```
function [xfp, Pfp, Xp, Wp] = pfFilter(x_0, P_0, Y, proc_f, proc_Q,
 meas_h, meas_R, N, bResample, plotFunc)
%PFFILTER Filters measurements Y using the SIS or SIR algorithms and a
% state-space model.
%
% Input:
%   x_0         [n x 1] Prior mean
%   P_0         [n x n] Prior covariance
%   Y           [m x K] Measurement sequence to be filtered
%   proc_f      Handle for process function f(x_k-1)
%   proc_Q      [n x n] process noise covariance
%   meas_h      Handle for measurement model function h(x_k)
%   meas_R      [m x m] measurement noise covariance
%   N           Number of particles
%   bResample   boolean false - no resampling, true - resampling
%   plotFunc    Handle for plot function that is called when a filter
%               recursion has finished.
% Output:
%   xfp         [n x K] Posterior means of particle filter
%   Pfp         [n x n x K] Posterior error covariances of particle
 filter
%   Xp          [n x N x K] Particles for posterior state distribution
 in times 1:K
%   Wp          [N x K] Non-resampled weights for posterior state x in
 times 1:K

% Your code here, please.
% If you want to be a bit fancy, then only store and output the
 particles if the function
% is called with more than 2 output arguments.


n = size(P_0,1);
K = size(Y,2);

% Pre allocate
xfp = [];
Pfp = zeros(n,n,K);
Xp = zeros(n,N,K);
Wp = [];
j = [];

% Draw the first particles for t=0
X = x_0 + mvnrnd(zeros(size(x_0)), P_0, N).';
W = ones(1, size(X,2)) / K;

% Do the filtering
for i = 1:K
    % Filter next step
    Xmin1 = X;
    [X, W] = pfFilterStep(X, W, Y(:,i), proc_f, proc_Q, meas_h,
 meas_R);
```

```
    if i > 1
        plotFunc(i, X, Xmin1, W, j);
    end

    % Update the outputs
    Wp = [Wp W.'];
    Xp(:,:,i) = X;
    xfp = [xfp X*W.'];
    Pfp(:,:,i) = (X - X*W.') * ((X - X*W.').' .* W.');

    if bResample
        [X, W, j] = resampl(X, W);
    else
        j = 1:size(X,2);
    end

end


end

function [X_k, W_k] = pfFilterStep(X_kmin1, W_kmin1, yk, proc_f,
 proc_Q, meas_h, meas_R)
%PFFILTERSTEP Compute one filter step of a SIS/SIR particle filter.
%
% Input:
%   X_kmin1     [n x N] Particles for state x in time k-1
%   W_kmin1     [1 x N] Weights for state x in time k-1
%   y_k         [m x 1] Measurement vector for time k
%   proc_f      Handle for process function f(x_k-1)
%   proc_Q      [n x n] process noise covariance
%   meas_h      Handle for measurement model function h(x_k)
%   meas_R      [m x m] measurement noise covariance
%
% Output:
%   X_k         [n x N] Particles for state x in time k
%   W_k         [1 x N] Weights for state x in time k

% Your code here!
% Sample q
X_k = proc_f(X_kmin1) + mvnrnd(zeros(size(X_kmin1, 1), 1) , (proc_Q),
 size(X_kmin1, 2)).';

% Calculate the weights
W_k = zeros(size(W_kmin1));
for i = 1:size(X_kmin1, 2)
   %W_k(i) = W_kmin1(i) * normpdf(yk, meas_h(X_k(:, i)),
 sqrtm(meas_R));
    W_k(i) = W_kmin1(i) * mvnpdf(yk, meas_h(X_k(:, i)), meas_R);
end

% Normalize the weights
W_k = W_k / sum(W_k);
```

```
end

function [Xr, Wr, j] = resampl(X, W)
%RESAMPLE Resample particles and output new particles and weights.
% resampled particles.
%
%   if old particle vector is x, new particles x_new is computed as
 x(:,j)
%
% Input:
%   X   [n x N] Particles, each column is a particle.
%   W   [1 x N] Weights, corresponding to the samples
%
% Output:
%   Xr  [n x N] Resampled particles, each corresponding to some
 particle
%               from old weights.
%   Wr  [1 x N] New weights for the resampled particles.
%   j   [1 x N] vector of indices refering to vector of old particles

% Your code here!
% Normalise the weights and calculate the cumsum or the portions of 0-
>1 they own
W = W/(sum(W));
W = cumsum(W);

% Generate random places to take samples and sort them so that the
 looping is minimized
u = rand(size(W));
u = sort(u);

j = [];
Xr = [];

% Do the resampling
prev_index = -1;
next_index = 1;
for i = 1:size(u,2)
    while 0 < 1
        % If the first place is between 0 and the first weight
        if prev_index == -1 && u(i) < W(1)
            j = [1 j];
            Xr = [Xr X(:, 1)];
            break

        % If the place is inside the portion of 0->1 the weight own
        elseif prev_index ~= -1 && u(i) >= W(prev_index) && u(i) <
 W(next_index)
            j = [next_index j];
            Xr = [X(:, next_index) Xr];
            break

        % The place was not in any place a weight owned, check the
 next weight
```

```
        else
            prev_index = next_index;
            next_index = next_index+1;
        end

    end
end

% All the samples are equally plausible
Wr = ones(1, size(W,2))/size(X,2);


end

function [xs, Ps, xf, Pf, xp, Pp] = nonLinRTSsmoother(Y, x_0, P_0, f,
 T, Q, S, h, R, sigmaPoints, type)
%NONLINRTSSMOOTHER Filters measurement sequence Y using a
% non-linear Kalman filter.
%
%Input:
%   Y           [m x N] Measurement sequence for times 1,...,N
%   x_0         [n x 1] Prior mean for time 0
%   P_0         [n x n] Prior covariance
%   f                   Motion model function handle
%   T                   Sampling time
%   Q           [n x n] Process noise covariance
%   S           [n x N] Sensor position vector sequence
%   h                   Measurement model function handle
%   R           [n x n] Measurement noise covariance
%   sigmaPoints Handle to function that generates sigma points.
%   type        String that specifies type of non-linear filter/
smoother
%
%Output:
%   xf          [n x N]     Filtered estimates for times 1,...,N
%   Pf          [n x n x N] Filter error convariance
%   xp          [n x N]     Predicted estimates for times 1,...,N
%   Pp          [n x n x N] Filter error convariance
%   xs          [n x N]     Smoothed estimates for times 1,...,N
%   Ps          [n x n x N] Smoothing error convariance

% your code here!
% We have offered you functions that do the non-linear Kalman
 prediction and update steps.
% Call the functions using
% [xPred, PPred] = nonLinKFprediction(x_0, P_0, f, T, Q, sigmaPoints,
 type);
% [xf, Pf] = nonLinKFupdate(xPred, PPred, Y, S, h, R, sigmaPoints,
 type);



N = size(Y,2);
```

```
n = length(x_0);
m = size(Y,1);

% Data allocation
Pp = zeros(n,n,N);
Pf = zeros(n,n,N);
Ps = zeros(n,n,N);

% Start with going forward
xp = [];
xf = [];
x = x_0;
p = P_0;
 for i = 1:size(Y,2)
    % Predict one step ahead in time.
    [x, p] = nonLinKFprediction(x, p, f, T, Q, sigmaPoints, type);
    xp = [xp x];
    Pp(:,:,i) = p;

    % Update the estimated position with the measurement
    [x, p] = nonLinKFupdate(x, p, Y(:, i), S(:,i), h, R, sigmaPoints,
 type);
    xf = [xf x];
    Pf(:,:,i) = p;
end
% Done with forward


% Start going backwards
% The last smoothed state is the same as the last filtered state
xs = xf(:,end);
Ps(:,:,end) = Pf(:,:,end);
for i = N-1:-1:1 % Smooth backwards one state at a time
    [x_smoothed, P_smoothed] = nonLinRTSSupdate(xs(:,1), Ps(:,:,i+1),
 xf(:,i), Pf(:,:,i), xp(:,i+1), Pp(:,:,i+1), f, T, sigmaPoints, type);

    xs = [x_smoothed xs];
    Ps(:,:,i) = P_smoothed;
end

end

function [xs, Ps] = nonLinRTSSupdate(xs_kplus1, Ps_kplus1,  xf_k,
 Pf_k, xp_kplus1, Pp_kplus1, f, T, sigmaPoints, type)
%NONLINRTSSUPDATE Calculates mean and covariance of smoothed state
% density, using a non-linear Gaussian model.
%
%Input:
%   xs_kplus1   Smooting estimate for state at time k+1
%   Ps_kplus1   Smoothing error covariance for state at time k+1
%   xf_k        Filter estimate for state at time k
%   Pf_k        Filter error covariance for state at time k
%   xp_kplus1   Prediction estimate for state at time k+1
%   Pp_kplus1   Prediction error covariance for state at time k+1
```

```
%   f           Motion model function handle
%   T           Sampling time
%   sigmaPoints Handle to function that generates sigma points.
%   type        String that specifies type of non-linear filter/
smoother
%
%Output:
%   xs          Smoothed estimate of state at time k
%   Ps          Smoothed error convariance for state at time k

% Your code here.
if type == "EKF"
    % Calculate the differentiation of the state in xf_k
    [x_kp1, dx] = f(xf_k,T);

    % Calculate the smoothed state and smoothed covariance
    G = Pf_k * dx.' * inv(Pp_kplus1);
    xs = xf_k + G*(xs_kplus1 - xp_kplus1);
    Ps = Pf_k - G*(Pp_kplus1 - Ps_kplus1)*G.';

elseif type == "UKF"  | type == "CKF"
    % Calculate the differentiation of the state in xf_k
    [sp, W] = sigmaPoints(xf_k, Pf_k, type);
    %x_kp1 = f(sp, T)*W.';

    P_kkp1 = zeros(size(Pp_kplus1));
    for i = 1 : size(W,2)
        P_kkp1 = P_kkp1 + (sp(:,i)-xf_k)*(f(sp(:,i), T)-
xp_kplus1).'*W(i);
    end

    % Calculate the smoothed state and smoothed covariance
    G = P_kkp1*inv(Pp_kplus1);
    xs = xf_k + G*(xs_kplus1 - xp_kplus1);
    Ps = Pf_k - G*(Pp_kplus1 - Ps_kplus1)*G.';


end


end
```

Published with MATLAB® R2020a