```
classdef cellTrajSim < handle
% cellTrajSim: Simulates a biased random walk representative of something
% that moves randomly by nature and is biased by an external cue to move in
% a particular direction.  An example is a cell undergoing chemotaxis.
%
% The biased random walker choose to change direction based on a Poissonian
% process with magnitude lambda [1/s].  When a reorientation decision is made,
% the orientation angle is drawn from a von Mises distribution with
% directionality factor, kappa.  The random walk is sampled every tSample
% [s] with an error factor representing centroid measurement error [um].
% The direction of external bias is set by angMean, and can change using
% the angMeanChgRate [1/s] variable.  The simulation time interval is tInt [s],
% and simulations continue until tEnd [s].
%
% Usage
% -----
%   cellTrajSim(lambda, kappa, speed, centMeasErr, angMean, angMeanChgRate, tInt,
%
% Arguments (input)
% -----------------
%   lambda - the average frequency of deciding to change direction [1/s]
%   kappa - directional factor (inverse width of von Mises Distribution)
%   speed - instanteous speed during random walk [um/s]
%   centMeasErr - centroid measurement error [um]
%   angMean - mean bias angle
%   angMeanChgRate - mean bias angle change rate [1/s]
%   tInt - simulation time interval [s]
%   tSample - sampling time interval [s]
%   tEnd - end simulation time [s]
%
% Public Methods List
% -------------------
%   simulateTraj();     % Determines simulation trajectory, coordinates saved to x
%                       % sampled coordinates with centroid measurement error are
%   calcSqDisp();       % Calculates squared displacement from xySampArray
%   fitTASDexp(fitTime); % fits TASD to exponential model, stopping fit a fitTime
%   fitTASDbeta(fitTime); % fits TASD to beta model, stopping fit a fitTime [s]
%   fitTASDbetaCME(fitTime); % fits TASD modified to acount for centroid measureme
%                            % to beta model, stopping fit a fitTime [s]
%
%
%
% Examples
% --------
%   sim = cellTrajSim   % Instantiates random walk simulation object with default
%
%   sim = cellTrajSim(lambda, kappa, speed, centMeasErr, angMean, angMeanChgRate,
%   sim.simulateTraj()
%                       % Instantiates random walk simulation object and simulates
%                       % trajectory with argument parameters
%
% Variables Returned
```

```
% ------------------
% All output variables are publicly accessable.  For example, to access
% the raw simulated trajectory (2D x-y coordinates) use,
%   simTraj sim.xyArray
%
% List of public object variables:
%   tArray -     N by 1 array of time coordinates, where N is the simulation
%                length, given by tEnd/tInt
%   xyArray -    N by 2 array of x-y coordinates
%   tSampArray - M by 1 array of sampled time coordinates, where M is
%                tEnd/tSample
%   xySampArray - M by 2 array of sampled coordinates.  Noise has been
%                 added to each sampled coordinate using a radial Gaussian
%                 with a width given by argument centMeasErr, which
%                 corresponds to real world centroid measurement error
%   angleArray - N by 1 array of the orientation of the cell at each timestep
%   turnsMAde -  N by 1 array tracking whether or not a decision to turn has
%                been made at each timestep
%
%
% Author: Alex J Loosley
% e-mail address: aloosley@brown.edu
% Release: 1
% Release date: 2/27/14

properties(GetAccess = 'public', SetAccess = 'private')

    % **Default Simulation Parameters**
    lambda = 0.5; % [1/s] Average number of pseudopod events per second
    speed = 0.2033; % [um/s]
    speedChgRate = 0.001 % [um/s]
    angMean = 0*(pi/180); % [rad] Overall direction [rad] (model invariant to choi
    angMeanChgRate = 0*(pi/180)/300; % [rad/s] Rate
    kappa = 2; % unitless factor, kappa, in Von Mises distribution

    centMeasErr = 2; %[um]

    % Simulation Discretization and Timing
    tInt = 1; % time interval [s]
    tSample = 10; % experimental sampling interval [s]
    tEnd = 12*60; % [s]

    % Tracking
    turnsMade
    tArray
    tSampArray
    xyArray
    xySampArray % trajectories sampled from simulation (i.e. every tSample = 10 se
    angleArray
    speedArray % new as of 02/09/2014

    % Tracking Event timing (Poisson Dist double checked)
    evDur % Event Duration Timer
    evDurOut % Formatted for output (zeros truncated off end)
```

```
% Dynamic Variables (spatial coordinates, time, walker orientation
% angle)
x
y
t
angle

% Indices
tIdx
tSampIdx
evIdx

% SD, TASD, MSD measures
% (SD = squared displacement, TA = time averaged)
sqDispTime
sqDisp

TAsqDisp
TAsqDisp_StE

sqDispExp
TAsqDispExp

% SD and TASD fitting and GOF
expFit
expFitGOF
expFitOutput
expFitRChi2
betaFit
betaFitGOF
betaFitOutput
betaFitRChi2
betaFit_dirTime
expFit_dirTime
expFit_asymp
expFit_asympOff

TASDexpFit
TASDexpFitGOF
TASDexpFitOutput
TASDexpFitRChi2
TASDbetaFit
TASDbetaFitGOF
TASDbetaFitOutput
TASDbetaFitRChi2
TASDbetaFit_dirTime
TASDexpFit_dirTime
TASDexpFit_asymp
TASDexpFit_asympOff

TASDbetaFitCME
TASDbetaFitCMEGOF
TASDbetaFitCMEOutput
```

```
        TASDbetaFitCMERChi2
        TASDbetaFitCME_dirTime

        % Other
        tempAngles % see comment below - work around for Von Mises distribution error
    end

    methods
        % Class Constructor
        function self = cellTrajSim(lambda, kappa, speed, centMeasErr, angMean, angMea
            if nargin > 0
                self.lambda = lambda;
                self.kappa = kappa;
                self.speed = speed;
                self.centMeasErr = centMeasErr;
                self.angMean = angMean;
                self.angMeanChgRate = angMeanChgRate;

                self.tInt = tInt;
                self.tSample = tSample;
                self.tEnd = tEnd;
            end

            self.turnsMade = zeros(round(self.tEnd/ self.tInt)+1,1);
            self.tArray = zeros(round(self.tEnd/ self.tInt)+1,1);
            self.xyArray = zeros(round(self.tEnd/ self.tInt)+1,2);
            self.angleArray = zeros(round(self.tEnd/ self.tInt)+1,1);
            self.xySampArray = zeros(round(self.tEnd/ self.tSample),2);
            self.speedArray = zeros(round(self.tEnd/ self.tSample),1);
        end
    end

    methods(Access = private)

        % Sets ICs to the origin of space and time (initial angle)
        function initialize(self)
            self.t = 0;
            self.x = 0;
            self.y = 0;
            self.angle = self.angMean;

            self.tArray(1) = self.t;
            self.xyArray(1,1) = self.x;
            self.xyArray(1,2) = self.y;
            self.angleArray = self.angle;
            self.speedArray = self.speed;

            self.tIdx = 1;
            self.tSampIdx = 1;
            self.evIdx = 1;
        end

        function addGaussNoiseXY(self)
            self.xySampArray(self.tSampIdx,1) = self.x + normrnd(0,self.centMeasErr);
```

```
            self.xySampArray(self.tSampIdx,2) = self.y + normrnd(0,self.centMeasErr);
            self.tSampIdx = self.tSampIdx + 1;
        end

        % trajectory Step based on a Poissonian process
        function trajStep(self)
            r = rand(1,1);
            if r <= self.lambda*self.tInt % CHANGE DIRECTION
                self.turnsMade(self.tIdx) = 1; % Notes that a turn was made
                self.evDur(self.evIdx) = self.evDur(self.evIdx) + self.tInt; % Minimum
                self.evIdx = self.evIdx + 1; % Initializes next event

                self.angMean = atan2(-self.y, 200-self.x);
                self.tempAngles = vmrand(self.angMean + self.angMeanChgRate*self.t, se
                self.angle = self.tempAngles(1); % I requested 2 random numbers at a t
                %
                % self.speed = random('exp', 0.000092*self.t + 0.2033);
                % check angle limit - not necessary using Von Mises distribution, "vmr
                %{
                chk = 1;
                while chk == 1
                    if abs(self.angle)> pi
                        tempAngles = normrnd(self.angMean + self.angMeanChgRate*self.t
                        self.angle = tempAngles(1); % I requested 2 random numbers at
                    else
                        chk = 0;
                    end
                end
                %}
                % self.angle = mod(self.angle+pi,2*pi)-pi; % sets angle between -pi an
                self.x = self.x + self.speed*cos(self.angle)*self.tInt;
                self.y = self.y + self.speed*sin(self.angle)*self.tInt;
                self.xyArray(self.tIdx+1,1) = self.x;
                self.xyArray(self.tIdx+1,2) = self.y;
                self.angleArray(self.tIdx+1) = self.angle;
                self.speedArray(self.tIdx+1) = self.speed;

            else % Otherwise DON'T CHANGE DIRECTION (or speed)
                self.evDur(self.evIdx) = self.evDur(self.evIdx) + self.tInt; % updates
                self.x = self.x + self.speed*cos(self.angle)*self.tInt;
                self.y = self.y + self.speed*sin(self.angle)*self.tInt;
                self.xyArray(self.tIdx+1,1) = self.x;
                self.xyArray(self.tIdx+1,2) = self.y;
                self.angleArray(self.tIdx+1) = self.angle;
                self.speedArray(self.tIdx+1) = self.speed;
            end

            % Update time counters
            self.t = self.t + self.tInt;
            self.tArray(self.tIdx + 1) = self.t;
            self.tIdx = self.tIdx + 1;
        end
    end
```

```
methods
    function simulateTraj(self)
        self.initialize()
        self.evDur = zeros(round(self.tEnd/self.tInt),1);


        while self.t < self.tEnd
            if mod(round(self.t/self.tInt),round(self.tSample/self.tInt)) == 0
                self.addGaussNoiseXY() %If the above is true, this samples the tra
            end

            self.trajStep()
        end

        self.evDurOut = self.evDur(1:find(self.evDur==0,1)-1);
        self.tSampArray = 0:self.tSample:self.tEnd;


    end

    % Square displacement (SD, TASD) and square displacement
    % exponent calculations
    function calcSqDisp(self)
        nSamp = size(self.xySampArray,1);
        self.sqDispTime = (self.tSample : self.tSample : (nSamp-1)*self.tSample)';

        % Square Displacement, reiterate simulation to get
        % ensemble average
        self.sqDisp = NaN(nSamp-1,1);
        for k_int = 1:nSamp - 1
            xd = self.xySampArray(1+k_int,1)-self.xySampArray(1,1);
            yd = self.xySampArray(1+k_int,2)-self.xySampArray(1,2);
            self.sqDisp(k_int) = xd^2 + yd^2;
        end

        % Time Averaged Square Displacement
        self.TAsqDisp = NaN(nSamp-1,1);
        self.TAsqDisp_StE = NaN(nSamp-1,1);
        for k_int = 1:nSamp-1
            rSq = NaN(nSamp-k_int,1);
            for k =1:nSamp-k_int
                xd = self.xySampArray(k+k_int,1)-self.xySampArray(k,1);
                yd = self.xySampArray(k+k_int,2)-self.xySampArray(k,2);
                rSq(k)=xd^2+yd^2;
            end
            self.TAsqDisp(k_int)=nanmean(rSq);
            self.TAsqDisp_StE(k_int)=nanste(rSq);
        end

        % meanSqDisp Exponent calculated by taking the forward
        % difference derivative of MSD vs time in loglog space

        self.sqDispExp = NaN(size(self.sqDisp,1)-1,1);
        self.TAsqDispExp = NaN(size(self.TAsqDisp,1)-1,1);
```

```
        for kk = 1:size(self.sqDisp,1)-1
            self.sqDispExp(kk) = log(self.sqDisp(kk+1)/self.sqDisp(kk)) / log((kk+
            self.TAsqDispExp(kk) = log(self.TAsqDisp(kk+1)/self.TAsqDisp(kk)) / lo
        end
    end


    % Fits the simulated MSD using data points up to time tEndFit (seconds)
    function fitTASDexp(self,tEndFit)
        model = fittype('a-b*exp(-time/dirTime)','independent','time','coefficient
        Value = 'NonlinearLeastSquares';
        opts = fitoptions('method', Value);
        opts.StartPoint=[2,1,20];
        opts.Upper = [2.5,2.5,tEndFit];
        opts.Lower = [1.2,0.1,1 ];
        [self.TASDexpFit, self.TASDexpFitGOF, self.TASDexpFitOutput] =...
            fit(self.sqDispTime(1:round(tEndFit/self.tSample)), self.TAsqDispExp(1
        self.TASDexpFit_dirTime = self.TASDexpFit.dirTime;
        self.TASDexpFit_asymp = self.TASDexpFit.a;
        self.TASDexpFit_asympOff = self.TASDexpFit.b;
        self.TASDexpFitRChi2 = self.TASDexpFitGOF.rmse^2;
    end

    function fitTASDbeta(self,tEndFit)
        model = fittype('(1+2*(time/dirTime))/(1+(time/dirTime))','independent','t
        Value = 'NonlinearLeastSquares';
        opts = fitoptions('method', Value);
        opts.StartPoint= [20];
        opts.Upper = [tEndFit];
        opts.Lower = [1 ];
        [self.TASDbetaFit, self.TASDbetaFitGOF, self.TASDbetaFitOutput] =...
            fit(self.sqDispTime(1:round(tEndFit/self.tSample)), self.TAsqDispExp(1
        self.TASDbetaFit_dirTime = self.TASDbetaFit.dirTime;
        self.TASDbetaFitRChi2 = self.TASDbetaFitGOF.rmse^2;
    end

    function fitTASDbetaCME(self,tEndFit)
        model = fittype('(1+2*(time/dirTime))/(1+(time/dirTime))','independent','t
        Value = 'NonlinearLeastSquares';
        opts = fitoptions('method', Value);
        opts.StartPoint= [20];
        opts.Upper = [tEndFit];
        opts.Lower = [1 ];
        [self.TASDbetaFitCME, self.TASDbetaFitCMEGOF, self.TASDbetaFitCMEOutput] =
            fit(self.sqDispTime(1:round(tEndFit/self.tSample)),...
            self.TAsqDispExp(1:round(tEndFit/self.tSample)).*(self.TAsqDisp(1:roun
            model, opts);
        self.TASDbetaFitCME_dirTime = self.TASDbetaFitCME.dirTime;
        self.TASDbetaFitCMERChi2 = self.TASDbetaFitCMEGOF.rmse^2;

    end

    function fitExpIn(self,tIn,yIn,tEndFit)
```

```
            model = fittype('a-b*exp(-time/dirTime)','independent','time','coefficient
            Value = 'NonlinearLeastSquares';
            opts = fitoptions('method', Value);
            opts.StartPoint=[2,1,20];
            opts.Upper = [2.5,2.5,tEndFit];
            opts.Lower = [1.2,0.1,1 ];
            [self.expFit, self.expFitGOF, self.expFitOutput] =...
                fit(tIn(1:round(tEndFit/self.tSample)), yIn(1:round(tEndFit/self.tSamp
            self.expFit_dirTime = self.expFit.dirTime;
            self.expFit_asymp = self.expFit.a;
            self.expFit_asympOff = self.expFit.b;
            self.expFitRChi2 = self.expFitGOF.rmse^2;
        end

        function fitBetaIn(self,tIn,yIn,tEndFit)
            model = fittype('(1+2*(time/dirTime))/(1+(time/dirTime))','independent','t
            Value = 'NonlinearLeastSquares';
            opts = fitoptions('method', Value);
            opts.StartPoint=[20];
            opts.Upper = [tEndFit];
            opts.Lower = [1 ];
            [self.betaFit, self.betaFitGOF, self.betaFitOutput] =...
                fit(tIn(1:round(tEndFit/self.tSample)), yIn(1:round(tEndFit/self.tSamp
            self.betaFit_dirTime = self.betaFit.dirTime;
            self.betaFitRChi2 = self.betaFitGOF.rmse^2;
        end
    end
end
```

Published with MATLAB® 8.0