Supplemental Material

Supplemental Movie 1. Nascent RNA at transcription sites in monkey nuclei, imaged using a wide-field microscope and CCD (90-nm pixels) as in Fig. 5(b). Each frame in the movie shows one of 100 windows (15x15 pixels) with a $S{:}N < 3$ (value indicated in the lower-left corner), and the 2-D localizations obtained by the different methods (tuned version of JD – *red circle*; CM – *blue dot*; MLS – *orange dot*; MLE – *green star*). Each window is shown at low magnification in the lower-right corner to give an impression of its appearance at a typical scale. The tuned version of JD performs at least as well as the others.

Supplemental Figure 1. The failure of MLS at low photon counts.
Using computer-generated images (15x15 pixels; 10-250 photons; $b = 0$; inset shows a typical image generated using 63 photons to give a $S{:}N = \infty$), the 1-D RMSE was calculated (10^3 localizations per data point) using MLS, and the fraction of times that the algorithm converged to a solution during the 200 iterations allowed determined. As the number of photons declines, MLS increasingly fails (as it becomes more susceptible to shot-noise that causes images to diverge from a Gaussian pattern). The onset of rapid decline coincides with the rapid increase in RMSE seen in Fig. 2.

Supplemental Figure 2. Tuning JD reduces localization error at low signal-to-noise ratios, but increases error at high ratios.
Using computer-generated images (15x15 pixels; 10-10,000 photons; $b = 10$), the 1-D RMSE (in nm or pixel units) was calculated (10^4 localizations per data point) using JD tuned by varying either μ_i alone, σ_i alone, or both together (as in Figs. 3(a), 3(b), and 3(c)). Errors given by CM are included for comparison, as they are identical to those given by the default version of JD. *Left*: At high $S{:}N$, tuning σ_i alone yields increased error, while tuning μ_i alone does not. *Right*: At high $S{:}N$, tuning both μ_i and σ_i yields high error that converges asymptotically to ~0.02 pixel.

Supplemental Figure 3. The effects of window size and the distance of a spot from the center of the window.
Using computer-generated images (183 photons; $b = 10$; $S{:}N = 2$), the 1-D RMSE (in nm or pixel units) was calculated using the tuned version of JD, CM, MLE, and MLS. (**a**) Varying window size. When the window chosen for analysis decreases from 15x15 to 7x7 pixels, errors given by the tuned version of JD are essentially invariant (i.e., the method performs robustly), but those given by CM and MLE fall. As errors given by MLE are known to increase as window size falls in the absence of background noise and at higher a higher $S{:}N$ [7], the fall seen here with both MLE and MLS is probably due to an increased likelihood of finding the true location purely by chance. Lines are data points fit to exponential functions. (**b**) Varying the distance of a spot from the center of the window (*grey lines* indicate error at farthest point). As the true location relative to the center of a window varies, the variance given by JD is negligible, but MLE and CM are most accurate when the spot is at the image center (a known behavior of CM in noisy images [10]). Curiously, MLS seems most accurate at this low $S{:}N$ when the spot is away from the image center.

Supplemental Figure 4. dSTORM images of microtubules in monkey cells (cos-7) processed using the different approaches.

The 30,000 images ('frames') of diffraction-limited spots of one field (collected using inclined illumination and a CCD) used for Fig. 5(b) were analyzed (some images and histograms from that figure are included here for comparison). (**a**) Window selection, and corruption with noise. (i) Mean projection of all spots; this projection yields an image analogous to that obtained using a wide-field microscope. (ii) A typical frame, illustrating individual diffraction-limited spots, from which windows that contained 1 centrally-located and isolated spot were selected (using a Gaussian spot-finding algorithm) for localization. (iii) Four isolated spots (11x11 pixel windows) selected by the algorithm from frame 30 (the *histogram* illustrates the 154,040 windows with different *S*:*N* ratios; a *S*:*N* > 9 is typical of that found in current STORM data). (iv) Windows in frame 30 after corruption with noise to reduce *S*:*N* to < 3 (*histogram*). (v) Mean projection of all corrupted windows; analogous to a 'noisy' wide-field image. (**b**) Reconstructed images prepared by passing all selected windows prior to noise corruption to the different algorithms (i.e., the tuned version of JD, plus CM, MLS, and MLE), compiling localization results, and then convolving localizations with a 20-nm Gaussian intensity profile to aid visualization (higher magnifications of the yellow box are shown below). The different approaches yield roughly equivalent images. At this high *S*:*N*, the MLE image (green boxes) should contain localizations with the least RMSE (Fig. 2), so we consider it to provide the truest representation of microtubule structure. (**c**) Intensity profile (the average intensity in arbitrary units, au, of a region 35 nm orthogonal to the line indicated) across lines *1* and *2* in (b). Generally, results given by JD and MLE are similar, as those returned by CM (*profile 1*) and MLS (*profile 2*) slightly diverge. (**d**) Reconstructed images prepared as in (b), but using the windows after corruption with noise. 142,728, 142,684, 142,275, and 142,669 localizations were returned by the tuned version of JD, CM, MLS, and MLE respectively. At this low *S*:*N* ratio, images prepared using the tuned version of JD (red boxes) return the least error (Fig. 3(c)), so we now consider them to be the truest representations. CM gives the 'fuzziest' image, presumably because it performs the worst at *S*:*N* >2. At the highest magnification (bottom row) – where the tuned version of JD, CM, MLS, and MLE yield 50.5, 50.9, 50.7, and 50.6 localizations per unit area, respectively – CM, MLS, and MLE probably return many mis-localizations (compare the number of isolated spots in the dark area at bottom-center in the bottom row). In the case of MLS and MLE, a failure to converge to a solution during the 200 iterations allowed is returned as a 'zero-result' in the upper-left corner of a window, and this generates a grid pattern after reconstruction that is especially obvious in the MLS image (*orange circles*). Such localizations are usually discarded, but have been retained here to allow fair comparison between a similar numbers of localizations from the different methods. (**e**) Intensity profiles across lines *1* and *2*. Profiles are slightly wider here, compared to those in (c). (**f**) Comparison of intensity profiles of line *2*. We expect MLE applied to uncorrupted windows to yield the truest representation (above); when using corrupted windows, the tuned version of JD returns a profile that is closer to the 'true' one (compared to MLS). (**g**) The fraction of nearest-neighbor distances (3-nm bins) given by all localizations obtained with the corrupted windows shown in the middle row of (d). Data using MLE with uncorrupted windows (which we assume gives the 'truest' representation; above), and for the same number of randomly-distributed spots, are included for comparison. Again, the tuned version of JD returns a distribution closest to the 'true' one. Results using MLS have been omitted, because many 'failed' localizations heavily skew the histogram.

Supplemental Figure 5. Shapes of functions used in equations (3) and (4).
Top: in equation (3), σ_i is tuned as a function of distance from the brightest pixel, d_i; the brightest pixel is at 0, and σ_i increases rapidly at ±3 to equal ∞ at ±4. *Bottom*: in equation (4), $g(x)$ is a piecewise Gaussian distribution function with a flat top (width 5 pixels) and Gaussian slopes that match the PSF. Applications of these two equations ensures that the distributions of photons from the brightest pixels are equal to the PSF, those from surrounding pixels become progressively wider the further away the pixels are, and those from distant pixels (i.e., > 3 σ_{PSF}) become infinitely wide.

JD Localization Software.

JDLoc_Test.m
This script demonstrates the use of JD_2D_tuned and JD_2D_optimized.

genSpot.m
A subroutine to generate an array of spot images to be used for localization. (Requires MATLAB Statistics Toolbox.)

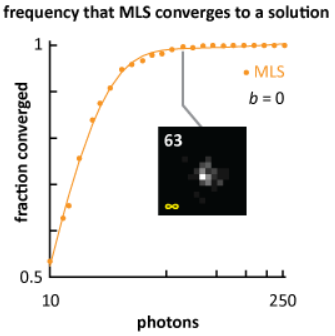
JD_2D_tuned.m
Single molecule localization using the Joint Distribution method tuned for maximum precision at low signal-to-noise ratios.

JD_2D_optimized.m
Single molecule localization using the Joint Distribution method optimized for speed and precision across a wide range of signal-to-noise ratios.

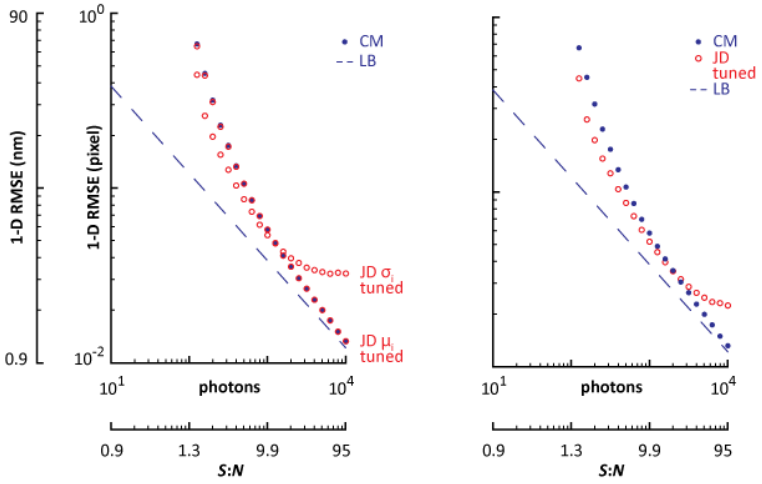
Instructions
Copy JDLoc_Test.m, genSpot.m, JD_2D_tuned.m, and JD_2D_optimized.m to your MATLAB directory. For a demonstration, run JDLoc_Test in MATLAB (MATLAB Statistics Toolbox is required). For more details type "help JDLoc_Test" in MATLAB. To localize your own image, type "help JD_2D_tuned" or "help JD_2D_optimized" in MATLAB and follow the instructions.

To incorporate Joint Distribution (JD) localization into your own script, all pertinent code is contained within JD_2D_tuned.m or JD_2D_optimized.m.

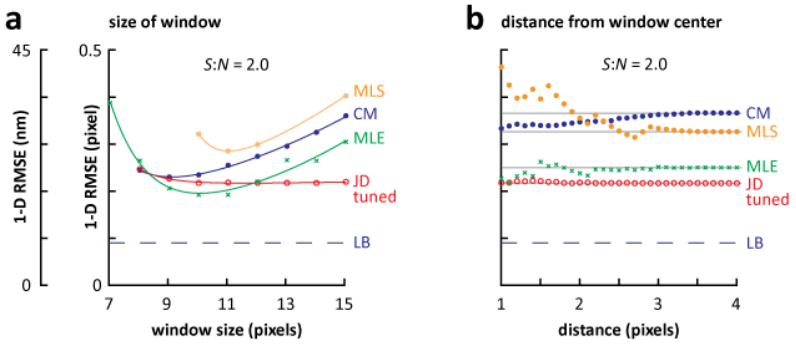
Supplemental Figure 1



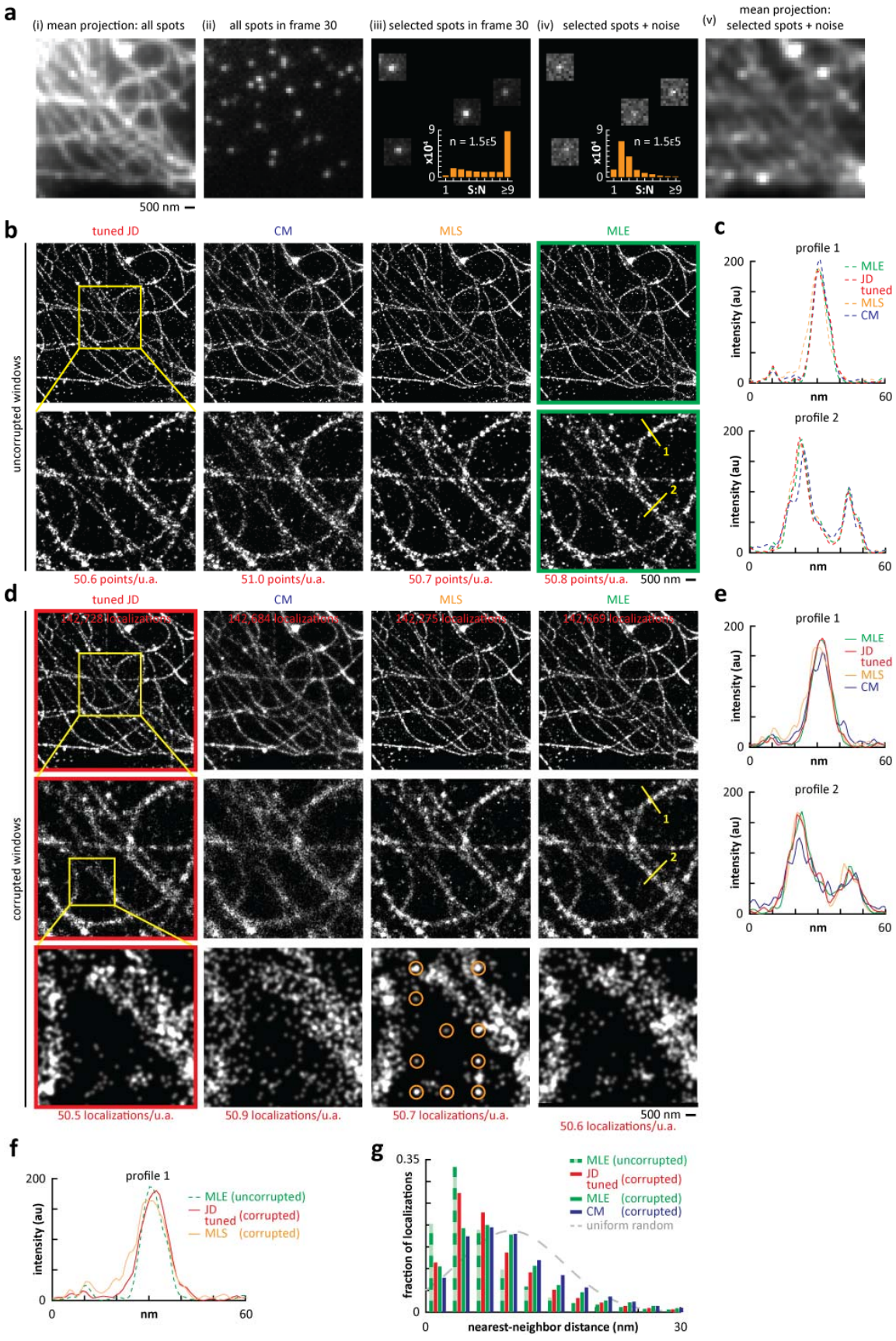
Supplemental Figure 2



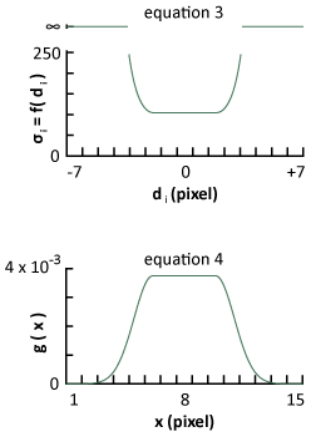
Supplemental Figure 3



Supplemental Figure 4



Supplemental Figure 5



```
%JDLoc_Test.m
%   This script demonstrates the use of JD_2D_tuned and JD_2D_optimized.
%
%   [] = JDLoc_Test ()
%
%   ------------------------------------------------------------------------
%   This code is free for academic use only. Please reference:
%       "A maximum precision closed-form solution for localizing
%       diffraction-limited spots in noisy images" Joshua D. Larkin and
%       Peter R. Cook, Optics Express (2012).
%
%   For commercial use please contact Joshua Larkin
%       email: joshlarkin at gmail.com
%
%   copyright Joshua D Larkin 2012
%   ------------------------------------------------------------------------
%
%   This code builds a stack of 2D images using subroutine genSpot, each of
%   which contains a single diffraction-limited spot. Spots in each image
%   are then localized serially using JD_2D_tuned and JD_2D_optimized. The
%   number of localizations per second is computed and reported. Derived
%   locations are compared to true locations, and the root-mean-square
%   error reported.
%
%   Note:
%       MATLAB Statistics Toolbox is required for image generation script
%
%   INPUTS:
%       None
%
%   OUTPUTS:
%       None

function JDLoc_Test


%--Generate spot images--

    %Variables
    Nspots = uint32(1000);    %number of spot images to generate in stack
    boxSize = uint8(15);      %dimension of each 2D image in pixels
    pixelSize = double(90);   %linear dimension of a square pixel in nm
    FWHMpsf = double(250);    %FWHM of microscope PSF in nm
    count = uint32(183);     %photon-event count per image
    noise = double(10);        %background noise in average photons/pixel

    [spotStack, spotLocs] = genSpot(Nspots, boxSize, pixelSize, FWHMpsf,...
                                    count, noise);
    disp(' ')
    disp('--')
    disp([int2str(Nspots) ' images generated, begining localization'])
    disp(' ')

    %Change psf FWHM to Sigma
    psf = FWHMpsf / 2.35;



%--Localize spots--
```

```
%   Tuned version of JD Localization
    mu1 = zeros(Nspots, 2); %center locations
    tic %timer start
    for i=1:Nspots
        mu1(i,:) = JD_2D_tuned(spotStack(:,:,i), pixelSize, psf);
    end
    t1=toc; %timer stop


%   Optimized version of JD Localization
    mu2 = zeros(Nspots, 2); %center locations
    tic %timer start
    for i=1:Nspots
        mu2(i,:) = JD_2D_optimized(spotStack(:,:,i), pixelSize, psf);
    end
    t2=toc; %timer stop



%--Compute Error--

    %root-mean-square error, components of each Cartesian coordinate
    rmse1Comp = sqrt(sum( (mu1 - double(spotLocs') ).^2)/double(Nspots));
    rmse2Comp = sqrt(sum( (mu2 - double(spotLocs') ).^2)/double(Nspots));
    %linear root-mean-square error, normalized to pixel size
    rmse1 = sqrt(rmse1Comp(1)^2 + rmse1Comp(2)^2) / pixelSize;
    rmse2 = sqrt(rmse2Comp(1)^2 + rmse2Comp(2)^2) / pixelSize;



%--Display Results--
    disp('Tuned JD:')
    disp(['  measured error (rmse) = ', num2str(rmse1,'%11.3g') ' pixels'])
    disp(['  computation rate = ' num2str(Nspots/t1) ' localizations/sec'])
    disp(' ')
    disp('Optimized JD:')
    disp(['  measured error (rmse) = ', num2str(rmse2,'%11.3g') ' pixels'])
    disp(['  computation rate = ' num2str(Nspots/t2) ' localizations/sec'])
    disp(' ')

end
```

```
%genSpot.m
%   A subroutine to generate an array of spot images to be used for
%   localization.
%
%   [spotStack, spotLocs] = genSpot (Nspots, boxSize, pixelSize, FWHMpsf,
%                                    count, noise)
%
%   ------------------------------------------------------------------------
%   This code is free for academic use only. Please reference:
%       "A maximum precision closed-form solution for localizing
%       diffraction-limited spots in noisy images" Joshua D. Larkin and
%       Peter R. Cook, Optics Express (2012).
%
%   For commercial use please contact Joshua Larkin
%       email: joshlarkin at gmail.com
%
%   copyright Joshua D Larkin 2012
%   ------------------------------------------------------------------------
%
%   This code builds a stack of 2D images, each of which contains a single
%   diffraction pattern. This is done by normally distributing 'photon'
%   coordinates in two dimensions about a randomly selected center, adding
%   uniformly distributed background noise, and binning photons into pixels
%   of specified size. The true center location of each spot is also
%   returned.
%
%   Note:
%       MATLAB Statistics Toolbox is required
%
%   INPUTS:
%       Nspots:     number of spot images to generate in stack (unit32)
%       boxSize:    dimension of each 2D image in pixels (uint8)
%       pixelSize:  linear dimension of a square pixel in nm (double)
%       FWHMpsf:    FWHM of microscope PSF in nm (double)
%       count:      photon-event count per image (uint32)
%       noise:      background noise in average photons/pixel (double)
%
%   OUTPUTS:
%       spotStack:  3-dimensional stack of computer generated spot images
%       spotLocs:   2xN array of true spot center locations in nm

function [spotStack, spotLocs] = genSpot (Nspots, boxSize, pixelSize,...
                                          FWHMpsf, count, noise)

%Reformat inputs
    %Change psf FWHM to Sigma
    psf = FWHMpsf / 2.35;

    %Image dimension in nm
    imDim = pixelSize * uint16(boxSize);

%Initialize Variables
    %Stack of images for output
    spotStack = zeros(boxSize, boxSize, Nspots, 'uint16');

    %array of true spot center locations [x,y]
    spotLocs = zeros(2, Nspots, 'uint16');

%Build Image Stack
```

```
for i=1:Nspots

    %--Generate photon observations--

        % randomize center location
        locMin = uint16(floor(imDim*0.33));
        locMax = uint16(ceil(imDim*0.67));
        xLoc = (locMin-1) + unidrnd(locMax-(locMin-1));
        yLoc = (locMin-1) + unidrnd(locMax-(locMin-1));

        % normal distribution of photon-events about center
        xCoords = normrnd(double(xLoc), psf, [1 count]);
        yCoords = normrnd(double(yLoc), psf, [1 count]);
            %note: it's possible for photons to be outside of image frame


    %--Generate background noise--

        % uniform distribution of background noise
        xNoise = unidrnd(double(imDim),...
                                    [1 uint16(double(boxSize)^2 * noise)]);
        yNoise = unidrnd(double(imDim),...
                                    [1 uint16(double(boxSize)^2 * noise)]);

        % add noise photon-events to spot photon-events
        d = [cat(2,xCoords,xNoise); cat(2,yCoords,yNoise)];

    %--Form histogram (pixels)--

        % create pixels by defining edges
        edges = cell(2,1);
        edges{1} = double(1:pixelSize:imDim);
        edges{2} = double(1:pixelSize:imDim);

        % populate pixels with photon-events
        n = coords2im(d, edges);

    %--Organize output--

        spotStack(:,:,i) = n;
        spotLocs(:,i) = [xLoc yLoc];

end

end

function [im] = coords2im(photonCoords, edges)
%subroutine to populate an image space, given photon coordinates
%
%Inputs:
%   photonCoords = [x vector; y vector] coordinates of photons in nm
%
%   edges = cell array of vectors containing the edge locations of pixels
%   in nm, where edges{1} = left edges of pixels in x-dimension, edges{2} =
%   top edges of pixels in y-dimension.
%
%Outputs:
%   im = output image
```

```
%

%Reformat inputs
    X = photonCoords(1,:);
    Y = photonCoords(2,:);

    Xbins = edges{1};
    Ybins = edges{2};

    XpixelSize = Xbins(2) - Xbins(1);
    YpixelSize = Ybins(2) - Ybins(1);

%Initialize variables
    im = zeros(size(Xbins,2), size(Ybins,2), 'uint16');

%Populate image with photon-events
    for i=1:size(photonCoords,2)

        %compute image coordinates from photon coordinates
        imX = ceil( X(i) / XpixelSize);
        imY = ceil( Y(i) / YpixelSize);

        %ensure photon is within image frame
        if ~any([imX imY] > size(im) | [imX imY] < [1 1])

            %place each photon in a pixel
            im(imY, imX) = im(imY, imX) + 1;

        end

    end

    %imshow(im,[]);

end
```

```
%JD_2D_tuned.m
%   Single molecule localization using the Joint Distribution method tuned
%   for maximum precision at low signal-to-noise ratios.
%
%   [mu0] = JD_2D_tuned(spotIm, pixelSize, psfSig)
%
%   ---------------------------------------------------------------------
%   This code is free for academic use only. Please reference:
%       "A maximum precision closed-form solution for localizing
%       diffraction-limited spots in noisy images" Joshua D. Larkin and
%       Peter R. Cook, Optics Express (2012).
%
%   For commercial use please contact Joshua Larkin
%       email: joshlarkin at gmail.com
%
%   copyright Joshua D Larkin 2012
%   ---------------------------------------------------------------------
%
%   This code performs the joint distribution localization of a
%   sub-diffraction sized particle imaged as a diffraction-limited spot
%   using a CCD array. If necessary, the image should be corrected for CCD
%   gain and offset such that unit pixel intensity represents a single
%   photo-electric event. The input is a single image of a single
%   diffraction-limited spot (isolated from a larger image of multiple
%   spots if necessary), assumed to contain a single fluorophore.
%   This 'tuned' version of JD localization has been optimized for spot
%   images with signal-to-noise ratios below 3, as defined in the
%   accompanying publication. Briefly, photons are shifted from pixel
%   centers by 1/4 of a pixel width, photon distributions from the
%   brightest pixel are set equal to the point-spread function while those
%   further away are progressively broadened. Photons too far from the
%   maximum intensity pixel to likely have come from the emitter of
%   interest are negated. Background noise in the image is estimated as the
%   mean intensity of peripheral pixels plus two standard deviations. The
%   influence on localization of photons thought to have come from
%   background is also negated.
%
%   INPUTS:
%       spotIm:     single spot image to localize (unit16)
%                   typical dimensions: 7x7 to 15x15 pixels (square only)
%       pixelSize:  linear dimension of square pixel in nm (double)
%       psfSigma:   sigma of a Gaussian function representing the
%                   microscope PSF in nm (double)
%
%   OUTPUTS:
%       mu:         [x,y] spot localization in nm, relative to upper-left
%                   corner of input image

function [mu0] = JD_2D_tuned(spotIm, pixelSize, psfSig)



%--Check Input Parameters--
    if size(spotIm,2) ~= size(spotIm,1)
        error('spot image must be square')
    end
    if ~isa(spotIm, 'uint16')
        spotIm = uint16(spotIm);
        warning('JDL:spotImClass',...
            'spotIm must be of class uint16; it has been converted')
```

```
    end
    if ~isa(pixelSize, 'double')
        pixelSize = double(pixelSize);
    end
    if ~isa(psfSig, 'double')
        psfSig = double(psfSigma);
    end



%--Determine Noise Level--
%   Sample outer two concentric rings of pixels (assuming these are
%   representative of uniform background noise in the image). Define
%   background as the mean value of these pixels plus two standard
%   deviations.

%   Interrogate pixel values at image perimeter, 2 pixels deep
        sizeX = size(spotIm,2);
        sizeY = size(spotIm,1);
        bottom = spotIm(1:2, 1:sizeX);
        top = spotIm(sizeY-1:sizeY, 1:sizeX);
        left = spotIm(3:sizeY-2, 1:2);
        right = spotIm(3:sizeY-2, sizeX-1:sizeX);

%   Define perimeter pixels as background
        bkgdValues = double(cat(2, bottom, top, left', right'));
        bkgdVector = bkgdValues(:);
        N = length(bkgdVector);

%   Compute background level as mean + 2 standard deviations
        meanBkgd = sum(bkgdVector) / N;
        stdBkgd = sqrt( sum( (bkgdVector-meanBkgd).^2 ) / (N-1) );
        bkgdLevel = ceil( meanBkgd + 2*stdBkgd );

%   Identify pixels with background noise
        ImBkgd0 = double(uint16(spotIm-bkgdLevel));



%--Shift photons from pixel centers--
%   Identify the brightest pixel and shift mu_i for all photons from other
%   pixels towards it by 1/4 of a pixel width in each dimension. For
%   photons from the brightest pixel, shift mu_i from the pixel center by a
%   value proportional to the intensities of adjacent pixels.

%   Determine pixel center values in nm (1 = x, 2 = y)
        centers(1,:) = pixelSize/2 : pixelSize : pixelSize*size(spotIm,2);
        centers(2,:) = pixelSize/2 : pixelSize : pixelSize*size(spotIm,1);

%   Determine location of max intensity pixel
        [~, Ix] = max(sum(ImBkgd0,1));
        [~, Iy] = max(sum(ImBkgd0,2));
        CM(1) = centers(1,Ix);
        CM(2) = centers(2,Iy);

%   Define shifted positions
        weightedCenters = zeros(size(centers));
        for i=1:2 %(1 = x, 2 = y)
```

```
            proj = sum(ImBkgd0,i); %project image onto one axis
            [~, maxI] = max(proj); %identify max intensity pixel
            j=1:length(centers); %independent variable along each axis

            %pixels left/above max pixel
            weightedCenters(i,:) = (centers(i,:) + pixelSize/4);

            %pixels right/below max pixel
            weightedCenters(i,j>maxI) = centers(i,j>maxI) - pixelSize/4;

            %max pixel
            if maxI>1 && maxI<length(centers) %ensure not at perimeter
                %scale center in max pixel by neighboring intensities
                leftInfluence = (proj(maxI) - proj(maxI-1)) / proj(maxI);
                scaledLeftInfl = (leftInfluence*pixelSize - pixelSize)/2;
                rightInfluence = (proj(maxI) - proj(maxI+1)) / proj(maxI);
                scaledRightInfl = (rightInfluence*pixelSize - pixelSize)/2;
                weightedCenters(i,maxI) = uint16(centers(i,maxI)+...
                    scaledLeftInfl-scaledRightInfl);
            else
                weightedCenters(i,maxI) = centers(i,maxI);
            end

        end



%--Assign Variable Distribution Widths-- (slow version)
%   Makes photon distributions narrower at max pixel and progressively
%   wider the further a pixels is from the max pixel. Because a finite (and
%   small for small images) number of possible solutions exist, to speed
%   things up all possible distribution/sigma functions can be defined once,
%   outside of this function, and passed to this function to be looked-up.

%   Initialize variables
        f = zeros(pixelSize*size(spotIm,2),2); %distribution function
        sf = zeros(pixelSize*size(spotIm,2),2); %sigma function
        So = 2.0 * pixelSize; %width of flat-top in distribution function
        for i=1:2 %(1 = x, 2 = y)

%   Create a piecewise Gaussian distribution function
            for x = 1:pixelSize*size(spotIm,2)
                if x < CM(i)-So
                    f(x,i) = 1/sqrt(2*pi*psfSig^2)*...
                                    exp(-(x-(CM(i)-So))^2/(2*psfSig^2));
                elseif x > CM(i)+So
                    f(x,i) = 1/sqrt(2*pi*psfSig^2)*...
                                    exp(-(x-(CM(i)+So))^2/(2*psfSig^2));
                else
                    f(x,i) = 1/sqrt(2*pi*psfSig^2);
                end
            end

%   Use distribution function to create sigma function
            for x = 1:pixelSize*size(spotIm,2)
                if x < CM(i)-3*psfSig,
                    sf(x,i)=Inf;
                elseif x > CM(i)+3*psfSig,
                    sf(x,i)=Inf;
```

```
                else
                    sf(x,i)=1/(2.5*f(x,i));
                end
            end

        end



%--Localize by Joint Distribution--
%   Involves projecting the image onto one dimension, forming a list of
%   photon coordinates, and assigning distributions for each photon-event;
%   then repeating for the second dimension and computing the center spot
%   location by as the joint of all independent distributions.

%   Initialize variables
        pe=cell(2,1); %photon-event list
        s=cell(2,1); %photon-event sigma list
        for i=1:2 %(1 = x, 2 = y)

%   Project image onto single axis
            projection = sum(spotIm,i); %(1 = x, 2 = y)
            bkgdProj = sum(ImBkgd0,i); %background pixels
            if i==2
                %rotate y-axis
                projection = projection';
                bkgdProj = bkgdProj';
            end

%   Build photon-event list
            pe{i} = zeros(1,sum(projection(:))); %photon-event list
            s{i} = Inf.*ones(1,sum(projection(:))); %set all sigma_i to Inf
            idx=1; %index of coordinates array

%   Convert pixel values into photon coordinates
            for j=1:size(projection,2)

%   Record mu_i
                nPhotons = projection(j); %number of photons to record
                if nPhotons>0
                    pe{i}(idx:idx+nPhotons-1) = weightedCenters(i,j)...
                                                    * ones(1,nPhotons);

%   Record sigma_i
                    if bkgdProj(j)>0
                        s{i}(idx:idx+bkgdProj(j)-1) = ...
                                    sf(uint16(weightedCenters(i,j)),i);
                    end

                end
                idx = idx + nPhotons; %increment index of coordinates array

            end

        end

%   Compute joint distribution
        photonEvents(1,:) = pe{1};
```

```
        photonEvents(2,:) = pe{2};
        photonDists(1,:) = s{1};
        photonDists(2,:) = s{2};
        sSquared = photonDists.^2;
        sSumInvSqr = sum(1./sSquared, 2);
        mu0 = sum(photonEvents./sSquared, 2) ./ sSumInvSqr;

end
```

```
%JD_2D_optimized.m
%   Single molecule localization using the Joint Distribution method
%   optimized for speed and precision across a wide range of signal-to-
%   noise ratios.
%
%   [mu0] = JD_2D_optimized(spotIm, pixelSize, psfSig)
%
%   ------------------------------------------------------------------------
%   This code is free for academic use only. Please reference:
%       "A maximum precision closed-form solution for localizing
%       diffraction-limited spots in noisy images" Joshua D. Larkin and
%       Peter R. Cook, Optics Express (2012).
%
%   For commercial use please contact Joshua Larkin
%       email: joshlarkin at gmail.com
%
%   copyright Joshua D Larkin 2012
%   ------------------------------------------------------------------------
%
%   This code performs the joint distribution localization of a
%   sub-diffraction sized particle imaged as a diffraction-limited spot
%   using a CCD array. If necessary, the image should be corrected for CCD
%   gain and offset such that unit pixel intensity represents a single
%   photo-electric event. The input is a single image of a single
%   diffraction-limited spot (isolated from a larger image of multiple
%   spots if necessary), assumed to contain a single fluorophore. This
%   'optimized' version of JD localization is precise across a wide range
%   of signal-to-noise ratios and is faster than the 'tuned' version.
%   Briefly, photon distributions are centered at pixel centers, and photon
%   distributions from the brightest pixels are set equal to the
%   point-spread function while those further away than 3*sigma_psf are
%   negated. Background noise in the image is estimated as the mean
%   intensity of peripheral pixels plus two standard deviations. The
%   influence on localization of photons thought to have come from
%   background is also negated.
%
%   INPUTS:
%       spotIm:     single spot image to localize (unit16)
%                   typical dimensions: 7x7 to 15x15 pixels (square only)
%       pixelSize:  linear dimension of square pixel in nm (double)
%       psfSig:     sigma of a Gaussian function representing the
%                   microscope PSF in nm (double)
%
%   OUTPUTS:
%       mu:         [x,y] spot localization in nm, relative to upper-left
%                   corner of input image

function [mu0] = JD_2D_optimized(spotIm, pixelSize, psfSig)



%--Check Input Parameters--
    if size(spotIm,2) ~= size(spotIm,1)
        error('spot image must be square')
    end
    if ~isa(spotIm, 'uint16')
        spotIm = uint16(spotIm);
        warning('JDL:spotImClass',...
            'spotIm must be of class uint16; it has been converted')
    end
```

```
    if ~isa(pixelSize, 'double')
        pixelSize = double(pixelSize);
    end
    if ~isa(psfSig, 'double')
        psfSig = double(psfSigma);
    end



%--Determine Noise Level--
%   Sample outer two concentric rings of pixels (assuming these are
%   representative of uniform background noise in the image). Define
%   background as the mean value of these pixels plus two standard
%   deviations.

%   Interrogate pixel values at image perimeter, 2 pixels deep
        sizeX = size(spotIm,2);
        sizeY = size(spotIm,1);
        bottom = spotIm(1:2, 1:sizeX);
        top = spotIm(sizeY-1:sizeY, 1:sizeX);
        left = spotIm(3:sizeY-2, 1:2);
        right = spotIm(3:sizeY-2, sizeX-1:sizeX);

%   Define perimeter pixels as background
        bkgdValues = double(cat(2, bottom, top, left', right'));
        bkgdVector = bkgdValues(:);
        N = length(bkgdVector);

%   Compute background level as mean + 2 standard deviations
        meanBkgd = sum(bkgdVector) / N;
        stdBkgd = sqrt( sum( (bkgdVector-meanBkgd).^2 ) / (N-1) );
        bkgdLevel = ceil( meanBkgd + 2*stdBkgd );

%   Identify pixels with background noise
        ImBkgd0 = double(uint16(spotIm-bkgdLevel));



%--Assign photons to pixel centers--
%   Identify the brightest pixel and set mu_i for all photons equal to the
%   centers of their respective pixels.

%   Determine pixel center values in nm (1 = x, 2 = y)
        centers(1,:) = pixelSize/2 : pixelSize : pixelSize*size(spotIm,2);
        centers(2,:) = pixelSize/2 : pixelSize : pixelSize*size(spotIm,1);

%   Determine location of max intensity pixel
        [~, Ix] = max(sum(ImBkgd0,1));
        [~, Iy] = max(sum(ImBkgd0,2));
        CM(1) = centers(1,Ix);
        CM(2) = centers(2,Iy);



%--Assign Variable Distribution Widths-- (slow version)
%   Makes photon distributions equal to the PSF near the max pixel and
%   infintiy more than 3 sigma_psf from the max pixel. Because a finite
%   (and small for small images) number of possible solutions exist, to
```

```
%   speed things up all possible sigma functions can be defined once,
%   outside of this function, and passed to this function to be looked-up.

%   Initialize variables
        sf = zeros(pixelSize*size(spotIm,2),2); %sigma function
        for i=1:2 %(1 = x, 2 = y)

%   Create sigma function
            for x = 1:pixelSize*size(spotIm,2)
                if x < CM(i)-3*psfSig,
                    sf(x,i)=Inf;
                elseif x > CM(i)+3*psfSig,
                    sf(x,i)=Inf;
                else
                    sf(x,i)=psfSig;
                end
            end

        end



%--Localize by Joint Distribution--
%   Involves projecting the image onto one dimension, forming a list of
%   photon coordinates, and assigning distributions for each photon-event;
%   then repeating for the second dimension and computing the center spot
%   location by as the joint of all independent distributions.

%   Initialize variables
        pe=cell(2,1); %photon-event list
        s=cell(2,1); %photon-event sigma list
        for i=1:2 %(1 = x, 2 = y)

%   Project image onto single axis
            projection = sum(spotIm,i); %(1 = x, 2 = y)
            bkgdProj = sum(ImBkgd0,i); %background pixels
            if i==2
                %rotate y-axis
                projection = projection';
                bkgdProj = bkgdProj';
            end

%   Build photon-event list
            pe{i} = zeros(1,sum(projection(:))); %photon-event list
            s{i} = Inf.*ones(1,sum(projection(:))); %set all sigma_i to Inf
            idx=1; %index of coordinates array

%   Convert pixel values into photon coordinates
            for j=1:size(projection,2)

%   Record mu_i
                nPhotons = projection(j); %number of photons to record
                if nPhotons>0
                    pe{i}(idx:idx+nPhotons-1) = centers(i,j)...
                                                * ones(1,nPhotons);

%   Record sigma_i
                    if bkgdProj(j)>0
```

```
                        s{i}(idx:idx+bkgdProj(j)-1) = ...
                                                sf(uint16(centers(i,j)),i);
                    end

                end
                idx = idx + nPhotons; %increment index of coordinates array

            end

        end

%   Compute joint distribution
        photonEvents(1,:) = pe{1};
        photonEvents(2,:) = pe{2};
        photonDists(1,:) = s{1};
        photonDists(2,:) = s{2};
        sSquared = photonDists.^2;
        sSumInvSqr = sum(1./sSquared, 2);
        mu0 = sum(photonEvents./sSquared, 2) ./ sSumInvSqr;

end
```