Notes on particle tracking

Raghuveer Parthasarathy

begun April, 2007 – last modified June 11, 2009

General framework:

Load a series of images, e.g. into a 3D array, where the third dimension is frame number.
There are two parts to particle tracking

1. *Finding objects in each frame*
   - Threshold – i.e. remove pixels whose intensity is less than some intensity level
   - Low-pass filter – i.e. average pixel intensities with local neighborhood, to remove high-frequency noise
   - Find objects – look for local intensity maxima, then fit neighborhood to Gaussian form, report center of Gaussian
2. *Link objects together to form a trajectory*

See "Examples" near the end!

Examining and modifying Andrew L. Demond's tracking functions. (RP's versions based on those written by ALD, based on Crocker *et al.* algorithms)

Required Files:

`bpass.m` – by Grier et al., for bandpass filtering
`distance.m` – called by `nnlink_rp.m`
`fityeqbx.m` – RP's "y=bx" line fitting routine, in /MATLAB
`fo4_rp.m` – finds objects in a frame; `called by im2obj_rp.m`
`im2obj_rp.m`
`nnlink_rp.m`
`testthresh.m`
`TIFFseries.m`

Other files:

`showtracks.m` – draws all tracks on one image frame
`simtrack.m` – simulates trajectories, in \Experiments and Projects\Particle Tracking\
`dedrift.m` – drift correction
`msdanalyze_rp.m` – for analyzing single-particle diffusion
`msdtr_rp.m` – for analyzing single-particle diffusion
`calcmeanD.m`
`corr2pt.m` – for 2-point correlations
`trackedit.m` – for rejecting unwanted (short or immobile) tracks
`stepvec_analyze.m` – for calculating frame-to-frame displacement distances and angles
`objs_tabout.m` – Output track data in an easily accessible format
`vidtracks.m` – Superimposes on the series of images the detected tracks, creating a "movie" of the tracked object motion (Richard Holton)
~~`bpfilter.m – creates bandpass filter`~~

Loading a series of images

Goal: load a series of TIFF images (frames of a sequence) into one image array -- outA(:,:,k) = frame #k. Allow the image to be cropped.

Function: `TIFFseries.m`

```
[outA] = TIFFseries()
```

Procedure: Type, for example, im = TIFFseries() – the program will prompt for the filenames, etc.

Thresholding, and object size

Goal: Determine the threshold intensity to consider. Following Andy's procedure, *thresh* is a number between 0 and 1 – we make a histogram of image intensities (in fo4_rp.m) and omit as particles intensity maxima that fall into the zero-to-*thresh* fraction of the distribution. Before thresholding the image is bandpass-filtered – the filter width will affect the threshold, as noise is "averaged-out." Therefore we should determine the threshold and the "object size" (i.e. filter size) together. Note that the larger the object size, the longer the image-dilation step in `fo4_rp.m` will take – this is the main limitation on the tracking speed!

Function: `testthresh.m`

```
function testthresh(A)
```

Procedure:

A graphical interface that allows thresh and objsize to be adjusted. Displays the resulting filtered and thresholded image. Note the values that seem best – the function does not save them. Leave "update" checked to constantly update the image. Uncheck "update" before exiting.

Old Function: `getthresh.m`

```
function [thresh objsize] = getthresh(A)
```

Old Procedure:

An interactive, iterative function. Input is a single image, A. The program asks for a threshold value and an object size (pixels) and shows what the image looks like if these are applied. The user can iterate and choose the threshold and object size that seems best. Note that high threshold values are likely best – around 0.95, for example.

Object recognition (single image)

Goal: return an array of "objects" from a single image

Function: `fo4_rp.m` (RP's modified version of ALD's `fo4.m`)

```
[objs] = fo4_rp(img, objsize, thresh, fitstr)
```

Procedure:

- `objsize` = object size (expected), pixels. `fo4_rp` uses this to define *h* for bandpass filtering, and to define the image dilation structure. Using too small an object size will increase the likelihood of tracking noise, and of finding multiple maxima from one particle. Using too large an object size will slow the program, and might lower the accuracy of particle center finding. Uses Grier et al. bandpass filter function: `bpass.m`.
- Dilation – uses the "structuring element" `ste` created e.g. by "`strel('disk', floor(3),0)`". Determine local maxima as points where (filtered) image equals dilated image, and intensity > threshold value. The threshold is input as a number between 0 and 1 –

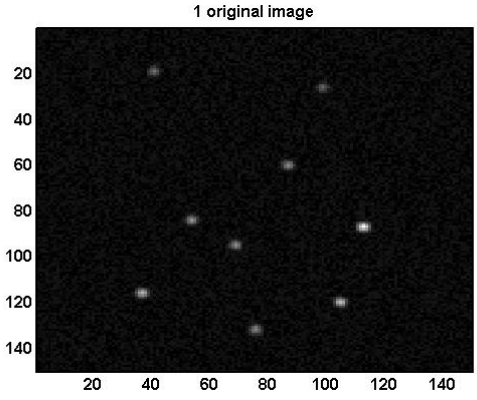
the program makes a histogram of all intensity values, and considers the points above the threshold fraction. Note: requires high value for threshold – otherwise picks out lots of noise spots.

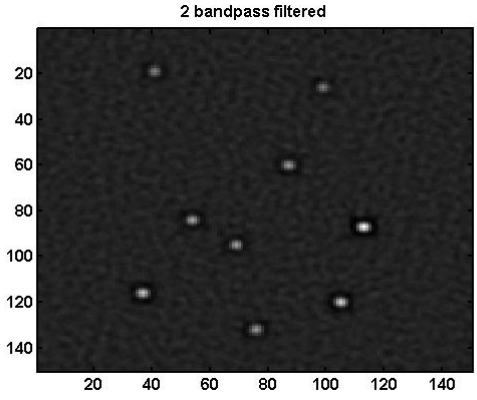
- Gets rid of maxima close to the image edge
- Determine particle center. Default option (omit `fitstr` input, or `fitstr == "gauss"`): by local 2D Gaussian fit to the intensity. (MATLAB multiple regression to 2D Gaussian.) Can instead find the centroid (center of mass) (`fitstr == "centroid"`). Realized (28May07) that this leads to strange near-integer quantization of position values; but also (09 June 2009) that this is necessary for images with saturated pixel intensities.
- Older version: **`fo4_rp_bpfilter.m`**: Bandpass filtering of image – uses objsize to create a bandpass filter, `h`, from **`bpfilter.m`** . Finds minima by filtering with the inverse bandpass filter. Input: *p*: Filter maxima by using delaunay triangulation to find each maximum's 3 closest minima; checks that the intensity of the maximum is > *p* standard deviations above that of the minima. (RP's modification – ALD's original uses a student's t-test). Use $p \leq 1$.
- Speed: to run `fo4.m` on a ~500x500 uint16 image with *objsize*=18 pixels takes about 7 seconds. Slowest part is *imdilate*. *objsize*=9 takes about 2.5 seconds per frame.

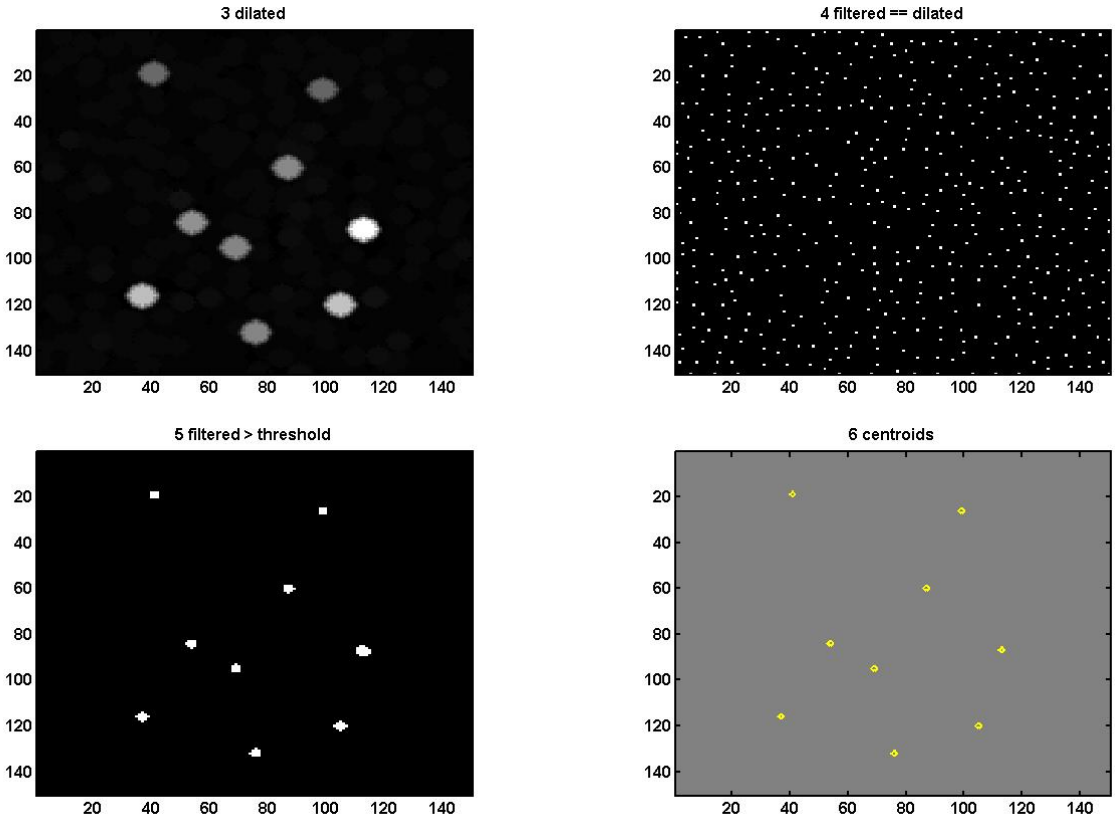
Example: Create an image of blurred points with Photoshop ("test_fr1.tif"). Add noise, and then use fo4_rp. Plot found points with:

```
for j=1:size(objs,2)
      rectangle('Position',[objs(1,j)-1,objs(2,j)-1,2,2]);
end
```

See figures below.







Objects are intersection of 4, 5, with additional tests

Differences between `fo4.m` and `pkfind.m`

- **Local maxima.** pkfind.m looks for local maxima by checking each point to see if it's brighter than its neighbors. f04 dilates and find points in which the image matches its dilation.
- **Thresholding.** pkfind checks if a point is brighter than some threshold. f04 makes a histogram of brightness and considers points brighter than some threshold fraction.
- **f04.m** locates brightness minima around each maximum, and discards maxima that aren't much different than the minima.
- **Centroid** finding is "built-in" to f04.m.

Object recognition (series of images)

Goal: return an array of "objects" from series of images
Function: **`im2obj_rp.m`** (RP's modified version, based on Andy's "customized" **`im2obj.m`**, based on Crocker et al. algorithms)

```
objs = im2obj_rp(im, objsize, thresh, fitstr)
```

Procedure:

- input `im` = 3D matrix of images – `im` (:,:,1) = image 1, etc.

- Calls **`fo4_rp`** for each frame
- `objsize` = object size (expected), pixels. `fo4_rp` uses this to define *h* for bandpass filtering, and to define the image dilation structure. Using too small an object size will increase the likelihood of tracking noise, and of finding multiple maxima from one particle.
- `thresh` = threshold (see **`fo4_rp.m`**)
- `dispopt` = if TRUE, display progress using a progress bar
- `fitstr` : Method for determining particle center, for input into **`fo4_rp.m`** (see description above). Optional input; default is Gaussian (`fitstr == "gauss"`).
- Checks for likely multiple identifications of the same particle --- finds pairs within the same frame that are separated by <2*`objsize`. If these are found, user should re-run the function with a larger value of `objsize`.

Linking objects into tracks

Goal: Identify objects from a series of frames that correspond to the same particle.

Function – RP's: `nnlink_rp.m` . RP's modification just identifies the "nearest neighbor" of an object in one frame as being the object in another frame that corresponds to the same particle. "Cut and paste" the rest of Andy's function – complicated.

```
objs = nnlink_rp(objs, step, memory)
```

(Function – Andy's:) `link4.m` . Andy links particles using a neural network algorithm, calling **`nlink.m`**, that looks at velocity correlations between the particles. We should avoid this since our particles are Brownian, and we want to *extract* the correlations.

Procedure:

- input `objs` = array of objects
- `dispopt` = if TRUE, display progress on screen
- Loop through frames. Find all objects in the current frame and the next frame.
- Determine a "forward map" of nearest neighbors – the index #s in the next frame that correspond to the nearest neighbors of the current frame
- Determine a "backward map" of nearest neighbors – the index #s in the current frame that correspond to the nearest neighbors of the next frame
- Keep only particles for which the two maps are the same
- Cull all links that are greater than sqrt(step*memory) away
- Assign new track ids to "newly found" particles
- I don't yet understand *memory*... I think we should use *memory* = 1.

Plotting tracks

Goal: Draw the particle tracks described in the object matrix.

Function: `showtracks.m`

```
showtracks(objs, Nmin, h)
```

Procedure:

- Displays track positions from objs
- If Nmin is empty ("[ ]") or 0 show all tracks; else, display only tracks with >Nmin points
- Display in the window with handle *h*. If there are only 2 input arguments, create a new window.

Correcting for sample drift

Goal: Correct tracked positions to account for (constant velocity) drift of the sample field
Function: `dedrift.m`

```
objs = dedrift(objs, dispopt)
```

Procedure:

- Create object matrix as usual, from `im2obj_rp` or `nnlink_rp`.
- For each track, this function determines the slope of x(t) and y(t), then subtracts a linear function with the median to the x and y positions of all objects in each frame. Does not use a linear fitting of mean positions, as this may be sensitive to particles entering or leaving the field of view. Does not track the change in position of the center of mass – for many particles, this doesn't change much as particles drift out of the field and new ones drift in. Notes which frames exist, to avoid problems from missing frames
- dispopt = display option (plot if true)

Calculating mean-squared-displacements

Goal: For each track, determine the mean-squared displacements
Function: `msdtr_rp.m`

```
msd = msdtr_rp(objs, fps, scale)
```

Procedure:

- Link objects into tracks, using `nnlink_rp`, e.g.
- "fps" = frame rate (frames per second)
- "scale" = image scale (microns per pixel)
- For each track, the function calculates $\Delta x^2(\tau)$ for every pair of separation times (τ) – see listing. More frames are available for smaller separation times; preserve the mean value of Δx^2 as well as the standard deviation, for use in line fitting by `msdanalyze_rp`.
- Output: array [msd]: ("msd12, e.g., is Δx^2 for track 1 for τ =2 frames, i.e. τ_{12} seconds).

```
msd = {msd11, msd12, ... , msd21, ...;
       std11, std12, ... , std21, ...;
       tau11, tau12, ... , tau21, ...;
       tr1,   tr1,   ... , tr2,   ...}
first index is track number, second is time-delay tau index
msd : mean mean-squared-displacement during the each time delay
std : standard deviation of msd during the each time delay
tau : the time step
tr  : the trackid, corresponding to the track id in objs.
```

Analyze, plot mean-squared-displacements

Goal: For each track, extract a diffusion coefficient and plot (optional) the mean-squared displacements versus time

Function: `msdanalyze_rp.m`

```
D = msdanalyze_rp(msd, cutoff, plotopt)
```

Procedure:

- Get msd matrix using `msdtr_rp`, e.g.
- For each track, calls **`fityeqbx`** to fit $\Delta x^2 = 4D\tau$ -- notes D and uncertainty from fit, σ_D. As noted in Saxton 1997, the function does not fit all the τ values, but employs a cutoff – suggested value 0.25 (i.e. one-quarter of the track length)
- If *plotopt*==true, plot $\Delta x^2(\tau)$, up to the cutoff value, for each track
- Output: D [array]
    - D(1,:) are the diffusion coefficient values -- $\mu m^2/s$, if proper scales
    - D(2,:) are the uncertainty (sigma_D) of fit values
    - D(3,:) are the number of points in the track's msd data
    - D(4,:) are the track ids

Calculate the mean diffusion coefficient

Goal: calculate the mean diffusion coefficient from all the tracks

Function: `calcmeanD.m`

```
[meanD stdD Ngood] = calcmeanD(D, Nmin, Dmin, dispopt)
```

Procedure:

- From the matrix D of diffusion coefficient values for each track (e.g. from `msdanalyze_rp.m`) calculate a mean diffusion coefficient.
- Input: *Nmin* -- for determining the mean diffusion coefficient, consider only tracks with more than *Nmin* points.
- Input: *Dmin* -- for determining the mean diffusion coefficient, consider only tracks whose D value is greater than *Dmin*. (First run with Dmin = 0 to see the full histogram).
- Input: *dispopt* – if true (or if not input), plot the histogram of D values.
- Output: meanD, stdD = mean diffusion coefficient (of *Ngood* tracks with *D* > *Dmin* and more than *Nmin* points), calculated as a simple mean, and the simple standard deviation. *Formerly*, used a weighted average – weights are the σ_D values, with stdD being the square root of the weighted mean square deviation – led to severe overemphasis of low-mobility tracks (RP & CH, 19July07).

Simulate tracks

Goal: Simulate 2D Brownian motion and generate images, to test particle tracking programs

Function: `simtrack.m`

```
function [A, x, y] = simtrack(Np, Nsteps, Inoise, Iparticle, ...
      objsize, stepstd, TIFFopt)
```

Procedure:

- Input: `Np` -- images are `Np` x `Np` pixels
- Input: `Nparticles` -- number of particles. All are created with the same size, brightness. Program DOES NOT check if the particles overlap

- Input: `Nsteps` -- number of steps to take (`Nframes = Nsteps + 1`)
- Input: `Inoise` -- to each frame we add random noise uniformly distr. in the interval [`0`,`Inoise`]
- Input: `Iparticle`, `objsize` -- particle appears as a Gaussian spot of max. intensity `Iparticle`, standard deviation `objsize` pixels.
- Input: `stepstd` -- at each step, move randomly in x, y with a Gaussian probability distribution centered at 0 with `sigma=stepstd`
- Input: `TIFFopt` -- if true, save simulated images as a series of 8-bit TIFFs
- Output: Series of images: `A`(`nrows, ncolumns, nframes`) -- 0-255 uint8; Series of x- and y-positions: `x`(`Nparticles, nframes`), `y`(`Nparticles, nframes`), pixels

Two-point correlations

Goal: Analyze particle track data to extract two-point correlation functions.
Function: **`corr2pt.m`**
Procedure: (see RP, CH)

Analyze two-point correlation data

Goal: Analyze two-point correlation output of corr2pt.m
Function: **`corr2pt_analyze.m`**
Procedure: (see RP, CH)

Track several series of images

Goal: For several series of images, obtained e.g. as a series of MAT files during the analysis of particles in a line trap, perform the tracking and return one object file.

Display track properties and reject unwanted tracks

Goal: Cut tracks from the object array based on being too short (too few frames) or being stuck (too little variance)
Function: **`trackedit.m`**

```
function [objs_out] = trackedit(objs, minNframes, minstd)
```

Procedure: (see m-file)

- First, track objects, link tracks (`nnlink_rp.m`)
- Input

```
objs -- 6-row linked object matrix, from nnlink_rp.m  (See that file,
        im2obj_rp for object structure)
minNframes -- minimal number of frames (optional; can input during
analysis)
minstd -- minimal standard deviation, px (optional; can input during
analysis)
```

- Output: an object array with only the "kept" frames

```
objs_out -- tracks that are kept (at least minNframes)
```

- If input arguments are omitted, ask for user input, and make plots.

Calculate frame-to-frame displacement (speed) and angle

Goal: Calculate frame-to-frame displacement (speed) and angle of traejctories

Function: `stepvec_analyze.m`

```
function stepvec = stepvec_analyze(objs)
```

Procedure:

- Input: `objs :` object matrix, e.g. from nn_link
- Output: stepvec [array]

  stepvec = {d11, d12, d13, ... , d21, ...;
  theta11, theta12, theta13, ... , theta21, ...;
  fr11, fr12, fr13, ... , fr21, ...;
  tr1, tr1, tr1, ... , tr2, ...}

  first index is track number, second is frame index (runs from 1 to N-1, where N is the number of frames of the track)

  dij : magnitude of the displacement of track i between frames j, j+1 (pixels)

  thetaij : angle (radians) of the displacement of track i between frames j, j+1

  fr : the frame number. If the track is present for all frames, this will be [2 3 4 5 ... N], corresponding to the steps of frame 1-2, 2-3, 3-4, etc. If the track starts at frame 15 and ends at 38, for example, this will be [16 17 ... 38]

  tr : the trackid, corresponding to the track id in objs.
- Output can be "printed" to a text file with **`objs_tabout.m`**

Output object or displacement matrix data to a tab-delimited file

Goal: Output track data in an easily accessible format

Function: `objs_tabout.m`

```
function outA = objs_tabout(A, fname)
```

Procedure:

- function to "print" the contents of an object matrix (position, in "objs") or a displacement matrix ("stepvec") as a tab-delimited text file
- In each of these, the first two rows are the "data" -- position (x,y) or displacement (distance, angle), the last row is the track id, and the second-to-last-row is the frame number.
- Input:

  A : objs matrix or stepvec matrix

  fname = output file name.
- Output

  outA : returns a matrix in the same format as the printed matrix

  Writes a tab-delimited text file. Each row is a frame, each col. is position or displacement data -- two columns per track.

  For frames in which the track does not exist, leave empty.

  E.g. for object data (objs) of M tracks with (up to) N frames:

  outA = {x11, y11, x21, y21, x31, y31, ... xM1, yM1;
  x12, y12, x22, y22, x32, y32, ... xM2, yM2;
  x1N, y1N, x2N, y2N, x3N, y3N, ... xMN, yMN}

  where the first index is the track id and the second is the frame number}

  E.g. for step data (stepvec) of M tracks with (up to) N frames:

outA = {d11, theta11, d21, theta21, d31, theta31, ... dM1, thetaM1;
d1(N-1), theta1(N-1), ... dM(N-1), thetaM(N-1)}
where the first index is the track id and the second is the frame number}

Calculate angular correlation function

Goal: Calculate the temporal correlation function of track angle

Function: `angleCorr_rp.m`

```
function thetacorr = angleCorr_rp(stepvec, fps)
```

Procedure:

- First use "`stepvec_analyze.m`" to get the frame-to-frame displacement (e.g. `stepvec = stepvec_analyze(objs_out);`)
- This function is similar in format to `msdtr_rp.m`
- Input:
  stepvec : displacement matrix, created by stepvec_analyze.m
  Row1 = displacement magnitude
  Row2 = angle
  Row3 = frame number. If the track is present for all frames, this will be [2 3 4 5 ... N], corresponding to the steps of frame 1-2, 2-3, 3-4, etc. If the track starts at frame 15 and ends at 38, for example, this will be [16 17 ... 38]
  Row4 = track id
  fps : frame-rate (frames per second)
- Output:
  thetacorr : angular correlation matrix, stored thusly
  thetacorr = {theta11, theta12, ... , theta21, ...;
  std11, std12, ... , std21, ...;
  tau11, tau12, ... , tau21, ...;
  tr1, tr1, ... , tr2, ...}
  first index is track number, second is time-delay tau index
  theta : normalized angular correlation during the each time delay
  std : standard deviation of theta during the each time delay
  tau : the time step
  tr : the trackid, corresponding to the track id in objs.
- **Analysis:** Note that the format of the output is very analogous to the mean-square-displacement output of `msdtr_rp.m`. The output "thetacorr" array can be "binned" by `binavg`, for easy plotting. Example, plotting the average correlation vs. time delay:

```
[Tx, stdx, nx, ind, outN] = binavg([thetacorr(3,:); thetacorr(1,:)], 0.5:60.5);
figure; plot(Tx(1,:), Tx(2,:), 'o', 'Color', [0.7 0.2 0])
xlabel('\tau, frames');
ylabel('Angular correlation function')
```

- **Analysis (2):** One can also use `msdanalyze.m` to plot the correlation function for each track individually: `T = msdanalyze_rp(thetacorr, 0.5, 1);  [Last option==1 for plotting]`

Create a movie of tracks

Goal: Make movie of tracks, superimposed on the image series

Function: `vidtracks.m`

```
function [ movie ] = vidtracks(im, objs, fps, filename, comp, qual,L)
```

Procedure:

- See file for details.
- Input:
  im : 3D image matrix, e.g. from TIFFseries.m
  objs : 6-row linked object matrix, from nnlink_rp.m or another program of that type (see im2obj_rp for object structure)
  etc. (several optional inputs)

To do:

- Include brightness matching in track identification? – or a "fix" afterwards, connecting tracks that end and tracks that start at a particular frame if they match brightness
- Combined graphical user interface
- Read Guilford paper on tracking algorithms
- Trackedit: GUI, with two images. 1) track frames 2) diffusion coefficient histogram. Two cutoffs, for each. Plot what's kept in a different color.
- Function to "invert" images for tracking black spots. Not really necessary for image matrices, since one can trivially invert "im = (2^N-1) – im", where N is the number of bits of the camera.

Example:

```
im = TIFFseries();  % loads TIFF images into an array ("im")
testthresh(im(:,:,1)); % interactively determine threshold and object size, using frame 1
objs = im2obj_rp(im, 8, 0.9999);  % find objects (here, objsize=8 and thresh = 0.9999)
objs_link = nnlink_rp(objs, 200, 1);
        % link objects, find tracks (here max step size squared = 200)
objs_out = trackedit(objs_link);
        % remove "unwanted" tracks – you'll be prompted for the inputs
        Or trackedit(objs_link, 10, 0.01); for min. 10 frames, min std 0.01 px
h = figure; imagesc(im(:,:,1)); colormap('gray') % Show the first image
showtracks(objs_out, [], h); % draws tracks on the image in figure h
```

Saving. A good idea: **Save the object matrices** (& other things)
Save the first image as an array (`A = im(:,:,1);`) and clear the "im" variable
MATLAB "save" command
e.g. `save('track_3_0p5.mat', 'objs_link', 'objs_out').`

Example #2: calculating mean square displacement

[Do all the steps in Example #1]

```
fps = 1085/60; scale = 0.33;  % frames per second, microns per pixel
msd = msdtr_rp(objs_link, fps, scale);
D = msdanalyze_rp(msd, 0.25, true);
[meanD stdD] = calcmeanD(D, 25, 0.0001, true); % mean diffusion coefficient for tracks with > 25 (here) pts.
```

Example #3: export as tab-delimited text

[Do all the steps in Example #1]

e.g. to output all the track information from objs_out matrix as a tab-delimited file, with rows = frames, and columns =tracks (2 cols./track):

```
outA = objs_tabout(objs_out, 'objs_out_tab.txt');
```

Example #4: angular correlations

[Do all the steps in Example #1]

```
stepvec = stepvec_analyze(objs_out);% To analyze the frame-to-frame displacement, angle
outA2 = objs_tabout(stepvec , 'stepvec _tab.txt');
      % One can export this as a tab-delimited file
thetacorr = angleCorr_rp(stepvec);
      % calculate the angular correlation function for each track
T = msdanalyze_rp(thetacorr , 1, true);
      % Optional – plot the angular correlations for each track
[mx, stdx, nx, ind, outN] = ...
       binavg([thetacorr(3,:); thetacorr(1,:)], 0.5:60.5);
      % Average the angular correlations for all tracks
figure; plot(mx(1,:), mx(2,:), 'o', 'Color', [0.7 0.2 0])
xlabel('\tau, frames');
ylabel('Angular correlation function')
-----
```

A good idea: **Save the object matrices** (& other things) – see above