxradius(j), displaying, beadradius,
savingmsdd, savingplot );
end
```

where `data` is the output of the `twopoint` function; `j` is the lag time index for which to fit the correlation; `minradius(j)` is the minimum separation distance (in micrometers) to be included in the fit; `maxradius(j)` is the maximum separation distance (in micrometers) to be included in the fit; `displaying` is a Boolean set to 1 to display some results as output (typically set to 0); `beadradius` is the bead radius, in micrometers; `savingmsdd` is a Boolean set to 1 to save the 2P MSD information; and `savingplot` is a Boolean set to 1 to save plots of the fitting results. These plots are useful to determine the evolution of the minimum and maximum separation distances over which the average should be calculated.

Care should be taken when limiting the range for the averages so as not to be too arbitrary. Ideally the range would remain fixed for all lag times.

The outputs include a series of subfolders starting at `[basepath 'rDrr_rDqq_figs']`. If `savingplot` is set to 1, the subfolders will contain plots of r*Drr vs r, r*Dthetatheta vs r for method (1), and r*Drr vs r for method (2). It will also plot the number of points used vs r for each tau. Error bars on r*Drr plots come from the standard deviation calculated by `twopoint`. Red symbols indicate data used for the averaging calculation, blue symbols indicate data at too short or too long separations to be included in the calculation and cyan data indicates negative correlations (also not included in the calculations).

If `savingmsdd` is set to 1, it will save a file called `msd2P` in that main subfolder which contains an `msd2P` matrix, a `flor2P` matrix and a `pois` matrix.
`msd2P` matrix format:
 `msd2P(:,:,1)` is the MSD data from the average r*Drr method
 `msd2P(:,:,2)` is the MSD data from the fit r*Drr to a line method
Each row corresponds to one lag time value. The columns are: column 1: lag time, in seconds; columns 2 and 3: raw MSD from r*Drr and r*Dthetatheta; columns 4 and 5: smoothed MSD; column 6: error estimate from the average standard deviations of the r*Drr points; columns 7 and 8: error estimates for r*Drr and r*Dthetatheta MSDs based on the standard deviation of r*Drr with respect to the average (0 for the fit to line method).

The smoothed MSDs are calculated by fitting the data locally to a second order polynomial, by weighing the points with an exponential of the log of the distance from the point of interest. The width of that exponential is hardcoded to 0.9 at the beginning of the `calling_two_fitting_methods.m` file.

`pois` matrix format: one row per lag time, with columns arranged as follows: column 1: lag time in seconds; column 2: Poisson ratio calculated from the ratio of the raw MSD_rr and MSD_theta_theta (columns 2 and 3 in `msd2P`); column 3: Poisson ratio calculated from the ratio of the smoothed MSD_rr and MSD_theta_theta (columns 4 and 5 in `msd2P`).

The flor2P matrix contains information about the linearly varying part in the line fit method. It can be useful to compare how large the drift is compared to the signal. Its columns are arranged as follows: column 1 and 2: slope*(2/(3*beadradius))*(average r) for Drr and Dthetatheta respectively.

Dependencies:
```
separating_beads_by_FOV

twopoint
    ltrinterp
    get_corr
        laccumulate

calling_two_fitting_methods
    fitting_automater_fit_to_line_shell
        msddE
            lfit
    fitting_automater_r_times_Drr_Dqq_shell
        msddE
            lfit
        errorbarlogx
    number_of_points_per_r
    lfit
    poisson_from_msdd
```

NOTES:
As long as the correlation data has no negative points and the range over which the correlation goes as 1/r is fixed for an experiment, one can call `msddE` directly. The steps for smoothing the MSD are currently commented out, so the last two lines of code in `msddE.m` should be uncommented. Its calling sequence is:

```
[msd2P flor] = msddE( data, rmin, rmax, dtmax, a, lfit2, 
width );
```

where `data` is the output from calling `twopoint`; `rmin` is the minimum separation distance (in micrometers) to be included in the fit (one value for all lag times); `rmax` is the maximum separation distance (in micrometers) to be included in the fit (one value for all lag times); `dtmax` is the maximum lag time (in seconds) for which a fit should be computed; `a` is the bead radius in micrometers; `lfit2` is a Boolean for selecting the fitting method: 0 for method (1) (the average) or 1 for method (2) (fit to a line); and `width` is the width of the Gaussian for smoothing (recommended: 0.9, or leave undefined to not smooth the MSD).

The outputs are the 2P MSD and the linearly varying part if method (2) is used, as described above in the outputs for `calling_two_fitting_methods`. The Poisson ratio can then be calculated using the `poisson_from_msdd` function:

```
pois=poisson_from_msdd( msd2P );
```

where the input is the 2P MSD from `msddE` and the output is the same `pois` matrix as described for `calling_two_fitting_methods`.

Dependencies:

```
msddE
    lfit

poisson_from_msdd
```

Now that the 2P MSD is calculated, one can calculate the complex viscoelastic modulus using the same functions as was done in the 1P case, using the generalized Stokes-Einstein relation. The conversion to and from the Laplace domain is done by fitting the data locally to a second order polynomial so that the first and second order log derivatives used along the way can be calculated easily. The values obtained for $G(s)$ and $G^*(\omega)$ should be accurate to 1% and 2% respectively using this method, as long as the second order log derivatives are not too large.

In the code folder **moduli**, the function to call is `calc_G`. This function does the modulus calculation and issues a warning if one of the second log derivatives exceeds a hardcoded threshold (0.15) above which calculation accuracy is no longer guaranteed. The polynomial fit is done by weighting the data around the point of interest by a Gaussian based on the distance from that point. Calling sequence:

From code root folder

```
cd Moduli

[omega,Gs,Gp,Gpp,dd,dda] = calc_G( msd2P(:,1,1),
msd2P(:,4,1), a, dim, T, clip, width );
```

where `a` is the bead radius, in micrometers; `dim` is the dimensionality of the data; `T` is the temperature in Kelvin; `clip` is a fraction of $G(s)$ below which $G'(\omega)$ and $G''(\omega)$ are meaningless; and `width` is the width of the Gaussian that is used for the polynomial fit. `dim` should be set to 1 in this calling sequence, since the MSD we compute from `msddE` is normalized to be a one-dimensional MSD. `clip` is typically set to 0.03, as values of G' and G'' much lower than 0.03 $G(s)$ are unlikely to be meaningful unless the data is really clean. The `width` can be experimented with, and could be increased for noisier data from the recommended starting value of 0.7.

The input data is again entirely given in the command line, with the first parameter a vector of lag times in second and the second a vector of corresponding MSDs in micrometers^2. We are typically using the smoothed MSD, hence column 4 in `msd2P`. Also, we use the averaging method (method (1)), hence the 3rd index is set to 1.

The output is a series of vectors: `omega` is the list of frequencies, in second^-1, corresponding to the lag times in the input; `Gs` is the Laplace domain modulus, in Pascal; `Gp` is $G'(\omega)$, in Pascal; `Gpp` is $G''(\omega)$, in Pascal; and `dd` and `dda` are the second log derivatives computed. The most important variables are typically `omega`, `Gp` and `Gpp`. The second log derivatives can be used to display against `omega` if the warning was triggered. If the value barely goes above the 0.15 limit, then the accuracy may not be too compromised.

Dependencies:

```
calc_G
    logderive
        polyfitw
```

For finding your image features in 2D and linking them together into trajectories to use as input to the microrheology code, see the above algorithms for 2D feature finding, tracking and tracking.