```
% function [Position Time EventTime EventPosition] =↙
SimulateTracksTwoDRunStopConstantSpeedVonMisesAngle...
%     (NumTracks, Dt, TimeTotal, TauR, TauS, Speed, AngularDeviation)
%
% Simulate tracks from a 2D CRW with constant speed and von Mises turning
% angle.  Inputs:
%
% NumTracks: the number of tracks to simulate
% Dt: the sampling interval time
% TimeTotal:  the total duration of each simulated track
% TauR: Mean duration of a run
% TauS: Mean duration of a stop (set to 0 for run-only process)
% AngularDeviation: The angular deviation of the underlying Von Mises
% turning angle pdf
%
% Outputs:
%
% Position: Cell of length NumTracks.  Each element is a 2 x M matrix containing
% the position coordinates of particles at times corresponding to the M
% entries of Time
% Time: vector length M, containing the sampling times
% EventTime: Cell of length NumTracks.  Each element contains a vector of
% times at which reorientations occur
% EventPosition: Cell of length NumTracks.  Each element contains a 2xN matrix
% containing the position coordinates at which jumps occur (where N is the
% number of reorientations in that track)
%

function [Position, Time, EventTime, EventPosition] =↙
SimulateTracksTwoDRunStopConstantSpeedVonMisesAngle...
    (NumTracks, Dt, TimeTotal, TauR, TauS, Speed, AngularDeviation)

if TauS==0
    bStops = false;
else
    bStops = true;
end

% convert angular deviation to Von Mises parameter Kappa
Kappa = AngularDeviationToKappa(AngularDeviation);

Time = 0:Dt:TimeTotal;

% estimate number of VJ events in advance
EstimatedNumJumps = 5*min(ceil(TimeTotal/TauR), 1e3); % deliberate overestimate with↙
upper lim

% initialise data containers
Position = cell(NumTracks, 1);
EventPosition = Position;
```

```
EventTime = Position;

% initialise particle status: all running initially in random directions
ThetaInit = VonMises2D(Kappa, NumTracks);

% loop over particles

for i=1:NumTracks

    bLoop = true;
    Count = 1;
    TimeTally = 0;
    Status = true;

    EventTime{i} = zeros(EstimatedNumJumps, 1);
    Lag = zeros(EstimatedNumJumps, 1);

    while bLoop

        if Status
            Lag(Count) = exprnd(TauR);
        else
            Lag(Count) = exprnd(TauS);
        end

        if bStops
        Status = ~Status;
        end

        TimeTally = TimeTally + Lag(Count);
        Count = Count+1;
        EventTime{i}(Count) = TimeTally;

        if TimeTally>TimeTotal
            bLoop = false;
        end

    end

    EventTime{i} = EventTime{i}(1:Count);
    Lag = Lag(1:Count-1);
    EventPosition{i} = zeros(2, Count);

    % repeat loop and fill in position

    Status = true;
    Theta = ThetaInit(i);
%     Lag = [EventTime{i}(2); diff(EventTime{i}(2:end))];

    for j=2:Count
```

```
        if Status % running

            EventPosition{i}(1,j) = EventPosition{i}(1,j-1)+Speed*Lag(j-1)*cos(Theta);
            EventPosition{i}(2,j) = EventPosition{i}(2,j-1)+Speed*Lag(j-1)*sin(Theta);

        else % stopped

            EventPosition{i}(:,j) = EventPosition{i}(:,j-1);
            % new speed and angle
            Theta = Theta + VonMises2D(Kappa);

        end

        Status = ~Status;

    end


    % compute interpolated position

    Position{i}(1,:) = interp1q(EventTime{i}, EventPosition{i}(1,:)', Time');
    Position{i}(2,:) = interp1q(EventTime{i}, EventPosition{i}(2,:)', Time');

    % remove initial zero and final out-of-bounds measurement from events

    EventTime{i} = EventTime{i}(2:end-1);
    EventPosition{i} = EventPosition{i}(:, 2:end-1);


end

end

% function Kappa = AngularDeviationToKappa(Sigma, bMethod)
%
% Created by GR 5/3/13
% Given supplied value(s) of the angular deviation, compute the equivalent
% kappa parameter for the von mises distribution.
%
% Inputs:
% Sigma - input angular deviation values
% bMethod - Bool switch for definition of ang dev:
% 0 sigma = sqrt{-2*log{rho_1}} [undefined for uniform dist]
% 1 sigma = sqrt{2*(1-rho_1)} [sqrt(2) for uniform dist]
%
% Outputs:
% Kappa - same shape as Sigma, containing equivalent kappa values

function Kappa = AngularDeviationToKappa(Sigma)

MaxKappa = 200; % beyond this value, cannot discern the value unambigously
```

```
RHS = 1-(Sigma.^2)/2;

% use minimiser to find corresponding kappa values

minsearch_fun = @(x) besseli(1,x)./besseli(0,x);

Kappa = zeros(size(Sigma));

for i=1:numel(Sigma)

    Kappa(i) = fminbnd(@(x) abs(minsearch_fun(x)-RHS(i)), 0, MaxKappa );

end

end

% function Theta = VonMises2D(Kappa, N)
%
% Draw N random variables from the Von Mises circular distribution with mean
% Mu and width parameter Kappa.  Each entry in Theta has a value between
% -pi and pi.  If N arg is missing, assume only one variable required.

function Theta = VonMises2D(Kappa, N)

if nargin==1
    N=1;
end

% quick sanity check on Kappa - if it is equal to zero then just use a
% random distribution on a circle

if Kappa==0

    Theta = 2*pi*rand(N,1) - pi;
    return;

end

t = 1 + sqrt(1+4*Kappa^2);
p = (t - sqrt(2*t)) / (2*Kappa);
r = (1+p^2)/(2*p);

% init Theta output vector

Theta = zeros(N, 1);

for i=1:N

    % bool flag to show when value is accepted
```

```
    bIter = true;

    while bIter

        u = rand(3,1);

        z = cos( pi*u(1) );
        f = (1+r*z)/(r+z);
        c = Kappa*(r-f);

        if ( (c*(2-c)-u(2)) > 0 )
            % ACCEPT
            bIter = false;
        elseif ( ( log( c/u(2) )+1-c ) > 0 )
            % ACCEPT
            bIter = false;
        end

    end


    Theta(i) = sign(u(3)-0.5) * acos(f);

end


end
```