ERIC M. FURST

PARTICLE TRACKING WITH MATLAB

DEPARTMENT OF CHEMICAL AND BIOMOLECULAR ENGINEERING
UNIVERSITY OF DELAWARE

Version 2015-10-20-a
First version June 27, 2013

Department of Chemical and Biomolecular Engineering
Allan P. Colburn Laboratory
150 Academy Street
University of Delaware
Newark, Delaware 19716 USA

email—furst@udel.edu
website—LEM.CHE.UDEL.EDU

October 21, 2015

Contents

List of Figures

Introduction

This tutorial summarizes the basic steps in particle tracking using the Matlab code authored by Daniel Blair and Eric Dufresne. The Matlab routines are based on the IDL code by John Crocker and Eric Weeks.

This tutorial is organized into three main chapters with the following aims:

- Summarize the methods for evaluating images best suited for particle tracking, including particle image size, image contrast, and quantifying signal-to-noise;
- Understanding the steps used to identify particles in each frame;
- Track particles in movies to find the positions in each frame and link these into trajectories that can be analyzed.

First-time readers may want to skip the first chapter on image quality and proceed directly to the particle tracking chapters. The routines for tracking in a movie are described in *Creating particle trajectories*, but readers should work through the subroutines of the chapter *Particle tracking steps* to understand important parameters that must be set. These parameters have a significant effect on the tracking quality.

Imaging for particle tracking

Image quality

Particle tracking begins with images of particles collected using video microscopy. While bright field images may be used, fluorescence microscopy provides a high contrast that leads to a good signal to noise ratio (SNR). The chief disadvantages of fluorescence imaging are photobleaching over time and the background fluorescence produced by particles out of the focal plane, which contributes to the noise when estimating the locations of the particles.

Fig 3 shows a sample image of fluorescent particles. The image has 8-bit depth, so the grayscale pixels have integer values between 0 and 255. The image is 574×574 pixels. It was cropped from a larger image from a megapixel (1024×1024) camera. Notice that the image has a non-uniform background intensity. This must be corrected before particle tracking. The image has a good dynamic range that uses a wide extent of the possible 8-bit values. A histogram of the pixel intensity is shown in fig 3. Only two pixels reach the maximum value 255, and the rest are distributed with intensities well below this. The brightest pixels associated with in-focus particles are in the range of 200-255. A slightly lower intensity could be used to avoid image saturation, when multiple pixels are at the highest intensity value of the camera.

Read this image file using the command:

```
>> I = imread('SampleImage.tif');
```

A quick way to check the image histogram is using the `imhist` command in Matlab:

```
>> imhist(I);
```

We can also count the number of saturated pixels using the `histc` command. Assuming an 8-bit image in which the maximum value is 255,

```
>> histc(I(:),255:255)
```

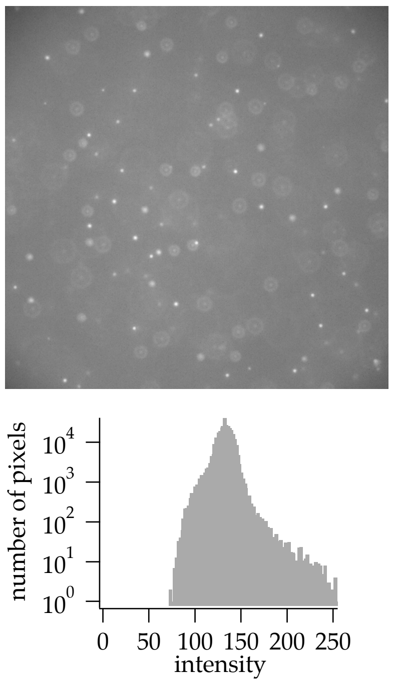

Figure 1: Sample fluorescence image and image histogram.

```
ans =

      1

>>
```

The sample output indicates that there is one pixel that has the maximum value, which is acceptable. A large number of saturated pixels would likely lead to poor particle tracking. It is best to detect saturated pixels at the beginning of an experiment.

In a fluorescence image, particles that are closest to the focal plane appear as bright disks with an approximately Gaussian intensity distribution. Particles above and below the focal plane will appear as rings with a central spot, as shown in fig 2. These images typify the convolution of the particle image with the *point spread function* (PSF) of the imaging system. The PSF represents an intensity distribution resulting from a point source when viewed through the microscope. The maximum lateral resolving power d of a microscope is determined by the diffraction limit

$$d = \frac{0.61\lambda_0}{\mathrm{NA}} \tag{1}$$

where NA is the numerical aperture of the objective.[1] For a high quality water immersion microscope objective with NA $=$ 1.2 and fluorescence emission $\lambda = 520$nm (the peak emission for fluorescein isothiocyanate, FITC) the lateral resolution is at best 520 nm, which is substantial fraction of the physical particle size. Non-immersion objectives are limited to NA values below 1; the lateral resolution for a 40× plan-apochromat objective with NA $=$ 0.7 is 890 nm. It is important that the rings remain symmetric as the particles move in and out of the focal plane. Rings that appear pinched or skewed are an indication that the illumination or imaging system is not aligned properly.

[1] The numerical aperture also determines the amount of light collected by the microscope.

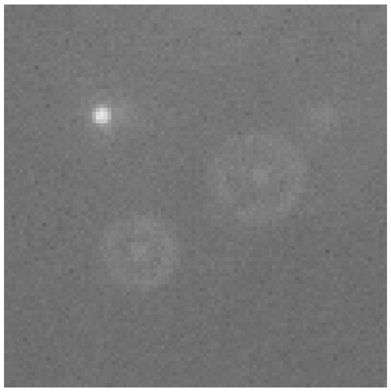
Figure 2: Images of one particle in the paraxial focal plane and two that are out of focus.

A number of factors influence the quality of video microscopy data, including:

1. exposure time and frame rate of the camera
2. detection noise
3. fluorescence brightness of the particles and electronic gain of the detector

Video microscopy uses an electronic camera with a pixelated array of detectors. Like any optical detector, the camera converts light power to electrical current, and can be a charge-coupled device (ccd), intensified ccd (iccd), electron-multiplied ccd (EM-ccd) or so-called CMOS device based on an active-pixel sensor. The basis of operation is similar. Each pixel accumulates photoelectrons during an *integration time*,[2] σ. The number of charges accumulated is converted by the camera to a numerical value (digital) or voltage (analog) representing the detected light intensity.

[2] Also referred to as exposure time or shutter time.

Frame rate and exposure time

Most video cameras today acquire images at a *frame rate* specified by the user, such as 100 frames per second (fps). The frame rate is the time between complete images, and ultimately limits the shortest lag times of particle tracking data $\tau_{\min} = 1/f$, are chosen such that the exposure time σ is at most one tenth as long as the time between frames, $\sigma \leq 0.1\tau_{\min}$. This ensures that the particles do not move too much during the image acquisition. The minimum exposure time will depend on the sensitivity of the camera, the intensity of fluorescence emission of the particles, and the overall tolerance to noise.[3]

[3] Are there any examples of the particle intensity as a function of size?

The frame rate of NTSC compatible ccd cameras (known as the RS-170 standard) is fixed at 30 fps. Such analog cameras are still common and can be recorded directly to video tape or DVD for later retrieval.[4]

[4] The NTSC standard produces 480i video—480 interlaced vertical lines. The horizontal resolution is typically 640 pixels and the pixels have an aspect ratio of 4:3. PAL video standards used in Europe are 525i at 25Hz.

Detection noise

Noise is inherent to electronic imaging systems due to both the quantization of light as photons and the electronics that carry minute photoelectron charges and convert them into the information we see as a pixel value in an image.

Photons are absorbed by the detector to create photoelectrons.[5] The instantaneous conversion rate $\alpha(t)$ in photoelectrons per time is proportional to the instantaneous incident power $W(t)$ at the detector,

[5] This is the semi-classical description of light; it propagates as waves, but is detected as particles.

$$\alpha(t) = \frac{\eta(\lambda)}{h\nu} W(t) \tag{2}$$

where $\eta(\nu)$ is the detector *quantum efficiency* at the wavelength $\lambda = c/\nu$ and h is Planck's constant.[6] While eqn 2 is straightforward in that the instantaneous photocurrent is proportional to the instantaneous power, consider that photons arrive at random intervals, and thus $W(t)$ is stochastic. Therefore, the number of photoelectrons generated during the detector integration time will vary. This fluctuation is the *shot noise*. If n is the measured number of photoelectron conversions over the detector integration time σ, then the mean value is simply $\langle n \rangle = \sigma \langle \alpha \rangle_\sigma$ photoelectrons per second, but the normalized variance is

[6] What are the typical quantum efficiencies of ccd cameras?

$$\frac{\langle n^2 \rangle - \langle n \rangle^2}{\langle n \rangle^2} = \frac{1}{\langle n \rangle} + \frac{\langle W^2 \rangle_\sigma - \langle W \rangle_\sigma^2}{\langle W \rangle_\sigma^2} \tag{3}$$

where $\langle W \rangle_\sigma$ is the time-averaged source intensity, which is also assumed to fluctuate. There are two contributions to the normalized variance of n in eqn 3—one from the fluctuations of the source, and the first term on the right side of the eqn, which comes from the quantum noise. We see that the shot noise scales inversely with the mean photoelectron current. Shot noise will limit the signal-to-noise ratio of an image at low light intensity levels or short exposure times. In general, the SNR is the inverse of the normalized variance. Based on eqn 3, In the absence of source fluctuations, the SNR is proportional to the rate of photoelectron conversions. Therefore, increasing the light intensity or integration time should improve the SNR.

In most cases, shot noise will be the limiting source of noise of the camera. Nonetheless, there are several forms of noise in video imaging that should be also kept in mind. An additional source of noise is the *dark current* of the detector. This noise is thermionic in origin for semiconductor-based detectors—the thermal energy at the detector active area produces a background current even in the absence of light. This is the reason that some sensitive cameras used under low light conditions or for short exposure times employ a cooled detector. Similarly, there is the *Johnson noise* of the camera electronics due to the finite temperature of the charge carriers.[7] Johnson noise is a manifestation of fluctuation-dissipation. Finally, *readout noise* arises in the photoelectron digitization process. For ccd cameras with slow readout frequencies (< 1 MHz), typical readout noises are small. However, for fast ccd cameras (readout frequencies ≥ 10 MHz), readout noise can dominate shot noise.

[7] See H. Nyquist, "Thermal Agitation of Electric Charge in Conductors", *Phys. Rev.* 32, 110 (1928)

Signal to noise

In image processing, it is common to characterize image SNR in decibels (dB) as[8]

$$\mathrm{SNR} = 10 \log \left(\frac{\sigma_{\text{image}}}{\sigma_{\text{noise}}} \right) \tag{4}$$

[8] See Russ, p. 376

where σ_{image} and σ_{image} are standard deviations of the image and noise pixel intensities, respectively. The simpler images of bright particles on a (mostly) dark background suggest simpler criteria for the SNR. Savin and Doyle,[9] for instance, calculate the signal as the difference between the local maximum brightness value of a particle and the average brightness around the spot. The noise is the standard deviation of the brightness in regions that exclude in-focus and out-of-focus particles.[10] The Rose Criterion states that a SNR of at least 5 dB is required to distinguish image features with 100% certainty.[11]

[9] Savin, T. & Doyle, P. S. Static and Dynamic Errors in Particle Tracking Microrheology. *Biophys. J.* 88, 623–638 (2005).

[10] Some define this as a contrast-to-noise (CNR) ratio. See, for instance, Edelstein, W. A.; Bottomley, P. A.; Hart, H. R.; Smith, L. S. (1983). "Signal, noise and contrast in nuclear magnetic resonance (NMR) imaging". Journal of Computer Assisted Tomography 7 (3): 391–401.

[11] Bushberg, J. T., et al., The Essential Physics of Medical Imaging, (2e). Philadelphia: Lippincott Williams & Wilkins, 2006, p. 280.

LET'S TAKE A CLOSER LOOK at the signal and noise histograms.

The function `SNR.m` provides a measure of the signal-to-noise ratio of a particle tracking image. Based on Savin and Doyle's method, particles are located by the particle tracking methods discussed below. These particle images are masked and the pixel intensities of the remaining image calculated. This constitutes the *background* from which particles are distinguished. Likewise, the pixel intensities of the masked particles are calculated.

This is a measure of the signal. The SNR is

$$\mathrm{SNR} = \frac{\overline{\mathrm{signal}} - \overline{\mathrm{noise}}}{\sigma_{\mathrm{noise}}}. \tag{5}$$

Generate a histogram of the noise with the command

```
>> I = double(imread('SampleImage.tif'));
>> [signal, noise] = SNR(I,5);
```

The second value in `SNR` is the typical feature size, roughly the dimensions of a particle in pixels. The `SNR.m` routine will output the following:

```
Maximum imagebp value is 53.1866
Using values > 31.912
located 24 particles.
Mean signal: 222
Mean noise: 129.9838
Standard deviation noise: 9.5949
Signal-to-noise: 9.5901(9.8182 dB)
```

An image appears that shows the masked regions (signal) and the background region used to calculate the signal-to-noise-ratio.

The signal-to-noise data can be written to files for further analysis. Likewise, generate a histogram for the signal,

```
>> [signalhist,signalbin] = hist(signal,(max(signal)...
                            -min(signal)));
```

and noise

```
>> [noisehist,noisebin] = hist(noise,(max(noise)-min(noise)));
```

then check the histograms in Matlab using

```
>> bar(signalbin,signalhist)
>> hold
Current plot held
>> bar(noisebin,noisehist)
```

The number of pixels that make up noise vastly outnumber the number of pixels that constitutes the signal. We normalize both by their respective areas to illustrate their relative distributions and relationships to each other.

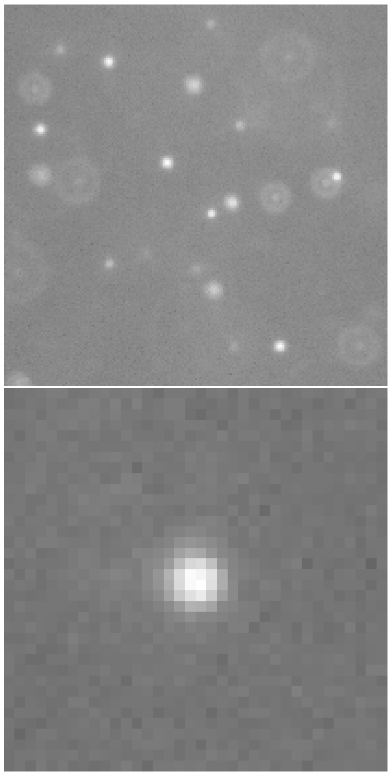
Figure 3: (Top) Image detail before filtering. (Bottom) A close-up of a single particle. Pixel-to-pixel noise is evident in this magnified image.

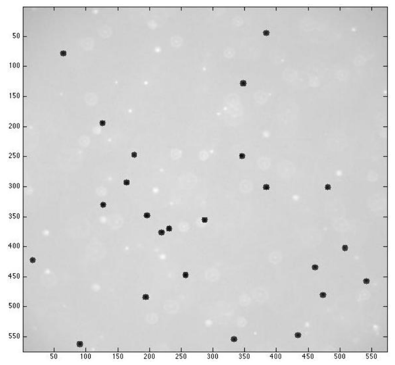

Figure 4: Masked regions (black) used to calculate the signal in reference to the unmasked background regions (noise). Out of focus particles constitute part of the noise in the image.

```
>> sigpix = sum(signalhist);
>> noisepix = sum(noisehist);
>> signalhist = signalhist / sigpix;
>> noisehist = noisehist / noisepix;
```

Now replot the normalized histograms.

```
>> bar(noisebin, noisehist)
```

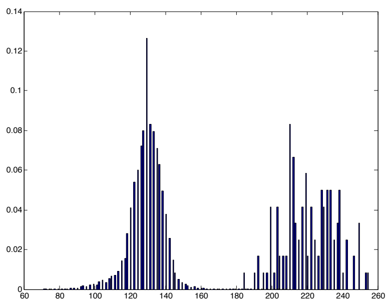

Figure 5: Normalized histograms of the noise (lower values) and signal (higher values). A wide separation of these distributions indicates a high signal-to-noise ratio.

Both signal and noise histograms can be written to files to use other data plotting software:

```
>> dlmwrite('noise_histogram',[noisehist;noisebin]');
>> dlmwrite('signal_histogram',[signalhist;signalbin]');
```

These are shown in fig 5. The normalized histograms clearly show the distribution of the signal and noise intensities, and their separation.

A final note: The `SNR` function masks candidate particles to calculate the image noise. Likewise, the brightness of the particle images is also calculated. However, the brightness distribution is sensitive to the feature size w. Depending on its value, pixels near the edges of particles are included in the noise or with the signal. To reduce this sensitivity, we double the value of w when calculating the mask for the noise image, and use $w - 2$ when evaluating the signal.

Other imaging artifacts

When the integration time of a ccd detector is sufficiently long, photoelectron charges collected in the sensor's electronic "bins" in the brightest part of the image will overflow to neighboring bins. This results in *blooming* of the image. The intensity or exposure time should be reduced to avoid this condition.

Finally, some cameras produce *interlaced* images. Each video frame consists of two *fields* consisting of the odd and even lines of the image. For the RS-170 standard, the fields are acquired 1/60th of a second apart. Since particles may move during the time between the fields are scanned, interlaced images can have a "wavy" appearance that results in poor tracking. These image frames should be *de-interlaced* by using only one of the fields before they are processed further.

Particle tracking steps

In this chapter, we go step-by-step through the particle tracking algorithm. We focus on how images are processed to identify the particles that will be tracked.

The key particle tracking functions introduced here are:

`bpass.m` – Band pass filter used to remove noise in images prior to locating particles

`pkfnd.m` – Identification and location estimate of particles in a single frame based on locating the brightest pixels

`cntrd.m` – Particle location refinement based on calculating the centroid of the intensity distribution around the central bright pixel

The particle tracking routines discussed in the next section will use these three functions on every frame of a movie.

Image filtering

The first step in particle tracking is to filter noise from the image. Read the image in with the command

```
>> I = double(imread('SampleImage.tif'));
```

Fig 3 is a cropped image represented by the two-dimensional array I before the filtering operations are applied.

Contrast gradients in the image complicate the process of identifying potential particles, but such variations can easily be subtracted using a boxcar average due to the small image size and wide separation of the particle features. The boxcar average is taken over a region $2w + 1$, where w is an integer larger than the single sphere's image radius in pixels, but smaller than the spacing between particles. Each pixel is given a new value

$$I_w(x, y) = \frac{1}{(2w+1)^2} \sum_{i,j=-w}^{w} I(x+i, y+j). \tag{6}$$

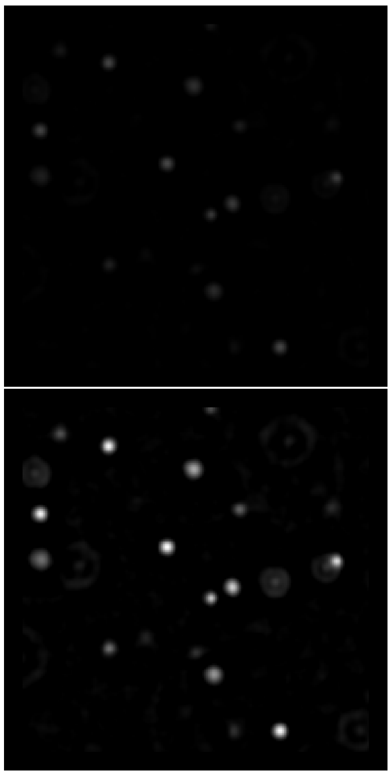
Figure 6: Image detail after applying the `bpass(I,1,10)` function. This top image uses the entire dynamic range of the color map, while the bottom has been scaled.

The second filter reduces the random noise of individual pixels by averaging the value of nearby pixels. We perform a Gaussian average for each pixel with a half-width $\xi = 1$ pixel over the same region w,

$$I_\xi(x,y) = \frac{\sum_{i,j=-w}^{w} I(x+i, y+j) \exp\left(-\frac{i^2+j^2}{4\xi^2}\right)}{\left[\sum_{i=-w}^{w} \exp\left(-i^2/4\xi^2\right)\right]^2}. \qquad (7)$$

Since both high- and low-pass filters are performed over the same region w, they are calculated simultaneously using the `bpass.m` function.[12] The function output is a new image, which we call `Ib`,

[12] Note that the Matlab code uses the feature *diameter* and not the radius!

```
>> Ib = bpass(I, 1, 10);
```

The three parameters of `bpass` are `image_array`, `lnoise`, and `lobject`, corresponding to the image, the noise correlation ξ and the region dimension w. View the filtered image with the command

```
>> imshow(Ib);
```

The output image, fig 6, shows the result of `bpass.m`. The image appears quite dim, since it now has a range of values between 1 and 51.2. An image scaled to the entire colormap can be viewed using

```
>> imshow(Ib,[0 max(max(Ib))]);
```

and saved by first scaling the image (assuming an 8-bit image) then writing the image file

```
>> Ibscaled = Ib./max(max(bcrop)).*255.;
>> imwrite(Ibscaled,gray,'ScaledImage.png');
```

WHEN SAVING FILES it is best to use a file format that does *not* use a lossy compression algorithm. The portable network graphics (PNG) format is a good choice, and TIFF files are also acceptable, as long as they do not use compression, such as JPEG. Matlab defaults to a uncompressed TIFF file, which leads to large file sizes.[13] Although Matlab will write JPEG format with a lossless compression algorithm, some operating systems, including Apple's OS X, do not read these files natively.

[13] A 1024×1024 array 12-bit ccd camera saves images on the order of 2MB. A 30 second video captured at 100 frames per second will yield about 6GB of uncompressed data.

A sequence of unscaled and scaled images using $2w = 5, 10$, and 20 are shown in fig 7. Notice that the brightness of the images increases as w increases. The brightness is given by the normalization constant

$$K_0 = \frac{1}{B}\left\{\sum_{i=-w}^{w} \exp\left[-(i^2/2\xi^2)\right]\right\}^2 - \left[B/(2w+1)^2\right] \qquad (8)$$

where $B = \left[\sum_{i=-w}^{w} \exp\left(-i^2/4\xi^2\right)\right]^2$.

Now that the background intensity variations and pixel-to-pixel noise have been filtered, we can proceed to finding the location of each particle.

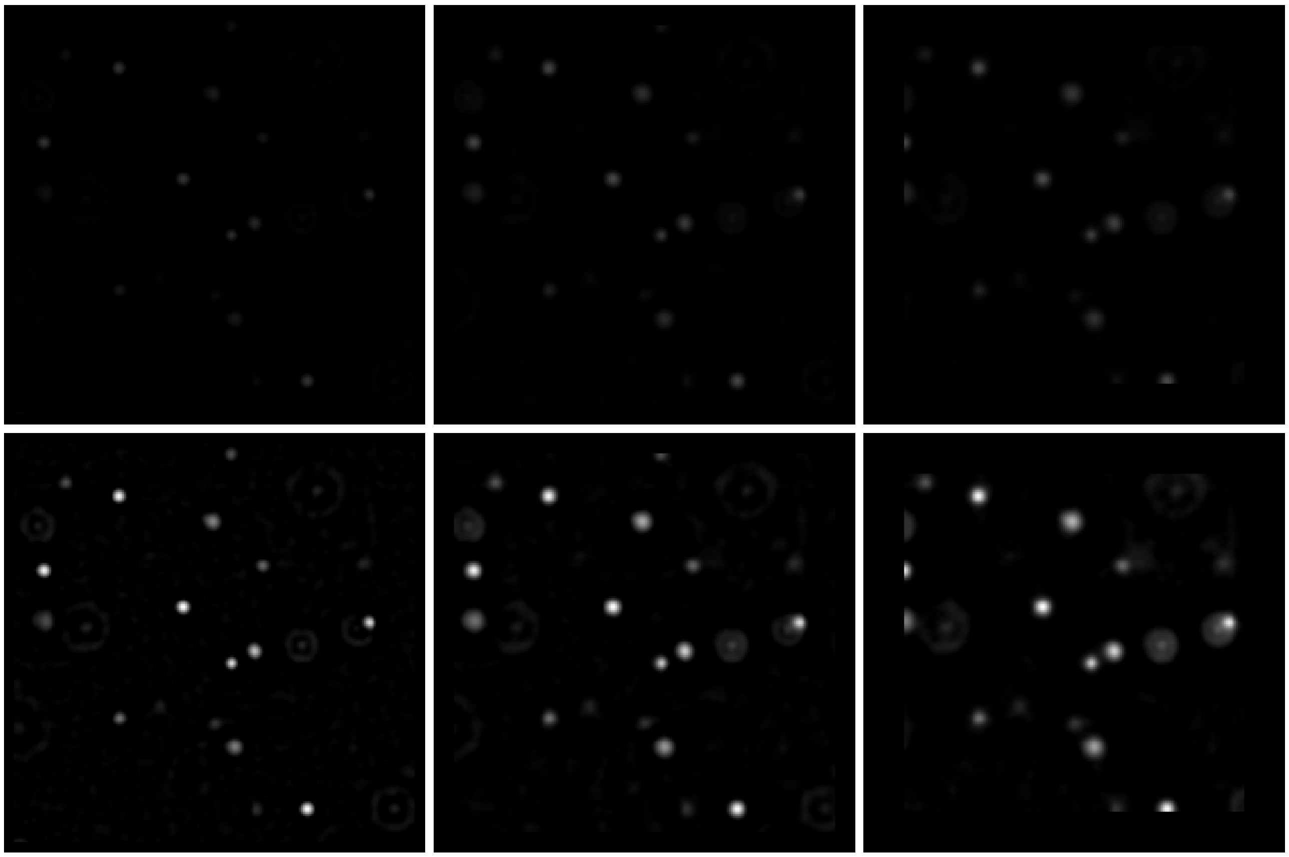
Figure 7: A sequence of unscaled (top row) and scaled (bottom row) images using $2w = 5, 10$, and 20 and $\xi = 1$.

Locate particles

Determining particle locations in an image follows two steps: identifying candidate particles by the brightest pixels, then refining these location estimates by calculating the centroid-weighted position.

First, locate the brightest pixels. A magnified image of a single particle is shown in Fig 8. Let $I(x, y)$ represent the intensity of the particle image. Remember, this isn't an "exact" image of the particle, but a convolution of the particle's fluorescence emission and the point spread function of the imaging system after applying the low- and high-frequency bandpass filters.

We first identify the locally brightest pixel for each particle, which serves as an initial estimate of the particle location, (x_0, y_0). The locally brightest pixels in the image are those with the highest values

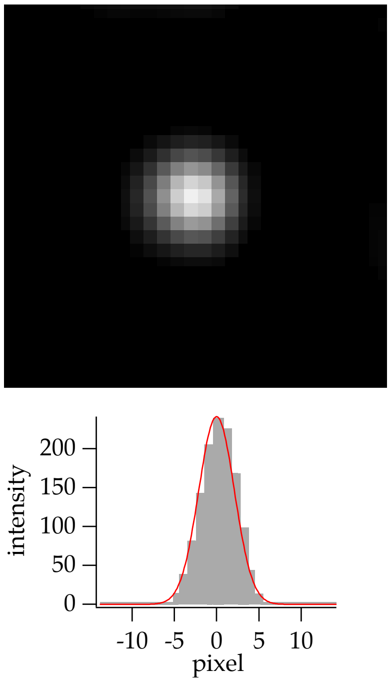

Figure 8: (Top) A magnified image of a single fluorescent particle after applying `bpass`. (Bottom) A histogram of the particle intensity and corresponding Gaussian fit.

within w pixels, where w is approximately the radius of the particle image and smaller than the average distance between particles. From here, most tracking routines only accept the brightest 30-40% of these pixels as particle candidates.

Referring to the magnified particle image in Fig 8 again, the brightest pixel is close to the image center, and to a reasonable approximation, the image I_i of this ith particle is two-dimensional Gaussian centered at the position $\mathbf{r}_i$,

$$I_i(\mathbf{r}) = I_0 \exp\left(-\frac{|\mathbf{r}-\mathbf{r}_i|^2}{s^2}\right). \tag{9}$$

The width of the intensity distribution in Fig 8 is $s = 2.98 \pm 0.04$ pixels. Notice that the brightest pixel isn't quite at the center of the fitted Gaussian distribution. It is located to the left by about 0.02 pixels. This offset reflects the fact that the center of the intensity *distribution* can be located more accurately than the maximum—which is at best within a half a pixel of the center—and serves as a method of refining the location of each particle to sub-pixel resolution.

To find the brightest pixels as candidate particles, we use the `pkfnd.m` function:

```
>> pk = pkfnd(Ib, 42, 10);
```

This function has three parameters `pkfnd(im,th,sz)`: `im` is the image to process, `th` is the threshold value for identifying the brightest pixels, and `sz` is an optional parameter for noisy data that should be set roughly to the diameter of the particle image.

A ROUGH ESTIMATE OF THE THRESHOLD VALUE can be determined by the maximum pixel value in the bandpass filtered image,

```
>> max(Ib(:))

ans =
   70.5293
```

A good estimate of the threshold value is about 60% of the maximum pixel value.

The `pkfnd` function above locates 28 candidate particles in our image,

```
>> whos pk
  Name          Size              Bytes  Class     Attributes

  pk           28x2                 448  double
```

The array consists of two columns, the initial positions of each particle (x_0, y_0). In fig 9 we overlay the values of `pkfnd` with the image `I`:

```
>> imshow(I,gray)
>> hold on
>> plot(pk(:,1),pk(:,2),'ro','MarkerSize',10,'LineWidth',2)
```

The first ten location estimates in this case are:

```
>> pk

pk =
    16   422
    40   442
    64    78
    90   562
   126   194
   127   330
   164   293
   176   247
   194   484
   196   348
```

Refine the initial location

The intensity weighted centroid surrounding the brightest pixel serves as a refinement for the particle location and provides *sub-pixel* resolution for particle tracking. For each particle, this correction is calculated as

$$\begin{pmatrix}\epsilon_x\\ \epsilon_y\end{pmatrix} = \frac{1}{m_0}\sum_{i^2+j^2\le w^2}\begin{pmatrix}i\\ j\end{pmatrix} I(x_0+i, y_0+j) \qquad (10)$$

where $m_0 = \sum_{i^2+j^2\le w^2} I(x_0+i, y_0+j)$ is the integrated brightness of the particle image. The refined particle location is

$$(x_i, y_i) = (x_0+\epsilon_x, y_0+\epsilon_y). \qquad (11)$$

Calculating the centroid location is accomplished using the `cntrd.m` function:

```
>> cnt = cntrd(Ib,pk,13);
```

The function has three input parameters, `cntrd(im,mx,sz)`. The parameter `im` is the image to process. The bandpass filtered image `Ib` is used here. The second parameter, `mx` are the locations of local maxima to pixel-level accuracy from `pkfnd.m`. The third parameter, `sz` is the diameter of the window used to calculate the centroid. The value of `sz` should be large enough to capture the entire particle, but not so large that it includes other nearby particles. The recommended size is the diameter used with `bpass` plus 2, `sz` = `2w + 2`, but `sz` must be an odd integer.

The `cntrd` function outputs four values for each particle: the x-coordinate, y-coordinate, brightness, and square of the radius of gyra-

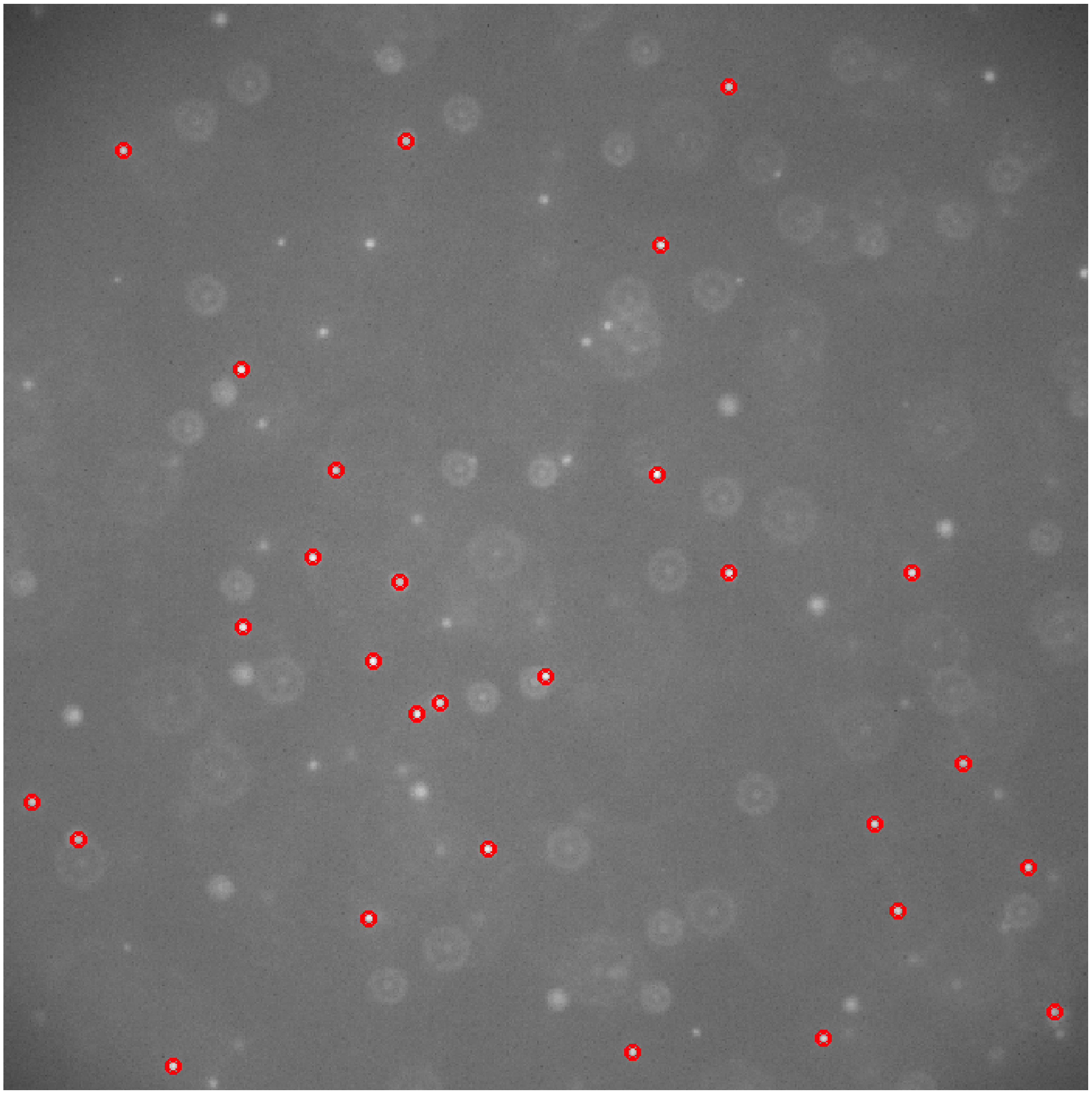

Figure 9: `pkfnd` output indicated by red circles on the original (unfiltered) image.

tion. Notice that not all particles are passed out of `cntrd.m`.

```
>> whos cnt
  Name          Size                 Bytes   Class       Attributes

  cnt         25x4                     800   double
```

Three particles are missing when we compare `cnt` to the original particle list `pk` produced by the `pkfnd.m` function. These particles are within 1.5× the value of `sz` from the edge of the image and are excluded from the final output. In fig 10 we replot the particle positions `cnt` on the original image in blue. The locations `pk` are still represented by red circles. Sample output for the first ten particles is listed below.

```
>> cnt

cnt =

   1.0e+03 *

    0.0397     0.4415     2.0651     0.0127
    0.0640     0.0780     1.8489     0.0093
    0.1263     0.1937     1.6106     0.0078
    0.1271     0.3301     2.0742     0.0085
    0.1641     0.2934     2.1422     0.0086
    0.1757     0.2471     1.3045     0.0086
    0.1945     0.4836     1.6830     0.0100
    0.1955     0.3480     2.2580     0.0095
    0.2099     0.3060     2.3052     0.0126
    0.2133     0.0724     2.2438     0.0126
```

The first row in `cnt` corresponds to the second row in `pk`. The original location $(40, 442)$ has been refined to $(39.7, 441.5)$.

Check for bias

An important check of the centroid location algorithm is to plot a histogram of the particle location corrections produced by `cntrd.m`. Once many particle positions are collected over multiple frames, the histogram should be examined. For instance, with the output array `pos_lst` where the x- and y-values of the centroids are `pos_lst(:,1)` and `pos_lst(:,2)`, respectively, the command is

```
>> hist(mod(pos_loc(:,1),1),20);
```

which produces a histogram of the x-positions modulo 1. The histogram should look flat, which ensures there is no bias in the position correction. We can also plot a histogram of the x- and y-positions modulo 1:

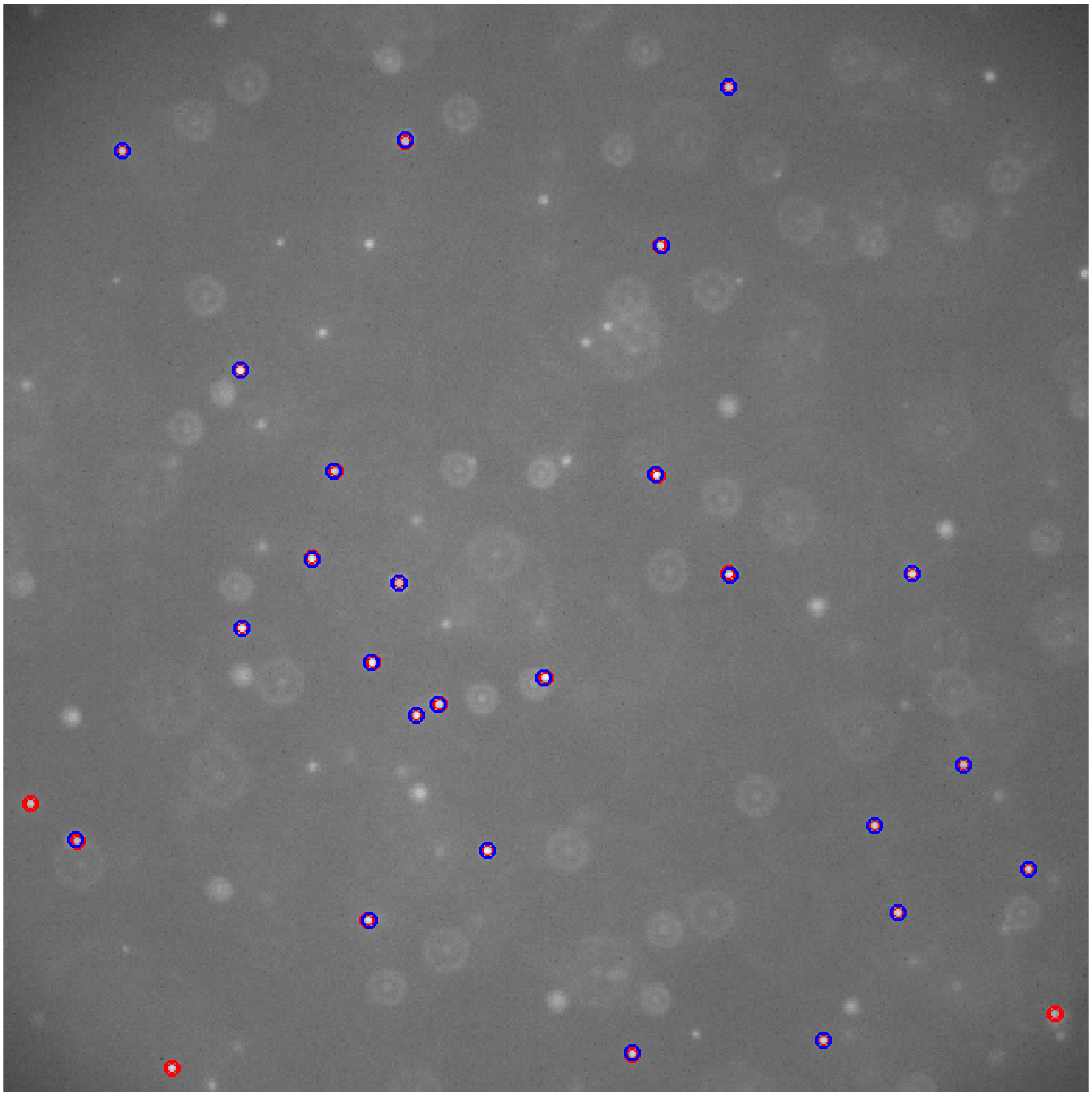

Figure 10: `cntrd.m` output indicated by blue circles on the original (unfiltered) image. The initial location estimates produced by `pkfnd.m` are shown in red.

```
>> hist(cat(1,mod(pos_lst(:,1),1),mod(pos_lst(:,2),1)),20);
```

or using a shortened form of the command

```
>> hist([mod(pos_lst(:,1),1);mod(pos_lst(:,2),1)],20);
```

This plot is shown to the right. A common failure is to have peaks in the histogram near 0 and 1 and a dip at 0.5. This pattern appears when the feature size is made too small, causing the x- and y-coordinates to round off to the nearest integer value. Fig 11 shows the centroid remainders for 10^4 particle positions. To generate the histograms, use the following commands:

```
>> [n, xout] = hist([mod(pos_lst_9(:,1),1);
                     mod(pos_lst_9(:,2),1)],20);
>> stairs(xout, n,'color',[0.5 0.5 0.5])
>> ylim([0 1600])
>> box off
```

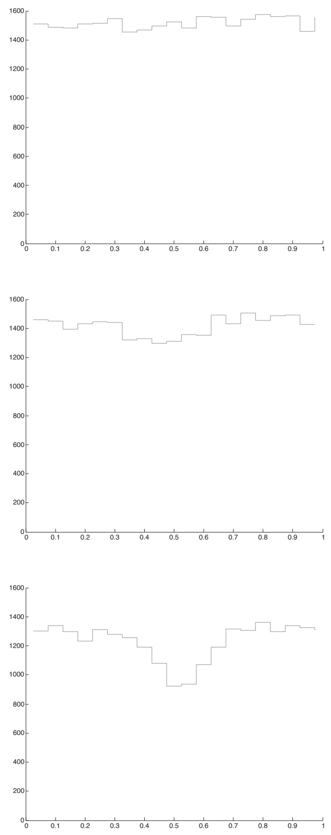

Figure 11: Histogram of x- and y-positions modulo 1 is used to discern bias in the centroid location algorithm. The top histogram was generated using a value of the feature size in pixels, `w = 11`, the middle using `w = 9`, and the bottom histogram using `w = 7`. There are 30,354 positions in each histogram.

Creating particle trajectories

Locating particles in each frame is the first step of particle tracking. The next step is to identify the same particle in neighboring frames so a trajectory can be assembled as a function of time. Because particle can diffuse into and out of the imaging plane, each frame may contain a different number of particles.

The key particle tracking functions introduced here are:

`Ftrack.m` – Call the `bpass`, `pkfnd` and `cntrd` functions and generate the (unsorted) position list of particles in each frame. The position list will be used by `track` to create particle trajectories.

`track.m` – Constructs trajectories from unsorted list of particle positions, i.e. the output of `Ftrack.m`

The previous steps showed us how to locate particles in each image frame of video sequence. The frames, if they are acquired at regular intervals[14], should act as a time stamp. Next we will use the particle location routines for a series of images that make up a video sequence. This is a two-step process of locating the particles in each frame, followed by connecting the particle positions in adjacent frames to create particle trajectories in time.

[14] Verify that the image acquisition is not skipping frames or acquiring images at uneven intervals.

Generate particle positions in all frames

First, we will process all of the frames in an image sequence using the routine `Ftrack.m`. This function creates a concatenated list of particle positions for each frame. `Ftrack` handles reading the image files successively and calling the functions `bpass`, `pkfnd`, and `cntrd`, corresponding to the image bandpass filter, peak finding routine, and centroid correction, respectively. The image sequence is assumed to be composed of individual files in the working directory with the filename `fileheadxxxx.tif`. The `Ftrack(filehead,first,last,feature)` function has four input parameters. The parameters `first` and `last` correspond to the image frame to start and end with. The parameter `feature` is the expected size (diameter) of the particle image, the same feature size that is used in `bpass`, `pkfnd` and `cntrd`. Note that `feature` needs to be odd. The function will return an error if it is not.

The video sequence we will analyze consists of 400 frames, saved as uncompressed tiff files, with the header "`frame`." Here is the output of the function. [15]

[15] It is interesting to adjust the threshold value to change the number of particles tracked.

```
>> poslist = Ftrack('frame',1,400,11);
Frame number: 10
Frame number: 20
Frame number: 30
Frame number: 40
Frame number: 50
...
Frame number: 390
Frame number: 400
>>
```

The output, `poslist`, is a matrix with five columns.

```
>> whos poslist
  Name              Size             Bytes  Class     Attributes

  poslist       15177x5             607080  double
```

Here are the first five rows of `poslist`.

```
>> format shortG
>> poslist(1:5,:)

ans =

      153.66       535.81   3.6716e+05       9.0154            1
      165.05       568.53   4.4652e+05       13.243            1
      193.36       702.24    3.001e+05       8.6519            1
      255.68       276.72   3.1019e+05       9.3801            1
      285.72       737.54   4.6125e+05       9.5741            1
```

The five output columns are the *x-position*, *y-position*, m_0 and m_2 values, followed by the *frame number*. The first and second moments, m_0 and m_2 are useful for evaluating the particle location results. For instance, the user may limit particles within specific value ranges of m_0 and m_2 as part of a cluster analysis, since these parameters tend to cluster systematically, as discussed by Crocker and Grier.

Save the `poslist` data with the command

```
>> dlmwrite('poslist_water_101fps',poslist)
```

Check the tracking results with the function `Ftrack_movie`. The input parameters are identical to `Ftrack`, but the routine generates the image with circles representing the `pkfnd` and `cntrd` results. Notice how the selection of variables such as the `threshold` value in `pkfnd` and the feature size affect the tracking results. You will see a number of particles “blinking” on and off as they move in and out of the focal plane. This out-of-plane movement has important implications when creating particle trajectories, as we discuss next.

Link particle trajectories

The final step is to link particle positions in each frame into trajectories as a function of time. The `track` routine accomplishes this by identifying the same particle in adjoining frames. To use `track(xyzs,maxdisp,param)`, we need two required parameters and one optional parameter. The first, `xyzs`, is the particle position data in the format:

```
           (x)          (y)           (frame)
     pos = 3.60000      5.00000       0.00000
           15.1000      22.6000       0.00000
           4.10000      5.50000       1.00000
           15.9000      20.7000       2.00000
           6.20000      4.30000       2.00000
```

Note that `pos(0:d-1,*)` contains the d coordinates and data for all the particles, at the different times and must be positive, while `positionlist(d,*)` contains the time t that the position was determined, and must be an integer (e.g. a frame number.) These values must monotonically increase and be uniformly gridded in time.

To prune down `poslist`, we can use the command

```
>> pos = poslist(:,[1:2,5]);
```

to include on the x and y positions and frame number. However, this is not entirely necessary, since the optional `param` can be used to pass additional information to the `track` routine that allows the other information about the particles to be sorted as well. More about that in a bit.

The second required parameter, `maxdisp`, is an estimate of the maximum distance that a particle moves in a single time interval. This enables particles to be associated in one frame to another and is based on the random thermal motion of non-interacting particles. While many systems of interest may not precisely fit this model of motion, the algorithm generally works well. However, it will not perform well for particles moving in a strong convection. Other methods are needed in that case.

One very important thing to recognize is that if a particle moves out of the focal plane and returns some time later, it will be given a different identifier and be treated as a separate particle. However, in some tracking software it is possible to specify the number of frames a single particle can be "missing" and still retain its identifying number. In the sample data, particle 1 is not tracked in frame 3. It's location is missing, but the particle is tracked again in frames 4 and 5. Linking a particle trajectory across skipped frames has the advantage of generating longer trajectories, which improves the statistics for longer lag times.

Finally, there is the optional `param` structure, which has four fields, which are assigned to four function variables:

```
memory_b      =    param.mem;
goodenough    =    param.good;
dim           =    param.dim;
quiet         =    param.quiet;
```

If the parameters are not specified, then they are assigned default values.

The variable `memory_b` is the number of time steps a particle can be skipped before it is assigned a new identifier. This is useful for maintaining trajectories of particles that move briefly out of the focal plane and reduces the degree of trajectory truncation. The default setting is zero. The second variable, `goodenough` limits trajectories to those with values equal to or greater than its value. The default value is two. This eliminates "noise" trajectories. The boolean variable `quiet` eliminates text output from the routine when it is set to 1.

Set the parameters for `track` using the command:

```
>> param = struct('mem',0,'good',0,'dim',2,'quiet',0)

param =

      mem: 0
     good: 0
      dim: 2
    quiet: 0
```

This creates a `structure` with the four required fields. Here we are using the default values. Then pass `param` to the function,

```
>> linked = track(pos,10,param);
50 of400 done.  Tracking41 particles158 tracks total
100 of400 done.  Tracking38 particles126 tracks total
150 of400 done.  Tracking36 particles138 tracks total
200 of400 done.  Tracking28 particles147 tracks total
250 of400 done.  Tracking31 particles105 tracks total
300 of400 done.  Tracking34 particles139 tracks total
350 of400 done.  Tracking37 particles114 tracks total
400 of400 done.  Tracking39 particles147 tracks total
>>
```

The total number of trajectories generated are (assuming a 4-column output)

```
>> max(linked(:,4))

ans =

   814
```

The number of trajectories generated for this data set is 814! Clearly there is some movement of particles into and out of the focal plane. Let's look at the distribution of trajectory lengths. The `tabulate` command is useful for this. For every particle identified in column 4, `tabulate` will return the number of positions returned and the frequency in the overall distribution.

```
>> tabulate(linked(:,4))
   Value    Count    Percent
       1       42      0.28%
       2        2      0.01%
       3        9      0.06%
       4       35      0.23%
       5       34      0.22%
       6        8      0.05%
       7        4      0.03%
       8        6      0.04%
       9       20      0.13%
      10       13      0.09%
     ...
```

We can plot a histogram of the trajectory lengths,

```
>> traj_length = tabulate(linked(:,4))
>> [n, xout] = hist(traj_length(:,2),20);
>> stairs(xout, n,'color',[0.5 0.5 0.5]);
>> box off;
```

which is shown in fig 12.

There are two things we can do at this point. We can accept "skipped" frames and link particle candidates across several frames, which will reduce the number of independent trajectories. This is accomplished using the `memory` variable. There is a drawback, however, in that our data will include trajectories with skipped time steps. We have to be careful when calculating the trajectory statistics, such as the mean-squared displacement, to take into account the fact that sequential particle positions may be separated by more than one time step. However, a key advantage for doing this is that the statistics at longer lag times will improve by reducing trajectory truncation.

Second, we can limit the output to trajectories of a certain length by specifying a value of `goodenough`. This parameter ensures that all trajectories included in the results have a minimum length. Giving `goodenough` a value of 2 or 3 will help reduce the overhead on the tracking routine—there is no sense having a trajectory with only one

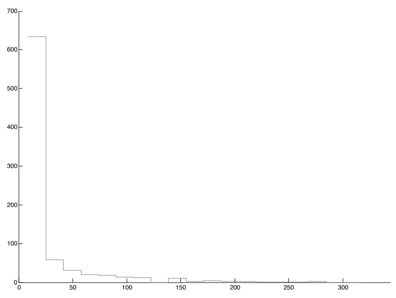
Figure 12: Histogram of trajectory lengths. The vast majority are short trajectories due to particle movement in and out of the focal plane.

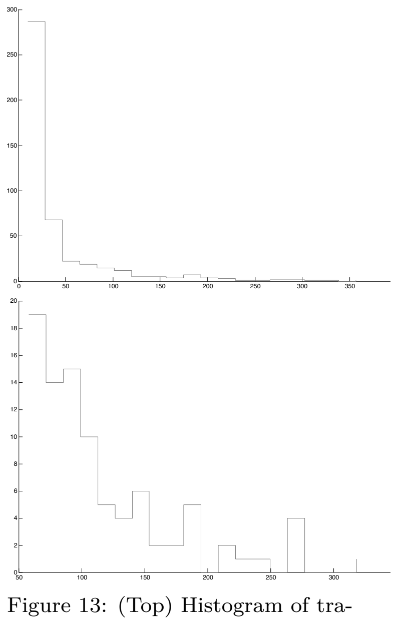
Figure 13: (Top) Histogram of trajectory lengths with mem = 2 and goodenough = 0. (Bottom) Trajectory lengths with mem = 0 and goodenough = 20.

position if we are to calculate the displacement of the particles. Such "single position trajectories" occur when particles are close to the edge of the imaging plane and move randomly into and out of the tracking threshold.

Changing `mem` to 2 reduces the number of short trajectories, as shown in fig 13. The second histogram shows the results of particle tracking with `mem = 0` and `goodenough = 20`, which truncates the data even further.

Now we have an array of particle positions as a function of time with increasing particle number. Use the last column to find the number of particles tracked. Obviously this is larger than the total number of particles in the frame at any time due to trajectory truncation. (How does this value change if `memory_b` is increased? What is the longest trajectory? What is the shortest trajectory? What are the trajectory statistics?

Trajectory analysis

Calculating the MSD

The function `msd = MSD(linked)` calculates the MSD for non-overlapping lag times. Using the raw input from the previous functions, the output will be the mean-squared displacement in pixels^2 versus the lag time in frames. The third column is the number of observations for each row. Shorter lag times have a larger number of observations. These decrease initially as the reciprocal of the number of frames. At longer lag times, trajectory truncation causes the number of observations to decrease even more quickly. Calibration is required to express these in terms of the real displacement, and the frame rate is needed to convert the lag time to a real time.

`MSD.m` will assume that `linked` consists of four columns: the x- and y- positions, the frame number and the particle ID. If `tracked` was passed along with m_0 and m_1, then it can be pruned down using the command

```
>> linked2 = linked(:,[1:2,5:6]);
```

assuming that columns 3 and 4 contain the extraneous data.

To summarize, calculate the MSD with the command:

```
>> msd = MSD(linked);
```

and plot it with

```
>> loglog(msd(:,1),msd(:,2:3))
```

The third column is the number of observations per data point (lag time).

Exporting the data. The MSD data can be exported for plotting using a graphics program. Use the command:

```
>> dlmwrite('msd_cmos_water_101fps_roi2048',msd);
```

Matlab code

What follows is Matlab code for the following functions:

`Ftrack.m` – Call the `bpass`, `pkfnd` and `cntrd` functions and generate the (unsorted) position list of particles in each frame. The position list will be used by `track` to create particle trajectories.

`MSD.m` – Take trajectory data from `track.m` and calculate the MSD.

`SNR.m` – Analyze the signal to noise ratio for an image.

`vignette.m` – Called by `SNR.m` to create masks

These functions are listed in the usual order in which they are called.

Ftrack.m

```
function pos_lst=Ftrack(filehead,first,last,feature)
% out=cntrd(im,mx,sz,interactive)
%
% PURPOSE:
%          For each image in a directory
%             Read image file
%              Filter the image
%              Find brightest pixels
%              Find pixel centroids
%              Concatenate positions to pos_lst
%
%
% INPUT:
% filehead: A string common to the image file names
%          i.e. we assume that the filename is of the form 'framexxxx.tif'
%
% start:    First frame to read
% end:      Final frame to read
%
% feature:  Expected size of the particle (diameter)
%
% NOTES:
%
% OUTPUT:
%
% CREATED: Eric M. Furst, University of Delaware, July 23, 2013
%  Modifications:

if mod(feature,2) == 0
    warning('feature size must be an odd value');
    out=[];
    return;
end

pos_lst=[];
for frame=first:last

    if mod(frame,50)==0
        disp(['Frame number: ' num2str(frame)]);
    end

    % read in file
    image = double(imread([filehead, num2str(frame,'%04u'),'.tif']));

    % Bandpass filter
    imagebp = bpass(image,1,feature);

    % Find locations of the brightest pixels Need to change the 'th'
    % (threshold) argument to something that is specified or determined.
    % Current value is 8000. A rough guide is 0.6*max(imagebp(:))
    pk = pkfnd(imagebp, 8000, feature);
    % Refine location estimates using centroid
    cnt = cntrd(imagebp, pk, feature+2);
    % cntrd can also accept "interactive" mode
    % cnt = cntrd(imagebp, pk, feature+2, 1);

    % Add frame number to tracking data. Use 0 as the reference frame.
    cnt(:,5) = frame;

    % Concatenate the new frame to the existing data
    pos_lst = [pos_lst, cnt'];

end

% Format the position list so that we have four columns: x, y, m_0, m_2, frame
pos_lst = pos_lst';

% It's better to separate the particle tracking from the trajectory
% analysis. Once the particles are located in each frame, we can re-run the
% trajectory anslysis using different parameters.
%
% After generating the unsorted position lists, run track.m to generate the
% particle trajectories. The routine takes the following input:
%      For the input data structure (positionlist):
%            (x)          (y)          (t)
%      pos = 3.60000      5.00000      0.00000
%            15.1000      22.6000      0.00000
%            4.10000      5.50000      1.00000
%            15.9000      20.7000      2.00000
%            6.20000      4.30000      2.00000
%
% Use the command:
%   trajectories = track([pos_lst(:,1:2),pos_lst(:,5)],10);
```

MSD.m

```
function msd = MSD(trackdata)
%
% PURPOSE:
%           Calcuate the MSD from the output of track.m
%
%
% INPUT:
% trackdata: Trajectory data of the form:
%           x-position     y-position    frame     particle i.d.
%
% NOTES: The particle i.d. is assumed to increment from 1 to a maximum
% number of particles. For each particle i.d. the frame number is assumed
% to increase. Trajectories can include skipped frames.
%
% OUTPUT:
% MSD: will have three columns:
%       Column 1: lag time (in frames)
%       Column 2: MSD (in pixels)
%       Column 3: number of observations in average
%
% CREATED: Eric M. Furst, University of Delaware, September 8, 2013
%  Modifications:

MSD_list = [];

% Step through all particles in the tracked data set
for(particleid=1:max(trackdata(:,4)))

    % Find the starting row of the particleid in trackdata
    % This is the offset
    index=find(trackdata(:,4)==particleid,1);

    % Find the number of frames for the particle
    % NOTE that this is NOT equivalent to time steps of 1 if the data has
    % "skipped frames"
    totalframe = sum(trackdata(:,4)==particleid);

    % The maximum frame separation is totalframe - 1
    max_step = totalframe - 1;

    % Only analyze if there is more than one frame to calculate
    % displacement
    if max_step >= 1
        disp(['Particle ', num2str(particleid),' of ',...
          num2str(max(trackdata(:,4))),'. Total frames: ',...
                                                num2str(totalframe)]);
        % Step through all frame separations starting from 1 up to max_step
        for step=1:max_step
%        Remove comment for debugging
%        disp(['Number of steps of length ', ...
%           num2str(step), ': ',num2str(fix(totalframe/step))]);
            for j=1:(fix(max_step/step))
            % Caculate lag time and mean-squared displacement between steps
               delta_t = trackdata(index+j*step,3)...
                   - trackdata(index+(j-1)*step,3);
               delta_r2 = (trackdata(index+j*step,1)...
                   - trackdata(index+(j-1)*step,1))^2 ...
                   + (trackdata(index+j*step,2)...
                   -  trackdata(index+(j-1)*step,2))^2;
%           Remove comment for debugging
%           disp(['  Step ', num2str(j),' of ',...
%               num2str(fix(totalframe/step)), ' Delta t = ',...
%               num2str(delta_t), ' Delta r^2 = ',num2str(delta_r2)]);

             MSD_list = [MSD_list, [delta_t,delta_r2]'];
            end
        end
    end
end

MSD_list = MSD_list';


% Build the MSD from the MSD_list
% the MSD will have three columns:
%   Column 1: lag time (in frames)
%   Column 2: MSD (in pixels)
%   Column 3: number of observations in average

msd = [];
min_lag = min(MSD_list(:,1));
max_lag = max(MSD_list(:,1));
for lag=min_lag:max_lag
    number_obs = sum(MSD_list(:,1)==lag);
    if(number_obs>=1)
        ind = find(MSD_list(:,1)==lag);
        msd = [msd, [lag mean(MSD_list(ind,2)) number_obs]'];
    end
end

msd = msd';

% Run with
% >>msd = MSD(result)
% Plot with
% >>loglog(msd(:,1),msd(:,2:3))


end
```

SNR.m

```
function [signal,noise]=SNR(image,feature,maskradius)
% [signal,noise]=SNR(image,feature,maskradius)
%
% PURPOSE: Calculate the SNR for a single image. We calculate the
% distribution of pixels that consist of our signal and the noise. The
% signal are the pixels associated with tracked particles. The noise
% consists of all pixels that are not signal. We do this by masking the
% tracked particles (assigning their region with a negative value). Note
% that the signal area is two pixels smaller than the feature size and the
% noise area is two TIMES feature size larger. This is to prevent the edges
% of particles from contributing to the signal or noise.
%
% Load the image with command: I = double(imread('test.tif'));
%
%
%
% INPUT:
%
% image: The image to analyze
%
% feature:  Expected size of the particle (diameter) that constitutes the
% signal region. A value 5 seems to work well.
%
% OPTIONAL INPUT:
%
% maskradius: If an image is vignetted, this is the radius
% of the image to use in calculations. Otherwise, this defaults to a value
% 450. A maskradius equal to -1 will ignore the vignette.
%
% NOTES:
%
% OUTPUT:
% signal: A vector of pixel values of the tracked particles
% noise: A vector of pixel values of the background image excluding tacked
% particles
%
% CREATED: Eric M. Furst, University of Delaware, July 25, 2013
%  Modifications:

if mod(feature,2) == 0 || feature < 3
    warning('Feature size must be an odd value >= 3.');
    signal=[];
    noise=[];
    return;
end

% Set the maskradius to a default value
if nargin==2
   maskradius=450;
end
```

```
% read in file
% image = double(imread(filename));
% Bandpass filter
imagebp = bpass(image,1,feature);

% Find locations of the brightest pixels Need to change the 'th'
% (threshold) argument to something that is specified or determined.
% A rough guide is 0.6*max(imagebp(:))
th = 0.6*max(imagebp(:));
pk = pkfnd(imagebp, th, feature);
disp(['Maximum imagebp value is ', num2str(max(imagebp(:)))]);
disp(['Using values > ', num2str(th)]);
disp(['located ' num2str(size(pk,1)) ' particles.']);

% Now we have particle positions. Need to mask the image at these
% locations. Make the particle mask. This is an array with -1 for
% values within a radius of half the feature size.
sz = 2*feature+1;
r = (sz-1)/2;
[x, y]=meshgrid(-(r):(sz-r-1),-(r):(sz-r-1));
particlemask=((x.^2+y.^2)<=r^2);
particlemask=((x.^2+y.^2)>r^2)-particlemask;

% go through list of particle locations
% make a circle of radius feature at each location with value -1
imagenoise = image;
for i=1:size(pk,1);
    x = pk(i,1);
    y = pk(i,2);
    imagenoise((y-r):(y+r),(x-r):(x+r))=...
        image((y-r):(y+r),(x-r):(x+r)).*particlemask;
end

% Make the mask for the signal
sz = feature-2;
r = (sz-1)/2;
[x, y]=meshgrid(-(r):(sz-r-1),-(r):(sz-r-1));
particlemask=((x.^2+y.^2)<=r^2);
particlemask=((x.^2+y.^2)>r^2)-particlemask;
```

```
% go through list of particle locations
% make a circle of radius feature at each location with value -1
imagesignal = image;
for i=1:size(pk,1);
    x = pk(i,1);
    y = pk(i,2);
    imagesignal((y-r):(y+r),(x-r):(x+r))=...
        image((y-r):(y+r),(x-r):(x+r)).*particlemask;
end

imagesignal = -imagesignal;

% Finally, mask the image to exclude vignette regions
imagenoise = vignette(imagenoise,maskradius,-1);
imagesignal = vignette(imagesignal,maskradius,-1);

% Now that we have maskimage, let's calculate the noise
% Statistics of all regions with positive values > 0
index = find(imagenoise>0);
noise = image(index);
index = find(imagesignal>0);
signal = image(index);

imagesc(imagenoise./max(imagenoise(:)));
disp(['Mean signal: ', num2str(mean(signal))]);
disp(['Mean noise: ', num2str(mean(noise))]);
disp(['Standard deviation noise: ', num2str(std(noise))]);
disp(['Signal-to-noise: ', num2str((mean(signal)-mean(noise))/std(noise)),...
    '(',num2str(10*log10((mean(signal)-mean(noise))/std(noise))),' dB)' ]);

end
```

vignette.m

```
function masked_image = vignette(image,radius,value)
% out=cntrd(im,mx,sz,interactive)
%
% PURPOSE:
%          Mask a vignetted image by setting the locations outside of a
%          radius equal to value. The default value is 0.
%
%
% INPUT:
% image:   The original image
% radius:  The radius of the desired vignette in pixels. If the radius is
%          -1, just return the original image.
% value:   Set to value. OPTIONAL. Default to 0.
%
% NOTES:
%
% OUTPUT:  A masked image
%
% CREATED: Eric M. Furst, University of Delaware, July 23, 2013
%  Modifications:


if nargin==2
   value=0;
end

if radius<0

    masked_image = image;

else
    % Make mask
    % Get the size of image
    %
    [ix, iy] = size(image);

    % cx and cy are the center coordinates of the circle
    cx=ix/2;
    cy=iy/2;

    %
    [x,y]=meshgrid(-(cx-1):(ix-cx),-(cy-1):(iy-cy));
    c_mask=((x.^2+y.^2)<=radius^2);
    mask_outside = (c_mask-1)*(-value) + c_mask;

    masked_image = image.*c_mask + mask_outside;
end
```

Notes on the routine `cntrd.m`

First, we basically make a mask with radius `r = (sz + 1)/2`. Here, `sz = 11`. The variable `ind` was created by making a similar array (`dst`) and using the values of the radius at each of the matrix entries.

```
>> msk(ind)=1.0;
>> msk

msk =

     0     0     0     0     1     1     1     1     0     0     0     0
     0     0     1     1     1     1     1     1     1     1     0     0
     0     1     1     1     1     1     1     1     1     1     1     0
     0     1     1     1     1     1     1     1     1     1     1     0
     1     1     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1     1     1
     0     1     1     1     1     1     1     1     1     1     1     0
     0     1     1     1     1     1     1     1     1     1     1     0
     0     0     1     1     1     1     1     1     1     1     0     0
     0     0     0     0     1     1     1     1     0     0     0     0
```

Notes on Matlab

Image processing

For help: `help images`

Help topic	Description
images/colorspaces	- Image Processing Toolbox — colorspaces
images/images	- Image Processing Toolbox
images/imuitools	- Image Processing Toolbox — imuitools
images/iptformats	- Image Processing Toolbox — File Formats
images/iptutils	- Image Processing Toolbox — utilities
images/imdemos	- Image Processing Toolbox — demos and sample images

Helpful image processing commands

Note that a grayscale image can be either `single` or `double` but must have values between 0 and 1. There are three commands to display an image, depending on the data type (image or array): `imshow`, `image`, or `imagesc`.

Command	Description
`I = imread('SampleImage.tif');`	read an image
`imshow(I)`	show image (native type)
`I = double(imread('SampleImage.tif'));`	read an image as type `double`
`image(I);`	display array as image (use for `double` types)
`imagesc(I);`	scale data and display as image
`imwrite(I,'filename.png')`	write an image file. Format set by extension.
`imhist(I);`	produce a histogram of the image `I`
`max(max(b))`	maximum value of matrix `b`
`max(b(:))`	alternate maximum value of matrix `b`
`histc(I(:),255:255)`	count the number of saturated pixels in an 8-bit image
`deconvblind`	Deblur image using blind deconvolution
`figure`	Create a new figure window. Select using `figure(1)`.

Image processing used in particle tracking

Command	Description
`imagesc`	Scale data and display as image
`imwrite(b,gray,'test.tif')`	Write matrix `b`, where `b = bpass(image,1,15)`

The "Image enhancement and analysis" help page provides some steps for flattening an image background (this could be helpful for analyzing the images of riso pasta.)

Image statistics

Command	Description
`mean(I(:))`	Calculate the mean value of an image
`std(I(:))`	Calculate the standard deviation of an image

Plotting commands

Command	Description
`plot(X, Y, params)`	Plot values
`plot(pos_lst(:,6),pos_lst(:,5),'.','Color',[0.5 0.5 0.5])`	Plot with gray dot symbol
`hist(data, bins)` or `hist(data)`	Create a histogram of `data`
`h = findobj(gca,'Type','patch');`	Change the color of a histogram
`set(h,'FaceColor',[0.5 0.5 0.5],'EdgeColor','w')`	
`box off`	Remove the plot frame
`[n,xout] = hist(...)`	returns vectors `n` and `xout` containing the frequency counts
`bar(xout,n)`	and the bin locations. Plot with the `bar` command or
`stairs(X, Y, 'line spec');`	the `stairs` command.
`set(gca, 'YScale', 'log')`	set logarithmic scale

Basic Matlab

When using Matlab matrices, remember that Matlab uses a (row, column) format.

Make a matrix with dimensions *row* × *column*:

```
>> matrix =
```

Loading a tab-delimited data file, such as the output to a particle tracking routine:

```
>> foo = dlmread('filename');
```

To write a delimitated file,

```
>> dlmwrite('filename', M, 'delimiter');
```

The default —

Given the 7-column vector `T`,

```
>> whos T
  Name            Size                  Bytes  Class     Attributes

  T            110943x7               6212808  double
```

select the first column using

```
>> T(:,1)

ans =

   493.4690
   493.4720
   493.5570
   493.5880
   493.1810
   493.5850
   493.3890
   493.1830
   493.2670
   493.2690
   ...
```

or a subset of the column,

```
>> T(1:10,1)
```

Size of matrix Use the size command.

```
>> size(back,1)

ans =

    92
```

For instance, to display the number of data points collected (here backscatterdist is a matrix with columns t, x, y, and z)

```
disp(['Number of backscattered trajectories: ', ...
    num2str(size(backscatterdist,2))]);
```

Display format

The command `format` sets the output format. The default setting is `short`. See `help format` for additional formats. Below is the difference between `short` and `shortG`

```
>> format short
>> foo

foo =

   1.0e+03 *

    0.0397    0.4415    2.0651    0.0127
    0.0640    0.0780    1.8489    0.0093
    0.1263    0.1937    1.6106    0.0078
    0.1271    0.3301    2.0742    0.0085
    0.1641    0.2934    2.1422    0.0086
    0.1757    0.2471    1.3045    0.0086
    0.1945    0.4836    1.6830    0.0100
    0.1955    0.3480    2.2580    0.0095
    0.2099    0.3060    2.3052    0.0126
    0.2133    0.0724    2.2438    0.0126

>> format shortG
>> foo

foo =

         39.7         441.5        2065.1          12.7
           64            78        1848.9           9.3
        126.3         193.7        1610.6           7.8
        127.1         330.1        2074.2           8.5
        164.1         293.4        2142.2           8.6
        175.7         247.1        1304.5           8.6
        194.5         483.6          1683            10
        195.5           348          2258           9.5
        209.9           306        2305.2          12.6
        213.3          72.4        2243.8          12.6

>>
```